Umple User Manual

Pdf version
Release 1.38.0, October 14, 2026

CRuiSE Research Group
School of Electrical Engineering and Computer Science
University of Ottawa, K1N 6N5 Canada

Project leader: Dr. Timothy Lethbridge timothy.lethbridge@uottawa.ca

http://www.umple.org

Umple (Lethbridge et al. 2021, CRuiSE group, 2026) is a tool for creating models of software and generating code in Java, Python, PHP, C++ and other languages. Umple's language is textual, but many kinds of diagrams and other outputs can be generated and used to help edit the text; these including state machines, state tables, class diagrams, feature diagrams, entity-relationship diagrams, message sequence tables, and instance diagrams. Umple allows embedding of several requirements formats such as user stories and use cases; Umple helps users understand the relationships (traceability) from the requirements to models and code, and vice-versa.

Umple can be used online, as a command-line tool or in many integrated development tools (IDEs) such as VSCode, Zed, Cursor or BBEdit. These tools use Umple's Language Server Protocol (LSP) capabilities to help people edit their text. Developers can make extensive use of the modeling notations (data models, state models), but also embed code in target languages such as Java or Python to complete their system. This enables direct generation of systems from the Umple textual source.

Umple's online tools enable users to use artificial intelligence (AI) to build systems reliably: Users pick a set of requirements in Umple and ask any AI to build Umple models, that can be deterministically transformed into code.

This document is the pdf version of the Umple user manual as of version 1.38.0 (October 2025) and supersedes the 2021 version (CRuiSE group, 2021) for version 1.30.2. Umple has been extensively improved since then, with numerous new capabilities. This version of the manual has over 890 pages and embeds over 550 examples.

Most people will use the online html version of the manual, available at https://manual.umple.org, however some people prefer a printable or citable document. This document has the same content as the online version, and links will take readers to the online manual. All examples in the manual have links that open the example code in classic UmpleOnline at https://try.umple.org or a newer experience at https://newumpleonline.umple.org.

Umple is an open-source project, developed by over 100 students at the University of Ottawa and elsewhere since 2007. The Umple compiler is written in itself. Umple is hosted on Github, at http://umple.org/code. Umple is targeted at open-source developers, so they can use it in their textual toolchain and develop code faster. It is also targeted at educators who want to teach modeling, as well as to enable students to create large, reliable, and complete executable systems from models, something that is more complex when using other modeling tools.

Umple is also a research platform for studying software modeling, programming language technology, agile software development, AI in software engineering, and user experience. There are numerous scientific papers published about Umple. The definitive paper to cite is by Lethbridge, Forward et al, (2021); that paper cites other relevant papers. A set of publications can be found at https://github.com/umple/umple/wiki/Publications .

Umple is supported by NSERC and is hosted by the Digital Research Alliance of Canada.

References

Lethbridge, TC., Forward, A., Badreddin, O., Brestovansky, D., Garzon, M., Aljamaan, H., Eid, S., Husseini Orabi, A., Husseini Orabi, M.,Abdelzad, V., Adesina, O., Alghamdi, A., Algablan, A., Zakariapour, A. (2021) Umple: Model-Driven Development for Open Source and Education, Science of Computer Programming, Elsevier, https://doi.org/10.1016/j.scico.2021.102665

CRuiSE Group, University of Ottawa, (2026) Umple Version 1.38.0, DOI: https://doi.org/10.5281/zenodo.4677562 https://zenodo.org/records/23354925

CRuiSE Group, University of Ottawa (2021) Umple User Manual, Version 1.30.1, https://doi.org/10.20381/ruor-26266

Getting Started

Umple is a technology for Model-Oriented Programming.

The Umple home page is https://umple.org. To download Umple for the command line or an IDE, go to https://umple.org/dl. To use Umple online as a web app, go to https://try.umple.org for the classic version or to https://newumpleonline.umple.org for version 2.

Umple allows you to do the following things:

1. **Create models textually**. Rather than drawing diagrams, it tends to be faster to create class diagrams, state machines and other types of models using Umple's textual format that looks just like programming-language code. Editing, comparing and many other tasks can often be done faster using this textual form. Umple has tools that allow you to edit the model textually (and see changes appear in a diagram) or to edit a diagram (and see changes appear in the textual code)

2. **Add modeling constructs directly into your programs, when you are programming in Java, Python, PHP, C++ or Ruby**. For example, when programming in Java, you can directly add a state machine that conforms with UML semantics into your code. This can save you a lot of coding. The Umple compiler acts as a pre-processor, first compiling the state machine to Java, and then compiling the Java into an executable program. Java methods found in the Umple code will be unchanged in the generated output. The same works for languages like Python and C++, although methods have to be tagged with the relevant language. See the Hello World examples to see how to do this.

3. **Specify requirements and have AI generate models**. Umple allows you to specify requirements in many ways. You can simply refer to existing external requirements, or embed them directly as files in the same repository with your Umple code. Umple allows you to specify requirements as user stories, use cases and other formats. Then you can use artificial intelligence to generate Umple models such as data models (class diagrams) or state models. They key idea is that rather than getting AI to go from requirements to code, you ask it to go from requirements to Umple (with embedded code for certain capabilities), and then have Umple generate the final system. This results in deterministic code that is more consistent as changes are made.

4. **Generate high quality code from models**. The Umple compiler creates code in languages like Java, Python, C++ and PHP that is of top quality. It is a goal of the Umple team to create the best open-source code-

generation tool available. The generated code never needs to be edited and never should be edited, since you can always embed native methods (i.e. Java, Php, etc.) in the Umple, or else use Umple's aspect oriented capabilities to alter the effects of generated code. As a result the concept of 'round tripping' is obsolete in Umple. You should treat Umple-generated code just like you would treat bytecode-or machine code, i.e. as a development artifact that can be thrown away and recreated. Nevertheless, we have endeavoured to make the generated code as clean and readable as possible so you can verify its correctness and learn how it works.

5. **Incrementally Umplify a program.** A program in a base language like Java should pass through the Umple compiler unchanged. As a result you can incrementally refactor such a program, stripping off complex code for such things as associations, state machines and certain patterns, and replacing it by simple Umple code. This can be done bit-by-bit, while testing thoroughly at each step. When you are done, you will have a program that should not only be more compact, but also more reliable and more maintainable. In addition, you have the benefit of being able to view UML diagrams of your program.

The term "Umple" derives from "UML Programming Language", "Simple" and "Ample".

The quickest way to get started with Umple is to go to UmpleOnline, and select an example listed under 'EXAMPLES'. Each of the user manual pages also allows you to instantly load the examples into UmpleOnline.

To learn more about Umple, read the links on the left of this page, or go to the Umple Home page. In particular, you should browse the tutorials and videos about Umple

---

See here for the statement regarding **privacy and other risks when using Umple**.

If you are an ordinary user and notice an error in this manual, please report it using our issue tracking system here *Flag it as a defect in the user documentation. If you still have trouble after reading this manual, please contact our* help mailing list and post a request for help. *If you are a contributor, you can click on the link 'edit in github' found at the bottom left of each manual page; also* instructions on how to edit this manual are here.

YouTube Video with Additional Explanation



Timothy Lethbridge





Watch on

Hello World Examples

As is customary when introducing a new language, here are some 'hello world' examples for Umple.

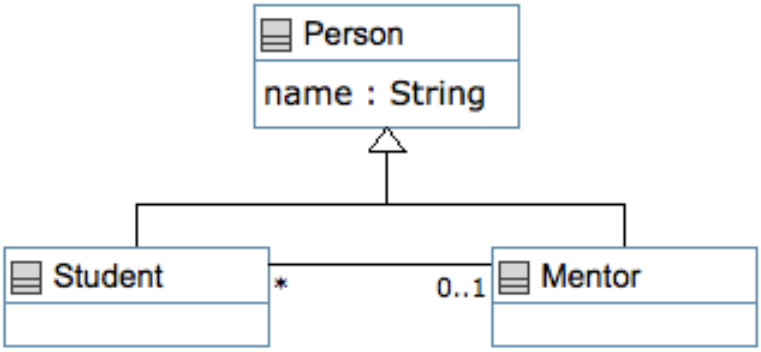
The first two examples demonstrate executable Umple programs with embedded Java main programs. Load these examples into UmpleOnline by clicking on the links. The click 'Execute It' to see the output. Or generate Java code by clicking on the 'Generate It' button; then click on the 'download zip file' link and run 'javac' on the result, followed by 'java' on the resulting class file.

About the first example below: This contains just Java code and is the same as a plain Java program; it illustrates a key feature of Umple: Umple *adds* features to existing languages: Code in the original language can and does remain the same. Umple just replaces and simplifies some (or a lot) of it.

About the second example below: This shows some very simple features of Umple: An attribute, an association, a generalization, some Java methods and the mixin capability:

- Attributes look very much like variables in Java or other similar programming languages; however they are more than that: Umple generates code for setting and getting each attribute. You can also add lots of other features to attributes, such as making them immutable (so they cannot change), adding constraints, or making them part of a 'key'.
- An association such as the one shown here maintains links between objects at run-time. Here the Mentor can find their student(s), and the students can find their optional Mentor.
- A generalization, as in other object-oriented language, sets up inheritance relationships. Umple uses the 'isA' keyword. Here both Student and Mentor inherit the name attribute.
- The methods in the example are just normal Java methods. If you generate java from this, compile using javac, and then run the Person class using java, its main program will run.
- You will notice that Person is apparently defined twice. This is a feature of Umple called using 'mixin' technology. All the elements are combined to create the resulting class.

Umple and UML: Here is the class diagram of the second example in UML. If you click on the 'open in UmpleOnline' link, you will see the UML diagram generated. You can then edit the UML diagram to change the code, or change the code to edit the UML diagram.



About the third and fourth examples below: These are the same as the first two examples, except that they contain embedded main programs in Python and C++. After you load these into UmpleOnline, select either Python or C++ from the 'GENERATE' menu, then select 'Generate It'.

Example Showing a Simple Class with a Java Main Method

view plain

```
01. /*
02.  * Simple Hello World example for Umple.
03.  * Compile this with Umple and it will generate Java
04.  * that is essentially the same.
05.  *
06.  * You could just as readily compile this code directly
07.  * with javac. However, this serves as the starting point:
08.  * Other examples in this manual show other things you
09.  * can do with Umple
10.  */
11. class HelloWorld {
12.   public static void main(String [ ] args) Java {
13.     System.out.println("Hello World");
14.   }
15.
16.   public static void main(String [ ] args) Python {
17.     print("Hello World")
18.   }
19. }
20.
21.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example Showing Three Classes with an Association and Attributes (Diagram is Above) and a Java Main Method

view plain

```
01. /*
02.  * Introductory example of Umple showing classes,
03.  * attribute, association, generalization, methods
04.  * and the mixin capability. Generate java and run this.
05.  *
06.  * The output will be:
07.  * The mentor of Tom The Student is Nick The Mentor
08.  * The students of Nick The Mentor are [Tom The Student]
09.  */
10. class Person {
11.   name; // Attribute, string by default
12.   String toString () Java {
13.     return(getName());
14.   }
15. 
16.   String __str__ () Python {
17.     return self.getName()
18.   }
19. }
20. 
21. class Student {
22.   isA Person;
23. }
24. 
25. class Mentor {
26.   isA Person;
27. }
28. 
29. association {
30.   0..1 Mentor -- * Student;
31. }
32. 
33. class Person {
34.   // Notice that we are defining more contents for Person
35.   // This uses Umple's mixin capability
36. 
37.   public static void main(String [ ] args) Java {
38.     Mentor m = new Mentor("Nick The Mentor");
```

```
39.     Student s = new Student("Tom The Student");
40.     s.setMentor(m);
41.     System.out.println("The mentor of "
42.       + s  + " is " +  s.getMentor());
43.     System.out.println("The students of "
44.       +  m  + " are " +  m.getStudents());
45.   }
46.
47.   public static void main(String [ ] args) Python {
48.     import Mentor
49.     import Student
50.
51.     m = Mentor.Mentor("Nick The Mentor")
52.     s = Student.Student("Tom The Student")
53.     s.setMentor(m)
54.     print("The mentor of " + str(s)  + " is " + str(s.getMentor()))
55.     print("The students of " + str(m) + " are " + str(m.getStudents()))
56.   }
57.
58. }
59.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example Showing a Simple Class with a Main Method for Python Generation or C++

view plain

```
01. /*
02.  * Simple Hello World example for Umple.
03.  * Compile this with Umple specifying either
04.  * -g Python or -g Cpp
05.  *
06.  * Or specify in UmpleOnline either Python or C++
07.  * in the generate menu
08.  */
09. class HelloWorld {
10.   public static void main(String [ ] args) Python{
11.     print("Hello World")
12.   }
13.
14.   public static void main(String [ ] args) Cpp{
```

```
    cout<<"Hello World!";
  }
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example Showing Three Classes with an Association and Attributes (Diagram is Above) for Python and C++ Generation

view plain

```
/*
* Introductory example of Umple showing classes,
* attribute, association, generalization, methods
* and the mixin capability.
* Generate either Python or Cpp and run this.
*
* The output will be:
* The mentor of Tom The Student is Nick The Mentor
* The students of Nick The Mentor are [Tom The Student]
*
* If you try to execute this with Java, it will
* indicate there is no Java main program. See
* the separate Java example in the user manual.
*/
class Person {
  name; // Attribute, string by default
  String __str__() Python {
    return self.getName()
  }
}

class Student {
  isA Person;
}

class Mentor {
  isA Person;
}
```

```
association {
  0..1 Mentor -- * Student;
}

class Person {
  // Notice that we are defining more contents for Person
  // This uses Umple's mixin capability

  public static void main(String [ ] args) Python{
    import Mentor
    import Student
    m = Mentor.Mentor("Nick The Mentor")
    s = Student.Student("Tom The Student")
    s.setMentor(m)
    print("The mentor of " + str(s)  + " is " +  str(s.getMentor()))
    print("The students of " +  str(m)  + " are " + str(list(map(str, m.get

}
  public static void main(String [ ] args)  Cpp {
    // Creating a mentor and a student.
    Mentor m("Nick The Mentor");
    Student s("Tom The Student");
    // Associating the student with the mentor
    s.setMentor(&m);
    m.addStudent(&s);
    // Retrieving and printing the mentor of the student.
    cout << "The mentor of " << s.getName() << " is " << s.getMentor
>getName() << endl;
    // Retrieving the list of students of the mentor.
    vector<Student*>* students = m.getStudents();
    cout << "The students of " << m.getName() << " are: " << endl;
    // Iterating over the students and printing their names.
    for (Student* student : *students) {
      cout << student->getName() << endl;
    }
    return 0;
  }
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Umple Tools

This page describes how to use Umple, either online, as a downloadable command-line compiler, or in various integrated development environments.

Umple files conventionally use the extension .ump. You use one or more of the following tools to edit, view and compile sets of .ump files.

- **UmpleOnline Classic** UmpleOnline allows you to instantly experiment with Umple on the web. You can explore examples or create your own and save them in the "cloud". You can generate diagrams, code in several languages and produce many outputs. The UmpleOnline section of this User manual describes how to use UmpleOnline. Note that UmpleOnline can be run locally on your machine by using a Docker image. See the DockerHub page for information.

- **UmpleOnline V2** UmpleOnline V2 is a new version written with AI that has almost all the same features as classic UmpleOnline, but presents a different user experience. Its repository is here. V2 is still being fine tuned, but both versions are fully usable. Part of our research is to determine what aspects of each version people like best. Both versions will be maintained for at least another year or two.

- **Visual Studio Code plugin**. This plugin, available in the VSCode marketplace, allows full-featured editing and compilation of Umple, including displaying of diagrams.. It includes Language Server Protocol support. See also the VSCode page in this user manual.

- **BBEdit tool support**. See the BBEdit page in this user manual.. It includes Language Server Protocol support.

- **Zed text editor plugin**. See the Zed page in this user manual.. It includes Language Server Protocol support.

- **Neovim text editor plugin**. See the Neovim page in this user manual.. It includes Language Server Protocol support.

- **Sublime text editor support**. See the Sublime page in this user manual.. It includes Language Server Protocol support.

- **Command-line based compiler** You can use Umple from the command line. The following are some ways to install it:
  - Download the latest version from continuous integration This is the same version as in UmpleOnline and is typically updated a few times each month. After downloading, run using Java 11 or later using

    ```
    java –jar umple.jar <options> <filename.ump> ...
    ```

  - Download the latest official release version. This is typically updated 1-4 times per year and should be chosen if you prefer stability over a period of time.

  - *Homebrew installation:* Run

    ```
    brew install umple
    ```

    on a Mac (also available for Linux). This will install the latest stable release, and will update automatically after any new release. Once installed, run as

    ```
    umple <options> <filename.ump> ...
    ```

    Note that homebrew will try to install openjdk as a dependency in homebrew first (if not already there). If you have Java installed outside of homebrew, you can avoid an unneeded installation by specifying --ignore-depenencies.

  - Windows versions on chocolatey and Linux versions under apt may be available.

The command line compiler has numerous options, including:

-g lang or --generate lang : Specify the output language (Java is the default; to ensure no output is generated, specify the word 'nothing' literally): Java,RTCpp, Php, Ruby,SQL,Json,Ecore,TextUml,Scxml Yuml, etc.
--help : Display the help message (describing the complete set of options)

--path : If a output language is specified using option -g, output source code will be placed to path
-u *code* : Specify a valid tiny umple snippet directly on the command line (can be used to add extra information to the files supplied as arguments).
-v, --version : Print out the current Umple version number
-c - : Compile the generated code (i.e. go directly from Umple to an executable system in Java, or to verify the syntax of generated PhP)

The command line compiler can handle thousands of files and millions of lines of Umple code if needed (Umple is written in itself, demonstrating this works well). The command line compiler can be incorporated into product toolchains using technology like Gradle or ant.

- **Eclipse plugin** To use Umple with Eclipse, you need to load an Umple Eclipse plugin. This can be obtained by pointing Eclipse at our update site. It has not been maintained recently and support may be dropped.

- **Docker image for UmpleOnline**: This allows you to run UmpleOnline on your own computer.
Requirement: Docker installed on your computer.

  Umple provides a Docker image of Umpleonline that allows you to explore all the features of Umple through a web browser, with the server running locally on your computer. To use this, do the following after installing Docker. Further information can be found on Umple's Docker Hub page

  - Make sure Docker is actually running on your computer
  - Pull the latest image (do this every time, so you get any changes):

    ```
    docker pull umple/umpleonline
    ```

  - Create a directory on your local machine to store your data (or reuse one that you had created previously for this purpose). In the example below we will use '~/umpledata'. As the image runs, a subdirectory of this will be created for each web session, containing the .ump files being edited, and the generated outputs. You can use the Umpleonline bookmarking capability to create a subdirectory within your local-machine directory that has a fixed name; you you can then access this repeatedly to keep working on the same

models.

- In a terminal (console), invoke the image (you can change the first 8000 to be another port both here and in the next instruction if you prefer):

```
docker run --rm -ti -p 8000:8000 -v ~/umpledata:/var/www/
```

- In a web browser, go to the following link: http://localhost:8000/umple.php

- To quit the image: a) If started as above in Linux or Mac, hit control-c; or b) run 'docker ps' to find the name of the running image and kill it using 'docker kill image'.

You can also accomplish the above by downloading and installing our script called udock and passing it the optional argument -d {dir} for your local storage directory (otherwise data will be stored in a temporary directory)

More information about Umple in Docker can be found at https://umple.org/docker

New Docker images are produced for each release of Umple, with occasional interim images. A list of Docker images can be found at the Umpleonline DockerHub site. The image tagged 'latest' will be downloaded by default.

Structure of Umple Code

Organizing the contents of an Umple (.ump) file

An Umple file will contain a number of top level items we call directives. These can be:

- **Entities** such as classes or interfaces. These first two kinds of entities look very similar to the definitions of classes or interfaces in Java. Indeed they can be identical to what you would find in a Java (or PHP or Ruby) file. However, the key difference is that they can also contain Umple code for UML constructs such as associations, attributes and state machines. Another difference is that in Umple, you can have multiple definitions of the same entity; the contents of the second and subsequent definition are *added* to the overall definition of the class.
  The specific entities you can create include the following
  - Classes
  - Interfaces
  - Traits: Reusable building blocks of classes
  - External definitions of classes
  - Standalone associations
  - Association classes
  - Standalone state machines
  - Global enumerations
- Use statements that direct Umple to include other Umple files in the system. It does not matter if you 'use' the same file repeatedly.
- Namespaces. These allow for improved modularization of a system.

Methods in classes

Much of the code in an Umple file is processed by the Umple compiler, and used to generate code in a 'base' or 'native' language (e.g, Java, PHP or Ruby) for the final system. However methods are treated differently: They are passed through essentially unchanged to the resulting system.

If you include methods in Umple code, you generally have to ensure that any given Umple file has methods of just one chosen target language (Java, PHP, Ruby. etc.). A special syntax is, however, available if you want to generate code in more than one target language.

Anything that Umple can't parse itself may be interpreted to be a method; this

can result in unintended results: What you intend to be some Umple code such as an attribute or association may end up being treated as 'extra code', i.e. a method, and passed through unchanged (normally with a warning).

The resulting system will contain many more methods than those that you explicitly include. This is because one of the key points about Umple is that it *generates* a high percentage of the methods you would normally need to write by hand if you were programming directly in the target language. In other words, when you compile Umple constructs such as associations, attributes and state machines, you are generating many methods that you can call; the set of methods you can call is the generated API. You can find out the API by using Javadoc, or a similar tool, on the generated code, or you can look at the quick reference manual page. One of the options in UmpleOnline is to generate Javadoc output.

Organizing a system containing many files

If your system is large, you should divide it into many files. One way to do this is to follow the Java convention of having one .ump file per class. Another common approach is to have one or more files for the model code (just the pure UML elements such as classes with their attributes, associations and state machines) and separate files for the methods; you can in fact have some files for Java methods, and other files for PHP or Ruby methods. The same model can then be used to develop systems that are deployed in multiple base languages.

The fact that Umple allows for multiple definitions to be added to create a complete definition, also means that you can create *mixins*. A mixin is a file that has some definitions that can be added to add extra features to a system. You can therefore organize your system, in whole or in part, by *feature*. The various pieces of code needed to implement a feature (including entire new classes, or bits such as associations and methods to add to existing classes), can therefore be grouped together. There are limits to this, however: At the current time, this mechanism does not allow you to override existing elements, which you might need to do to add a feature. Taken together, these mechanisms allow for a form of product-line development.

Examples of Mixins

The below video shows some ways to incorporate mixins to Umple classes. Follow the following links to watch further examples for:

- Mixins and state machines.
- Mixins and aspect oriented programming.

- Mixins and associations.

YouTube Video with Additional Explanation



Abdualziz AlGablan





Watch on

Syntax

```
// The core of umple is a "program". This is the grammar's 'start symbol'
// Comments and lone semicolons are ignored
program- : [[useProgram]] | [[umpleProgram]]

// Directives are the top-level items in an umple file. See manual page
TypesofDirectives
// A directive is either used to configure the system or else is
// an actual entity of the system to be modelled or generated
directive- : [[checkForUnintendedBracket]]
   | [[glossary]]
   | [[generate]]
   | [[suboption]]
   | [[distribute]]
```

| [[generate_path]]
| [[filter]]
| [[useStatement]]
| [[namespace]]
| [[requirement]]
| [[reqImplementation]]
| [[tracerDirective]]
| [[entity]]
| [[debug]]
| [[strictness]]
| [[toplevelExtracode]]
| [[toplevelException]]

// Use statements allow incorporation of other Umple files. See UseStatements
useStatement : use [use] (, [extraUse])*

// Namespaces divide the code into logical groups. See NamespaceDirectives
namespace- : namespace [namespace] [[namespaceoption]]? ;

// The main top level elements to be found in an Umple file
entity- : [[mixsetIsFeature]]
| [[requireStatement]]
| [[mixsetDefinition]]
| [[classDefinition]]
| [[traitDefinition]]
| [[fixml]]
| [[interfaceDefinition]]
| [[externalDefinition]]
| [[associationDefinition]]
| [[associationClassDefinition]]
| [[stateMachineDefinition]]
| [[templateDefinition]]
| [[enumerationDefinition]]
| [[toplevelCodeInjection]]

// Comments follow the same conventions as C-family languages. UmpleComments

comment- : [[inlineComment]] | [[multilineComment]] | [[annotationComment]]

Convincing Potential Adopters

Convincing yourself, your development team or your management to use Model-Oriented Programming with Umple

Here are some arguments to make to help convince you, your management and your colleagues to adopt model-oriented programming and Umple in particular:

1. **Model-code duality:** Umple means you no longer have to maintain models separately from the code. Your code and model become the same thing. Your models therefore don't get out of date and your volume of separately-maintained documentation becomes less. Model elements in the Umple code, along with their comments, become the design documentation.

2. **Rapid application development:** Models containing the essential structure of a system can be developed extremely fast, and modified extremely rapidly. This is because most people can edit Umple text faster than they can use a drawing tool. But the ability to use a drawing tool is not sacrificed, as you can still draw an Umple model as a diagram with UmpleOnline!

3. **Less code to write:** Code for UML constructs like associations, state machines and certain patterns is generated. Less code means fewer bugs. Developers can concentrate on the more interesting or critical parts of their code.

4. **Fewer bugs:** It is extremely difficult to consistently write bug-free code for complex constructs like state machines, or associations that maintain referential integrity and respect multiplicity constraints. Umple does this for you.

5. **The full power of UML features:** Unlike Umple, most other code generators do not generate code that enforces multiplicity in class diagrams, or allows unlimited levels of nesting of state machines. Umple has incorporated the research of several PhD and masters theses to deliver state-of-the-art code generation.

6. **Text-diagram duality:** Umple allows visualization of key aspects of your code with UmpleOnline. Furthermore the diagram can be edited to change the code, or the code can be edited to change the diagram. Both of these occur in real time.

7. **Product-line and feature-oriented development:** Umple brings mixin technology to Java and PhP, allowing you to easily build different versions

of your software for different customers or hardware platforms. Umple code can also be organized on a feature-by-feature basis.

8. **Aspect orientation to adapt generated code:** You can inject your own code before and after any generated methods to enforce constraints or change semantics.

9. **Generation for multiple languages:** Umple allows development targeted to several different programming languages at the same time: All the developer has to do is to maintain language-independent files for the models, and link them to language-specific code for algorithms.

10. **Incremental adoption:** You can start with just one Umple statement in a million lines of code and gradually increase Umple usage. Your software will remain fully compilable and functional as you gradually increase your uptake of Umple.

11. **Use just like a pre-processor:** Programmers have been confidently using pre-processors for half a century.

12. **Fully open source with a liberal license:** If you don't like something about Umple you can fix it, and contribute to the community. Because it uses an MIT-style license, there are no restrictions on your use of Umple or its generated code.

13. **Commitment to the future:** The team at the University of Ottawa developing Umple plan to continue active development for the years to come. This will not become one of those research programs that comes to an end and is then abandoned. We recognize that ideas can take decades to percolate to the community. We plan to be right there supporting Umple as this happens.

Countering the counterarguments

Here are some common arguments against adopting a new software development technology, and how they don't apply in model-oriented programming:

1. **Complexity:** Adding a new technology might add more complexity to our project so might require more learning by our employees and might make our systems harder to understand:
   Not true for Umple, because:
   - Umple is just a set of small extensions to well-known languages like Java, PhP, Ruby and (coming soon) C++.
   - Umple acts just like a preprocessor: traditional code is still there either embedded in the Umple, or with Umple embedded in the traditional code.
   - Umple can be used incrementally. You can use just a few pieces of it here and there to gain substantial advantages.
   - There is an easy-to-access user manual, with examples for all constructs and messages.
   - Umple will save a lot of coding and maintenance of boilerplate code, code for patterns, and code for constructs such as state machines.
   - Studies have shown that code using Umple is actually easier to understand than traditional code, and people learn to be better modelers when using Umple. See here for a list of peer-reviewed publications.
   - Umple is deliberately simpler than UML and other modeling technologies.

2. **Confidence:** Our company would have no way to be confident about whether it works or not and we do not have the time or resources to do the needed due diligence:
   Not true for Umple, because:
   - You can adopt Umple in extremely small increments and increase your use of it as you gain confidence.
   - If you encounter a problem with a particular use of Umple you can easily avoid that case since Umple can be blended in any combination with traditional code.
   - When using Umple a large amount of your code will be generated using templates that have been extensively tested; this saves both coding and testing time.
   - Umple is developed using test-driven development at several levels.

There are a large number of tests that are run during every build covering all the important use cases. If you encounter a problem we can fix the problem (or you can) and then we can add your case to the test suite to ensure it does not recur.
    - Large systems have been built using Umple, including the Umple compiler itself. Since Umple compiles itself any bugs are rapidly detected.
    - The developers of Umple use Umple in their daily development. This is not typically true of other modelling tools.

3. **Support:** If the technology stopped being supported, we would not be able to maintain our code base.
   Not true for Umple, because:
    - You will always have access to the generated code base. Even if Umple vanished, you would have access to the code that had been generated.
    - The generated code is designed to be easily understandable, even though it is not intended to be edited and does not need to be read in normal development situations.
    - Umple is open source, so you would always have access to it.

4. **Process:** Our testing and building processes are likely to only work for well-known technologies:
   Not true for Umple, because:
    - The Umple compiler works just like compilers of other languages and is tested and built in the same manner.

5. **Compatibility:** We use a lot of re-used, proprietary or legacy code that might be incompatible with the new technology:
   Not true for Umple, because:
    - Umple can work with any existing libraries in Java, PHP, Ruby etc. You can subclass existing classes even if you can"t see the source; you can write arbitrary code to call APIs of existing libraries, or you can convert existing code to Umple if you wish.
    - Umple is particularly good at helping incrementally re-engineer (i.e. umplifiy) legacy code: You can convert your legacy code little-by-little. But you also have the option to leave it unchanged and just use Umple for extensions or new code.

List of Umple Keywords

Here is the list of all keywords in the Umple modeling language. These cannot be used as identifiers in cases where that would cause confusion.

The following tags are used to group certain keywords

- [top] Keywords that can appear at the top level of an Umple model *(see also the grammar for top level items)*.
- [class] Keywords that can appear at the top level inside a class (and also generally inside a trait) *(see also the grammar for items that can appear in classes)*.
- [state machine] Keywords used to define state machines *(see also the grammar for state machines)*.
- [attribute] Keywords used to define attributes.
- [method] Keywords used to define methods.
- [constraints] Keywords specially associated with runtime constraints.
- [model constraints] Keywords specially associated with model constraints (checked at compile time).
- [tracing] Keywords specially associated with the runtime tracing.
- [testing] Keywords specially associated with test generation.

- **abstract:** [class] [method] Specifies that a class or method is abstract. An abstract class can not have its own instances and instances must be created from subclasses. An abstract method has no body in this class and causes the class to be abstract. *See Abstract Methods*

- **active:** [class] Specifies that instances of the class will each automatically run their own thread. *See Active Objects*

- **add:** [tracing] Indicates that the tracing should occur when an association link is being added. *See Tracing Associations*

- **after:** [top] [class] [method] Requests that code should be injected after the existing code in one or more methods (either generated methods or methods written by the user) *See Before and After Statements*

- **after:** [state machine] Triggers a transition after a specified amount of time. See Timed Transitions

- **after every:** [state machine] Continuously triggers a transition at specified time intervals in a given state, while the object remains in that state. *See Timed Transitions*

- **all:** [method] In the context of a before, after or around statement,

indicates that code should be injected in every method that matches the specified pattern (not just custom or generated code).

- **and:** [constraints] Logical AND operator used in constraints, which include state machine guards. *See Simple Constraints.*
- **around:** [top] [class] Specifies code will be inserted in place of matching methods. The original code is placed wherever the around_proceed keyword is found. *See Around Statement*
- **around_proceed:** In the context of an around statement, denotes where the original block of code should be placed. *See Around Statement*
- **as:** [state machine] Declares a state machine in a class that is a copy of a standalone state machine.*See Standalone State Machines.*
- **assertAttribute:** [testing] Asserts attribute's presence in generated code. *See Basic Testcase*
- **assertEqual:** [testing] Asserts that two objects are equal. *See Basic Testcase*
- **assertFalse:** [testing] Asserts that condition is false. *See Basic Testcase*
- **assertMethod:** [testing] Asserts methods's presence in generated code. *See Basic Testcase*
- **assertNull:** [testing] Asserts that object is null. *See Basic Testcase*
- **assertTrue:** [testing] Asserts that condition is true. *See Basic Testcase*
- **association:** [top] Defines an association between two classes, describing the links that can be defined at run time. This keyword is used when an association is created separately from the two classes. *See Independently Defined Associations*
- **associationClass:** [top] Defines a class in place of a many-to-many association. *See association class definition*
- **attribute/attr:** [model constraints] Used in model constraints to test whether an attribute matching a pattern exists. *See model constraints*
- **autounique:** [attribute] Automatically assigns a unique value to an attribute each time a class is instantiated. *See unique attributes*
- **before:** [top] [class] [method] Requests that code should be injected before existing code in one or more methods. *See Before and After Statements*

- **cardinality:** [constraints] [tracing] Operator that asserts the existence of a given number of instances of a class in an association. *See Tracing Constraints*
- **class:** [top] Defines a new class.This should be followed by a class name. *See Class Definition*
- **const:** [class] Defines a new constant. See Attribute Definition
- **custom:** [method] Indicates that code should be injected in every custom (user-written) method that matches the specified pattern, but not methods that are generated. *See Code Injection in Custom Methods*
- **defaulted:** [attribute] Provides a default value for an attribute. *See Attribute Definition*
- **depend:** [class] Denotes a dependency on another class, interface or namespace. *See Depend Clause*
- **distributable:** [class] [top] Allows for the execution of multiple objects accross different running Java Virtual Machines. *See Basic Distributed Systems*
- **do:** [state machine] Runs a process while in a state. *See State Machine Actions and Do Activities*
- **emit:** [class] Specifies the method to be generated for a particular template *See Basic Templates*
- **entry:** [state machine] Indicates an action to perform when entering a state. *See State Machine Actions and Do Activities*
- **enum:** [class] [top] Defines a user-defined data type with a set of constants. *See Enumeration Definition*
- **exit:** [state machine] Indicates an action to perform when exiting a state. *See State Machine Actions and Do Activities*
- **false:** [attribute] Boolean literal which a boolean attribute can be set to. *See Umple Built in Data Types*
- **filter:** [top] Defines a new filter to allow generation of code or diagrams from only a subset of classes and/or associations. *See Filters*
- **final:** [state machine] Precedes a state name and marks it as the final state of a state machine. *See Final States*
- **Final:** [state machine] Denotes the destination of a transition that

terminates the state machine once completed. *See Final States*

- **for:** [tracing] Preceded by an attribute, denotes the number of times the attribute's value should be changed for tracing to occur. *See Tracing Constraints*
- **forced:** Preceded by a *distributable* statement, this sets all classes in the program to be distributable, that is, allowed to be instantiated on multiple machines. *See Basic Distributed Systems*
- **generate:** [top] Specifies the languages to be generated when compiling (Java is the default); can also be specified on the command line. *See the list of keywords that can be specified in the grammar*. A list is also available from the command line by specifying --help. There is also a section in the user manual that describes the capabilities of key generators.
- **generated:** [method] Indicates that code should be injected in every generated method that matches the specified pattern, but not custom methods. *See Before and After Statements*
- **generic:** [testing] Defines a new generic test, which is a template that generates test cases for elements matching a pattern specified by the user. *See Generic Testing*
- **giving:** [tracing] Operator that specifies part of the condition under which tracing occurs. *See Tracing Constraints*
- **has:** [model constraints] Operator that asserts the existence of an association between two classes. *See model constraints*
- **hops:** In the context of a *filter* statement, indicates additional classes that can be included, which are linked to the already included classes by associations or generalizations. *See Filters*
- **ignore:** Subsequent to a *strictness* statement, this denotes a warning that should not appear, in order to reduce clutter during compilation. *See Strictness Directive*
- **immutable:** [attribute] [class] Defines an attribute that only be set when initialized. At the class level, specifies that the whole class (all attributes and associations) will be immutable. *See Immutable pattern*
- **implementsReq:** [top] [class] Binds one or more previously-defined requirements (optionally a specific quality class) to the surrounding entity — a top-level mixset, a class, a trait, an attribute, an association, a state machine, a transition, or a method. *See Requirements*

- **include:** Within a filter directive, it indicates the classes that are to be included. *See Filters*
- **includeFilter:** Within a filter directive, it indicates named filters to be included. *See Filters*
- **inner:** [class] Defines a new inner non-static class.
- **interface:** [top] Defines a new interface; specifies a set of methods that a class must implement. *See interface Definition*
- **internal:** [attribute] Denotes an internal attribute, for which no get or set statement will be generated, and which will not appear in the constructor. *See Attribute definition*
- **isA:** [class] Denotes a superclass, or implemented interface, or included trait. *See isA clause*
- **key:** [class] Marks an attribute or set of attributes as keys for equality and hashing. *See Keys for Equality and Hashing*
- **lazy:** [attribute] Causes the attribute to be initialized to a default value (0, false,empty string) according to its types, and ensures the attribute is not made an argument to the constructor. *See attribute definition*
- **mixset:** [top] Defines a new mixset, a named block of Umple that can be optionally included in the system, in order to create a variant or product line, or separately-defined feature. *See basic mixsets*
- **model:** [class] [model constraints] Defines specific conditions that must be satisfied by a model. *See model constraints*
- **namespace:** [top] Declares of group of classes that will be put in a package so as to avoid conflicts with classes having the same name in a different package. *See namespace directives*
- **not:** [constraints] Logical NOT operator used in constraints. *See Simple Constraints.*
- **of:** [model constraints] Denotes an object whose attribute is being evaluated. *See model constraints*
- **off:** Preceded by a *distributable* statement, this prevents all classes to be distributable. *See Basic Distributed Systems*
- **or:** [constraints] Logical OR operator used in constraints. *See Simple Constraints.*

- **pooled:** [state machine] Similar to a queued state machine that allows for the processing of events on separate threads but not always in the order they arrive, since if in a state that does not accept an incoming event, another acceptable transition that arrived later may be taken. *See Pooled State Machines*
- **pre:** [method] [constraints] Defines a pre-condition that must be satisfied to run a method. *See preconditions*
- **prefix:** [testing] Defines an element with a name that begins with the pattern specified. *See Generic Testing*
- **private:** [method] Sets the accessibility level of a class or its members to private. *See Method Definition*
- **protected:** [method] Sets the accessibility level of a class or its members to protected. *See Method Definition*
- **public:** [method] Sets the accessibility level of a class or its members to public. Can be omitted in Umple if the default is to be accepted (public methods, private attributes) *See Method Definition*
- **quality:** Requirement language tag (used inside a *req* definition) that introduces a set of named quality classes such as *High*, *Medium*, *Low* for a non-functional property. Quality classes can be bound to alternative designs via *implementsReq* and compared side-by-side in the generated requirements document. *See Requirement Quality Classes*
- **queued:** [state machine] Placed before the state machine declaration, this allows for the processing of events received by a state machine in a separate thread (to prevent deadlocks) and in the order they appear. See also pooled./ *See Queued State Machines*
- **--redefine:** Namespace option used to redefine a previously defined namespace. *See Namespace Directives*
- **regex:** [testing] Defines an element that matches the pattern specified. *See Generic Testing*
- **remove:** [tracing] Indicates that the tracing should occur when an association is deleted. *See Tracing Associations*
- **req:** [top] Defines a requirement with an identifier and an optional language tag (*plain* text, *userStory*, *useCase*, or *quality*). Requirements can then be attached to model elements via *implementsReq*. *See Requirements*

- **RMI:** Preceded by a *distributable* statement, this indicated the use of Remote Method Invocation (RMI) in a distributed system. *See Basic Distributed Systems*
- **self:** [class] This indicates a symmetric reflexive Association, an association from a class to itself for which there is no clear distinction between the association's ends. *See Reflexive Association*
- **settable:** [attribute] Denotes an attribute that can be set, which is the default, as opposed to immutable. See Attribute Definition
- **singleton:** [class] Marks a class as a singleton class, which can only have one instance. *See singleton pattern*
- **sorted:** [class] Specifies that elements of an association will be sorted according to an attribute. *See Sorted Associations*
- **statemachine:** [top] [state machine] Defines a state machine template at the top-level without including it in any class *See Standalone State Machines*
- **static:** [class] Defines an attribute or method to be static. A static attribute has only one value for the class. A static method can only access static attributes and other static methods.
- **strictness:** [top] Allows the control of messages that may be issued by the umple compiler, such as suppressing warnings. *See strictness Directive*
- **sub:** Within a *hops* statement of a filter, indicates the number of levels of subclasses to show. *See Filters*
- **subclass:** [model constraints] Used to assert the current class has a certain subclass defined elsewhere. *See model constraints*
- **suffix:** [testing] Defines an element with a name that ends with the pattern specified. *See Generic Testing*
- **super:** Within a *hops* statement of a filter, it indicates a numbered super class. *See Filters*
- **superclass:** [model constraints] Used to assert the current class has a certain superclass, defined elsewhere. *See model constraints*
- **test:** [testing] Defines a new test case. *See Basic Testcase*
- **trace:** [tracing] Indicates the UML entity to trace. *See Tracing Basics*
- **tracer:** [tracing] Indicates the tracer type to be used (Default is *Console*).

See Tracers

- **trait:** [top] Defines a new trait, which is a group of class content items. *See Traits*
- **true:** [attribute] Boolean literal which a boolean attribute can be set to. *See Umple Built in Data Types*
- **unique:** [attribute] Ensures that an attribute has a particular value for each instance of a class. *See unique attributes*
- **unspecified:** [state machine] Ensures that no event in a state is ignored; transitions labelled unspecified are taken if an event arrives that matches no other transition in this state. *See Unspecified Events*
- **until:** [tracing] Boolean operator that makes up a condition under which tracing occurs. *See Tracing Constraints*
- **use:** [top] Allows your program to include a model or blocks of code from another file, or activates a mixset. *See Use Statements*
- **where:** [tracing] Boolean operator that makes up a condition under which tracing occurs. *See Tracing Constraints*
- **WS:** Preceded by a *distributable* statement, this indicates the use of web services in a distributed system. *See Basic Distributed Systems*

Philosophy

The following are the key philosophies that govern the evolution of the Umple language and model-oriented programming technology. These are derived from those that first appeared in a paper published in WCRE 2010.

Elements of the Philosophy and Vision

In the following, L refers to a programming language such as Java, Python, PHP or Ruby for which Umple can generate code, or which can be embedded in Umple.

Also in the following, *Modeling* refers to representing programs at a higher level of abstraction than in a programming language, either textually or with diagrams.

P1. Modeling is programming and vice versa: UML and other modeling concepts can be expressed textually in Umple; hence, one can model in UML using Umple. For a programmer, Umple looks like a programming language, therefore, to such a person they are just programming more abstractly.

P2. An Umple programmer should never need to edit generated code to accomplish any task. The need for round-tripping (editing generated code and then reflecting the edits back into the model) is avoided.

P3. An Umple compiler can accept and generate code that uses nothing but modeling abstractions. The resulting executable can be a module providing an API rather than a complete program. If it has a 'main' method and a few extra algorithmic methods it can be a complete program. P3 is a corollary of P1.

P4. A program without Umple features can be compiled by an Umple compiler. This is the inverse of P3. In other words, any program P in base language L compiled by an Umple/L compiler will generate P. This provides a convenient starting point for a programmer who wants to begin using Umple incrementally: They can just change from using an L compiler to an Umple/L compiler.

P5. A programmer can incrementally add Umple features to an existing program. This allows for iterative conversion of a base-language L program into Umple/L, with each step being a straightforward refactoring. We call this process umplification.

P6. Umple extends the base language in a minimally invasive and safe way. A programmer familiar with language L should see the addition of an

Umple feature as just a natural extension of L. In other words, Umple appears harmonious with L. As an example, for C-family languages this is accomplished by co-opting the curlybracket block idiom and adding a very small number of additional keywords. Another way of looking at this is that Umple can be seen as a pre-processor added to the base language, that uses a syntax that is very similar to the base language. This should eliminate or reduce any fear that adopting Umple should somehow mean learning a new language. Umple can be used in very small amounts if needed. And those fearful of what would happen if Umple were to 'go away' can rest assured that they would still have good quality generated code which could in this circumstance be modified.

P7. Umple features can be created and viewed diagrammatically or textually. One can use model (e.g. UML) diagramming tools to generate an Umple program that contains only modeling abstractions, as in P2. Similarly, since Umple features map directly to modeling constructs such as UML class diagrams and state diagrams, one can easily render any Umple program as a UML diagram. This can be done in real-time using an Umple editor such as UmpleOnline that supports Umple textual code, as well as modeling visualizations. Elements of code that are not Umple abstractions, such as the bodies of methods, are omitted from the diagram.

P8. Umple goes beyond UML, since it directly implements patterns, separation of concerns mechanisms, and other programming idioms. Umple can, for example, generate code for the singleton pattern. Such capabilities further increase Umple's level of abstraction.

P9. The base language code added to an Umple program corresponds to UML's concept of an action language. To use UML for model-driven development, i.e. complete generation of an application from a UML model, UML calls for the use of an action language for the algorithmic details. UML diagrammatic modeling tools currently allow snippets of action language to be specified for elements such as state machine actions. Users of these tools have to switch between the visual editor for manipulating the modeling elements, and the textual editor for manipulating the action language snippets. This context switching from diagram to text makes it difficult to understand an entire program when using traditional UML tools.

Versions of this document

- The master source of this document, where it can be edited, is in the Umple Wiki
- This document has also been made available in the Umple User Manual

Other relevant documents

- Best Practices for those using Umple to Model and Program
- Software Engineering Process to be used by Contributors to Umple

Languages supported

Umple code can result in generation of various languages. Code from these languages can be inserted into Umple, or Umple can be inserted into code for the languages. The resulting systems can be used as part of larger projects involving these languages.

Here is the status of the programming languages currently:

- Java: Complete functionality. Umple is written in itself using embedded and generated Java. In UmpleOnline the Java code can be executed in the cloud.
- Python: Very comprehensive support, generated directly from the model; the Python page lists the few features not generated yet. In UmpleOnline the Python code can be executed in the cloud.
- Php: reasonably comprehensive generation from Umple features.
- Ruby: Partial support. No state machine support.
- C++: Complete for basic cases, but still beta.

Umple also generates a rich assortment of other outputs. The complete current list can be seen using the command line with the --help option; the list also appears below.

Several Graphviz diagrams can be generated (UML class diagrams, UML state diagrams, Entity-relationship diagrams, etc.), as can formal methods code in NuXmv and Alloy, and various forms of analysis outputs such as state tables and metrics.

Syntax

```
// The generate clause can be used to generate multiple outputs
// The --override is used to say that subsequent generate statements will be ignored
generate : generate [=language:Java
   |Nothing
   |Php
   |RTCpp
   |SimpleCpp
   |Ruby
   |Python
   |Cpp
   |Json
   |FeatureModelJson
```

```
  |StructureDiagram
  |Yuml
  |Violet
  |Umlet
  |Simulate
  |TextUml
  |Scxml
  |GvStateDiagram
  |GvClassDiagram
  |GvFeatureDiagram
  |GvClassTraitDiagram
  |GvEntityRelationshipDiagram
  |Alloy
  |NuSMV
  |NuSMVOptimizer
  |Papyrus
  |Ecore
  |Xmi
  |Xtext
  |Sql
  |StateTables
  |EventSequence
  |InstanceDiagram
  |Umple
  |UmpleSelf
  |USE
  |Test
  |SimpleMetrics
  |PlainRequirementsDoc
  |Uigu2
  |ExternalGrammar
  |Mermaid]
 ( [=suboptionIndicator:-s
  |--suboption] " [**suboption] "
 )* ;

generate_path : generate [=language:Java
```

|Nothing
|Php
|RTCpp
|SimpleCpp
|Ruby
|Python
|Cpp
|Json
|FeatureModelJson
|StructureDiagram
|Yuml
|Violet
|Umlet
|Simulate
|TextUml
|Scxml
|GvStateDiagram
|GvClassDiagram
|GvFeatureDiagram
|GvClassTraitDiagram
|GvEntityRelationshipDiagram
|Alloy
|NuSMV
|NuSMVOptimizer
|Papyrus
|Ecore
|Xmi
|Xtext
|Sql
|StateTables
|EventSequence
|InstanceDiagram
|Umple
|UmpleSelf
|USE
|Test
|SimpleMetrics

```
  |PlainRequirementsDoc
  |Uigu2
  |ExternalGrammar
  |Mermaid] " [**output] " [=override:--override
  |--override-all]?
( [=suboptionIndicator:-s
  |--suboption] " [**suboption] "
)* ;
```

Umple Comments

Comments provide a mechanism to document your work to provide some insight into *why* you are performing a certain action, as opposed to simply showing *how*.

Comments should be used just like in any programming language.

Comments immediately before class definitions, attribute definitions, association definitions and method definitions, as well as comments embedded in methods will be output into the generated code. Comments before definitions in Java will use Javadoc style; this means that when you generate JavaDoc output the API documentation will contain the comments. You are encouraged to create your Java comments using Javadoc tags.

Comments can be either inlineComments, or multilineComments.

Example Showing Inline Comments

view plain

```
01. // The Umple system is both fun
02. // and efficient to development with
03.
04.
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example Showing Both Inline And Multiline Comments

view plain

```
01. // The Umple system is both fun and
02. // efficient to development with
03.
04. /*
05.   This apple can only be compared to
06.   other apples
07. */
08.
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Inline Comments

Use two slashes to place comments after any text on the line. The comment ends at the end of the line. This is the same syntax as in Java, C++ and several other languages.

Example

```
view plain
01. // Umple is both fun and efficient to develop with
02.
03.
04.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

inlineComment- : // [**inlineComment]

Multiline Comments

Use multiline comments starting with slash-star and ending with star-slash to document your work. You should put a comment block, for example, at the beginning of each file.

This syntax is the same as in Java and C++.

Example

```
view plain

/*
  This apple can only be compared to
  other apples
*/
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

multilineComment- : /* [**multilineComment] */

Requirements

Requirements allow for the same comment to appear in different sections of the code without typing out the comment multiple times.

They consist of two components: the requirement definition, which contains the comment statement to be added, and the requirement implementation, which specifies where the requirement statement should be added.

Multiple requirements can be called through an implementsReq statement.

A requirement can be tagged with multiple entities like classes, attributes, associations, state machines, traits, methods, interface etc.

Requirement-definition syntax

A requirement is introduced with the `req` keyword, followed by an identifier and an optional requirement-language tag. The body is enclosed in curly braces:

```
req <ReqIdentifier> [<reqLanguage>] { ... }
```

The supported requirement languages are:

- **plain** (default, tag may be omitted) — free text.
- **userStory** — free text optionally structured with `who`, `when`, `what`, and `why` sub-blocks.
- **useCase** (also written `usecase`) — free text, or the steps of an interaction between an actor and the system. See the section below.
- **quality** — introduces named quality classes (e.g. *High*, *Medium*, *Low*) for a non-functional property. See Requirement Quality Classes.

useCase requirements

A `useCase` requirement describes an interaction between an actor and the system. Its body may be free text, or may be structured with any combination of the sub-blocks below, written one per line and in any order:

- `who`, `when`, `what` and `why` — the same sub-blocks used by `userStory` requirements.
- `userStep <id> { ... }` and `systemResponse <id> { ... }` — the steps of the scenario. The identifier may be a number or alphanumeric, and the steps are kept in the order they are written.
- `isA <ReqIdentifier> [, <ReqIdentifier>]* ;` — the use case is a special case of the named use cases, as with UML generalization.

- `include <ReqIdentifier> [, <ReqIdentifier>]* ;` — the use case incorporates the steps of the named use cases, as with the UML include relationship.

A step may carry a condition in square brackets, written like a state machine guard. The step applies only when the condition holds, so several steps can share one identifier to describe alternative outcomes of the same point in the scenario:

```
req UC5 useCase {
  isA UC3;
  include UC4;
  userStep 1 { select a payment method }
  systemResponse 1 [card accepted] { confirm the order }
  systemResponse 1 [card declined] { ask for another card }
}
```

See Requirements Examples for complete useCase models in plain, structured and related forms.

implementsReq statement

An `implementsReq` statement attaches one or more requirements to the surrounding entity. It can appear at the top level (binding to a following mixset or class), inside a class, trait, state machine, attribute, association, or method:

```
implementsReq <ReqIdentifier> [(<QualityClassName>)] [, <ReqIdentifier>
```

The optional parenthesised argument selects a specific quality class of the referenced requirement, which is required when using `implementsReq` to differentiate alternative designs on a non-functional property:

```
implementsReq Speed(High), Security(Perfect);
mixset DesignA { ... }
```

See also:

- Requirements Examples for a gallery of requirement models.
- Requirement Quality Classes for non-functional requirement declarations.
- Requirement Quality Comparison Table for generating a side-by-side comparison of alternative designs.
- Sorting Requirements for suboptions that control requirement ordering and filtering.

Example

```
view plain

01. req R01 {
02.   This is a comment we would like to add
03.   multiple times in different locations.
04.
05.   The identifier used for this requirement is
06.   R01.
07. }
08.
09. req R02 {
10.   This is a second requirement statement.
11. }
12.
13. implementsReq R01;
14. class Example {
15.   implementsReq R02;
16.   var1;
17.   var2; implementsReq R01, R02;
18. }
19.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Requirement tagging with multiple entities

```
view plain

01. req R1 {
02.   First requirement with the identifier R1.
03. }
04.
05.
06. req R2 {
07.   Second requirement with the identifier R2.
08. }
09.
10.
11. req R3 {
12.   Third requirement with the identifier R3.
13. }
14.
15.
```

```
req R4 {
  Fourth requirement with the identifier R4.
}


req R5 {
  Fifth requirement with the identifier R5.
}


req R6 {
  Sixth requirement with the identifier R6.
}


// Requirements tagged with class
implementsReq R3;
class A {

  // Requirements tagged with attribute
  implementsReq R2;
  att1;
}


// A second class
class B {

  // Requirement tagged with association
  implementsReq R2;
  * -> * A;

  // Requirement tagged with state machine
  implementsReq R3;
  sm {
    s1 {
      e -> s2;
    }
    s2 {
    }
  }

  // Requirement tagged with method
  implementsReq R5;
  String m1(String s) {
```

```
61.     return(s);
62.   }
63. }
64.
65.
66. // Requirement tagged with trait
67. implementsReq R4;
68. trait C {
69.   g;
70. }
71.
72.
73. // Requirement tagged with interface
74. implementsReq R6;
75. interface Itest {
76.   String m2(String s);
77. }
78.
79.
80.
81.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Requirements Examples

Examples of requirements documents in Umple that can be used to generate code

This page provides examples of Umple requirements using plain-text, structured userStory and structured useCase styles. They show possible ways of specifying requirements. They can be used to guide you before manually creating a model, or to use UmpleOnline AI capabilities to generate actual Umple code.

Hotel booking requirements for creating class diagrams or state machines

view plain

```
01. // The following is a (partial) set of requirements for
02. // a hotel booking that can be used to experiment
03. // with generating class diagrams and state machines in Umple // using
04. // or merely as an example of how requirements can be
05. // specified in Umple.=
06. // Req H008 was used in the following thesis by Parva Pathak
07. // https://ruor.uottawa.ca/items/b3679a91-5445-45ce-b289-
    bfddba3010f6
08.
09. req H001 {
10.   The hotel chain keeps a list of hotels in the system, each
11.   with their address and set of rooms
12. }
13.
14. req H002 {
15.   There can be various types of rooms, characterized by
16.   quality of the view (Number of guests allowed, Excellent,
17.   good, poor), Number of Queen beds, Number of King beds,
18.   Number of Single beds., whether there is a safe, whether
19.   there is a pullout couch, and whether it is a suite such
20.   that the bedroom is behind a separate door)
21. }
22.
23. req H003 {
24.   Each type of room is given a price that will be advertised
25.   for it for each day. Prices are set up 9 months in
26.   advanced, but can be changed dynamically.
27. }
28.
```

```
req H004 {
  Each room has a type.
}

req H005 {
  On any given day (overnight from one night to the following
  morning), each room can be either available, allocated to a
  guest, or unavailable due to maintenance.
}

req H006 {
  When a room is booked, the price each day for the booking
  is confirmed (it will be the room-type price, or less if
  discounts are given).
}

req H007 {
  A booking has a start date, an end date, a number of adults and a nur
}

req H008 {
  The hotel runs a system that allows users to book rooms online, provi
  {
    The user can optionally login to the hotel website
    The user can choose a location
    The user can choose a hotel
    The user can choose the number of guests
    The user can choose the date range of the visit
    The user can choose a room type
    The user can pay for the hotel
    The user can cancel the booking
  }
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Drivers license system requirements for creating class diagrams or state machines

view plain

```
// The following is a (partial) set of requirements for
```

```
02. // the process of getting a driver's license in Ontario,
03. // Canada.
04. // It can be used for generating state machines in Umple
05. // using AI,
06. // or merely as an example of how requirements can be
07. // specified in Umple.=
08. // Requirements 1-8 were used in the following thesis by
09. // Parva Pathak
10. // https://ruor.uottawa.ca/items/b3679a91-5445-45ce-b289-
    bfddba3010f6
11. // It has been extended to also add knowledge for generating
12. // a class diagram
13.
14. req L01-LicenseTypes {
15.   The user can have no license, a G1 license, a G2 license or
16.   a G license
17. }
18.
19. req L02-G1Test {
20.   To get a G1, a test must be complete
21. }
22.
23. req L03-G2Test {
24.   To get a G2, the user must have a G1 and a test must be
25.   completed
26. }
27.
28. req L04-GTest {
29.  To get a G, the user must have a G2 and a test must be
30.  completed
31. }
32.
33. req L05-Expiry {
34.   Each type of license can expire.
35. }
36.
37. req L06-nonrenewG1orG2 {
38.   If a G1 or G2 expires then the license is lost
39. }
40.
41. req L07-renewG {
42.   If a G expires it can be renewed
43. }
44.
45. req L08-suspension {
```

```
  Each type of license can be suspended
}

req L09-licensedata {
  A license has a licences number and an issued date
}

req L11-holderdata-{
  The system will keep the following information about the
  holder of a license: Name, Date of Birth, Address
}

req L12-testdata {
  The system will keep track of the tests of each license
  holder or applicant (who has not yet passed a test), along
  with the score on each test.
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Blackjack game requirements for state machine generation

view plain

```
// The following is a (partial) set of requirements for
// the card game Blackjack, that can be used to experiment
// with generating state machines in Umple using AI,
// or merely as an example of how requirements can be
// specified in Umple.
// It was used in the following thesis by Parva Pathak
// https://ruor.uottawa.ca/items/b3679a91-5445-45ce-b289-bfddba3010f6

req 001dealing {The player and dealer will be dealt cards}

req 002playerchoice {The player can choose to hit or stand}

req 003dealerchoice {The dealer can choose to hit or stand}

req 004dealerwin {If dealer gets a blackjack the dealer wins}

req 005playermaywin {If the player stands and the dealer has
```

```
18. a lower score the player wins}
19.
20. req 006dealermaywin {If the player stands and the dealer has
21. a higher score the dealer wins}
22.
23. req 007samescoremore {If the dealer and player have the same
24. score the game will push}
25.
26.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Very simple credit card approval partial requirements

view plain

```
01. // The following is a (partial) set of requirements for
02. // a credit card approval system that can be used to
03. // experiment
04. // with generating state machines in Umple using AI,
05. // or merely as an example of how requirements can be
06. // specified in Umple.=
07. // It was used in the following thesis by Parva Pathak
08. // https://ruor.uottawa.ca/items/b3679a91-5445-45ce-b289-
    bfddba3010f6
09.
10. req CC1 {
11.   The different stages of the approval process must be
12.   included (Pre Approval, Cancelled, Complete, Failed)
13. }
14.
15. req CC2 {
16.   The status of the account must be included
17.   (On Hold, Not On Hold)
18. }
19.
20. req CC3 {
21.   The transaction can be successful or fail
22. }
23.
24.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Agent loop system requirements for state machine generation

view plain

```
01. // The following is a set of requirements for an agent loop
02. // system that can be used to experiment with generating
03. // state machines in Umple using AI, or merely as an example
04. // of how requirements can be specified in Umple.
05.
06. req AL01-IdleEntry {
07.   The agent loop waits in an idle mode until a run is started or continue
08.   then it enters the assistant turn.
09. }
10.
11. req AL02-AssistantTurnToolRequest {
12.   During the assistant turn, if the assistant requests tool execution,
13.   the agent loop enters the tool turn.
14. }
15.
16. req AL03-AssistantTurnCompletion {
17.   During the assistant turn, if the assistant completes its response,
18.   the agent loop proceeds to a queue check step.
19. }
20.
21. req AL04-AssistantTurnFailureOrAbort {
22.   During the assistant turn, if the assistant fails or aborts,
23.   the agent loop ends the run.
24. }
25.
26. req AL05-ToolTurnPermissionGate {
27.   During the tool turn, if a requested tool requires permission,
28.   the agent loop pauses to await a permission decision.
29. }
30.
31. req AL06-ToolTurnReturnToAssistant {
32.   During the tool turn, when tool execution completes,
33.   the agent loop returns to the assistant turn.
34. }
35.
36. req AL07-ToolTurnSteeringInterrupt {
37.   During the tool turn, a steering interrupt can immediately return the
```

```
  to the assistant turn.
}

req AL08-PermissionGranted {
  When awaiting tool permission, granting permission allows the agent l
  to resume tool execution.
}

req AL09-PermissionDenied {
  When awaiting tool permission, denying permission returns the agent
  to the assistant turn without continuing tool execution.
}

req AL10-QueueCheckContinue {
  After the assistant completes, if a queued message is available,
  the agent loop continues with another assistant turn.
}

req AL11-QueueCheckEnd {
  After the assistant completes, if no queued message is available,
  the agent loop ends the run.
}

req AL12-ResetToIdle {
  After the run ends, the agent loop can be reset back to idle.
}

req AL13-QueuedProcessing {
  The agent loop processes queued messages in order, and queue check
  occurs after an assistant completion.
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Plain-text userStory requirement example

view plain

```
req US1 userStory {
  As a customer, I want to reset my password so that I can regain acces
}
```

```
04.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Structured userStory requirement example using who, when, what, and why

view plain

```
01. req US2 userStory {
02.   who { customer }
03.   when { password is forgotten }
04.   what { reset my password }
05.   why { regain access to my account }
06. }
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Partial structured userStory requirement example

view plain

```
01. req US3 userStory {
02.   who { administrator }
03.   what { manage users }
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Plain-text useCase requirement example

view plain

```
01. req UC1 useCase {
02.   A registered customer places an order: they review the cart, confirm t
03.   delivery address and choose a payment method, and the system emai
04. }
05.
```

```
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Structured useCase requirement using who, when, what, why, userStep and systemResponse

view plain

```
01. req UC2 useCase {
02.   who { registered customer }
03.   when { the cart holds at least one item }
04.   what { place an order }
05.   why { receive the goods at home }
06.
07.   userStep 1 { review the cart }
08.   systemResponse 1 { show the order total including delivery }
09.   userStep 2 { confirm the delivery address }
10.   systemResponse 2 { reserve the stock }
11.   userStep 3 { choose a payment method }
12.   systemResponse 3 { email a receipt }
13. }
14.
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Partial structured useCase requirement example

view plain

```
01. req UC3 useCase {
02.   who { store administrator }
03.   what { cancel an order }
04.
05.   userStep 1 { choose the order to cancel }
06.   systemResponse 1 { release the reserved stock }
07. }
08.
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Use cases related by isA and include, with conditional steps

view plain

```
01. req UC4 useCase {
02.   what { place an order }
03.   userStep 1 { review the cart }
04.   systemResponse 1 { show the order total }
05. }
06.
07. req UC5 useCase {
08.   what { authenticate the customer }
09.   userStep 1 { enter email and password }
10.   systemResponse 1 { open a session }
11. }
12.
13. req UC6 useCase {
14.   who { returning customer }
15.   what { place an order using a saved address }
16.   isA UC4;
17.   include UC5;
18.
19.   userStep 1 { confirm the saved address }
20.   systemResponse 1 [address still valid] { show the order summary }
21.   systemResponse 1 [address no longer valid] { ask for a new address
22.   userStep 2 { pay }
23. }
24.
25.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Quality requirement with named quality classes

view plain

```
01. // Quality requirements introduce named quality classes
02. // (discrete levels of a non-functional property) that can
03. // later be bound to alternative designs via implementsReq.
```

```
04.
05.  req Speed quality {
06.    High {}
07.    Medium {}
08.    Low {}
09.  }
10.
11.  req Security quality {
12.    Perfect {}
13.    SlightlyRisky {}
14.    Weak {}
15.  }
16.
17.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Binding quality classes to alternative mixsets via implementsReq

view plain

```
01.  // implementsReq binds a requirement (and optionally a specific
02.  // quality class) to the following entity. Each mixset below is
03.  // an alternative design tagged with its own quality levels.
04.
05.  req Speed quality {
06.    High {}
07.    Low {}
08.  }
09.
10.  req Security quality {
11.    Perfect {}
12.    Basic {}
13.  }
14.
15.  implementsReq Speed(High), Security(Perfect);
16.  mixset DesignA {
17.    class FastSecureCar {}
18.  }
19.
20.  implementsReq Speed(Low), Security(Basic);
21.  mixset DesignB {
22.    class SlowCar {}
```

```
}

use DesignA;


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Full model producing an alternative-design comparison table

view plain

```
// Full example producing the alternative-design comparison
// table. The reqQC suboption of PlainRequirementsDoc emits an
// HTML table with one row per quality requirement and one
// column per quality-tagged mixset.

generate PlainRequirementsDoc;
suboption "reqQC";

require subfeature [1..1 of {DesignA, DesignB}];

req Speed userStory {
  who { architect }
  what { compare speed across designs }
}

req Speed quality {
  High {}
  Low {}
}

req Security userStory {
  who { architect }
  what { compare security across designs }
}

req Security quality {
  Perfect {}
  Basic {}
}

implementsReq Speed(High), Security(Perfect);
```

```
32. mixset DesignA {
33.   class FastSecureCar {}
34. }
35.
36. implementsReq Speed(Low), Security(Basic);
37. mixset DesignB {
38.   class SlowCar {}
39. }
40.
41. use DesignA;
42.
43.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Sorting Requirements

Requirements can be sorted by ID using the suboption 'reqSortID' and Statement by using the suboption 'reqSortStat'.

The default is to sort by ID when no suboption is provided.

Additionally there is also a suboption 'reqHideNotImpl' to filter out the not implemented requirements.

Example Showing Sorting by ID

view plain

```
01. // This is an example to use reqSortID to sort requirements by thier ID.
02. suboption "reqSortID";
03. req A01 {
04.   b requirement.
05.  }
06. req Z01 {
07.   a requirement.
08.  }
09. implementsReq Z01;
10. class Example {
11.   implementsReq A01;
12.   var1;
13.  }
14.
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example Showing Sorting by Statement

view plain

```
01. // This is an example to use reqSortID to sort requirements by thier Sta
02. suboption "reqSortStat";
03. req A01 {
04.   b requirement.
```

```
}
req Z01 {
  a requirement.
}
implementsReq Z01;
class Example {
  implementsReq A01;
  var1;
 }
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example Showing Filtering of not implemented requirements

view plain

```
// This is an example to show how not implemented requirements can b
suboption "reqHideNotImpl";
req B01 {
  b is a requirement.
}
req A02 {
  a is a requirement.
}
class Example1 {
  implementsReq A02;
  var1;
 }
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Requirement Quality Classes

A requirement can be tagged with the `quality` language, which marks it as a non-functional requirement characterised by a set of named *quality classes*. Each class represents a discrete level or option for the property being described (for example *High* / *Medium* / *Low* for speed, or *Perfect* / *SlightlyRisky* / *Weak* for security).

Syntax:

```
req <ReqIdentifier> quality {
  <QualityClassName> {}
  <QualityClassName> {}
  ...
}
```

Quality classes are declared with empty bodies. Non-empty bodies are currently accepted by the parser but produce warning 403 and have no effect on generated output.

An `implementsReq` statement can reference a specific quality class using parenthesised arguments, and several bindings can be combined in a single statement:

```
implementsReq Speed(High);
implementsReq Speed(High), Security(Perfect);
```

When alternative designs (typically different mixsets) are each tagged with quality classes referring to the same set of quality requirements, the designs can be compared on a common set of criteria. See Requirement Quality Comparison Table for the generated output, and Requirements for the general requirement-statement syntax.

Basic quality requirement declaration

view plain

```
01. // Quality requirements introduce named quality classes
02. // (discrete levels of a non-functional property) that can
03. // later be bound to alternative designs via implementsReq.
04.
05. req Speed quality {
06.   High {}
07.   Medium {}
```

```
  Low {}
}

req Security quality {
  Perfect {}
  SlightlyRisky {}
  Weak {}
}


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Binding quality classes to alternative mixsets

view plain

```
// implementsReq binds a requirement (and optionally a specific
// quality class) to the following entity. Each mixset below is
// an alternative design tagged with its own quality levels.

req Speed quality {
  High {}
  Low {}
}

req Security quality {
  Perfect {}
  Basic {}
}

implementsReq Speed(High), Security(Perfect);
mixset DesignA {
  class FastSecureCar {}
}

implementsReq Speed(Low), Security(Basic);
mixset DesignB {
  class SlowCar {}
}

use DesignA;

```

27.

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Requirement Quality Comparison Table

Once alternative designs have been tagged with quality classes through `implementsReq` (see Requirement Quality Classes), the Plain Requirements Doc generator can emit a comparison table. The table lists every quality requirement as a row and every quality-tagged mixset as a column, so the alternatives can be compared on a shared set of criteria.

The feature is activated with the `reqQC` suboption of the `PlainRequirementsDoc` generator. From inside an Umple file:

```
generate PlainRequirementsDoc;
suboption "reqQC";
```

Or from the command line:

```
umple -g PlainRequirementsDoc -s reqQC your_model.ump
```

In UmpleOnline you can also select Requirements Doc with Comparison Table from the Generate dropdown, which invokes the same generator with `reqQC` pre-applied without requiring the `suboption "reqQC";` directive in the source file.

The currently-active mixset (the one selected via a `use` statement or through the UmpleOnline feature-tree selector) is prefixed with an asterisk in the generated table header.

See also Sorting Requirements for other suboptions supported by the same generator, including `reqSortID`, `reqSortStat`, and `reqHideNotImpl`.

Generating a comparison table for two alternative mixsets

view plain

```
// Full example producing the alternative-design comparison
// table. The reqQC suboption of PlainRequirementsDoc emits an
// HTML table with one row per quality requirement and one
// column per quality-tagged mixset.

generate PlainRequirementsDoc;
suboption "reqQC";

require subfeature [1..1 of {DesignA, DesignB}];

```

```
req Speed userStory {
  who { architect }
  what { compare speed across designs }
}

req Speed quality {
  High {}
  Low {}
}

req Security userStory {
  who { architect }
  what { compare security across designs }
}

req Security quality {
  Perfect {}
  Basic {}
}

implementsReq Speed(High), Security(Perfect);
mixset DesignA {
  class FastSecureCar {}
}

implementsReq Speed(Low), Security(Basic);
mixset DesignB {
  class SlowCar {}
}

use DesignA;


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Types of Directives

Directives appear as the 'main' entries in an Umple file. The following are the main types:

- Use statements include other Umple files to allow for modularization and reuse.
- **Entity declarations** define the top level model or code elements including classes, interfaces, traits, associations, association classes, and enumerations. Other model or code elements, such as methods or attributes must be placed *inside* these entities. Note that the same entity can be declared more than once to add new internal contents. This is called using a mixin (one mixes together the contents of multiple declarations).
- Aspects (code injections) tell Umple to inject certain code before, after or around certain methods of certain classes. These can be used to modify the behaviour of both custom and generated code.
- Mixset directives. These are named blocks of Umple code that can be included (optionally) with a use statement. Several of them can have the same name. If there is no use statement for the mixset, that code is ignored. These can be used to build product lines and different software versions.
- Namespace directives gather entities in logical groups. Within a namespace, entities must have different names. Namespaces affect code generation; for example they correspond to packages in Java.
- Strictness Directives tell the compiler to issue extra warnings or errors in certain situations, or to turn certain errors into warnings, or to suppress certain warnings.
- **Generate directives** tell Umple one or more generated outputs to create (see the grammar below for the list of languages and other outputs that can be generated). You can omit these entirely and specify them on the command line or using UmpleOnline. As a second argument on the generate directive, you can specify a path (directory) where the output should be put. You can also specify --override-all to ensure subsequent directives are ignored.
- **Tracer directives** tell Umple which tracing tool to use in order to trace execution of the code.

Syntax

```
// Directives are the top-level items in an umple file. See manual page
TypesofDirectives
// A directive is either used to configure the system or else is
// an actual entity of the system to be modelled or generated
directive- : [[checkForUnintendedBracket]]
   | [[glossary]]
   | [[generate]]
   | [[suboption]]
   | [[distribute]]
   | [[generate_path]]
   | [[filter]]
   | [[useStatement]]
   | [[namespace]]
   | [[requirement]]
   | [[reqImplementation]]
   | [[tracerDirective]]
   | [[entity]]
   | [[debug]]
   | [[strictness]]
   | [[toplevelExtracode]]
   | [[toplevelException]]

// The generate clause can be used to generate multiple outputs
// The --override is used to say that subsequent generate statements will be ignored
generate : generate [=language:Java
   |Nothing
   |Php
   |RTCpp
   |SimpleCpp
   |Ruby
   |Python
   |Cpp
   |Json
```

```
  |FeatureModelJson
  |StructureDiagram
  |Yuml
  |Violet
  |Umlet
  |Simulate
  |TextUml
  |Scxml
  |GvStateDiagram
  |GvClassDiagram
  |GvFeatureDiagram
  |GvClassTraitDiagram
  |GvEntityRelationshipDiagram
  |Alloy
  |NuSMV
  |NuSMVOptimizer
  |Papyrus
  |Ecore
  |Xmi
  |Xtext
  |Sql
  |StateTables
  |EventSequence
  |InstanceDiagram
  |Umple
  |UmpleSelf
  |USE
  |Test
  |SimpleMetrics
  |PlainRequirementsDoc
  |Uigu2
  |ExternalGrammar
  |Mermaid]
( [=suboptionIndicator:-s
  |--suboption] " [**suboption] "
)* ;
```

```
generate_path : generate [=language:Java
   |Nothing
   |Php
   |RTCpp
   |SimpleCpp
   |Ruby
   |Python
   |Cpp
   |Json
   |FeatureModelJson
   |StructureDiagram
   |Yuml
   |Violet
   |Umlet
   |Simulate
   |TextUml
   |Scxml
   |GvStateDiagram
   |GvClassDiagram
   |GvFeatureDiagram
   |GvClassTraitDiagram
   |GvEntityRelationshipDiagram
   |Alloy
   |NuSMV
   |NuSMVOptimizer
   |Papyrus
   |Ecore
   |Xmi
   |Xtext
   |Sql
   |StateTables
   |EventSequence
   |InstanceDiagram
   |Umple
   |UmpleSelf
   |USE
   |Test
```

```
   |SimpleMetrics
   |PlainRequirementsDoc
   |Uigu2
   |ExternalGrammar
   |Mermaid] " [**output] " [=override:--override
   |--override-all]?
  ( [=suboptionIndicator:-s
   |--suboption] " [**suboption] "
  )* ;

// Use statements allow incorporation of other Umple files. See UseStatements
useStatement : use [use] ( , [extraUse] )*

// Namespaces divide the code into logical groups. See NamespaceDirectives
namespace- : namespace [namespace] [[namespaceoption]]? ;

// The main top level elements to be found in an Umple file
entity- : [[mixsetIsFeature]]
   | [[requireStatement]]
   | [[mixsetDefinition]]
   | [[classDefinition]]
   | [[traitDefinition]]
   | [[fixml]]
   | [[interfaceDefinition]]
   | [[externalDefinition]]
   | [[associationDefinition]]
   | [[associationClassDefinition]]
   | [[stateMachineDefinition]]
   | [[templateDefinition]]
   | [[enumerationDefinition]]
   | [[toplevelCodeInjection]]
```

Use Statements

Use statements allow you to decompose your system by embedding or referencing files containing other model elements (such as classes) into your current model, or to include optional code blocks defined in mixsets.

A model file or mixset is included only once; subsequent "use" commands for the same file will be ignored. Preceding a mixset name with '!' indicates to not use the mixset, and cancels out any previous use statement that requested to use the mixset.

A common technique is to create a 'master' Umple file that does nothing but have a list of use statements.

Parts of an individual class can be specified in separate files, and brought together using several use statements. For example the associations or attributes could be in one (or several) files, and the methods could be in one (or more) additional files.

Another way to decompose a system is to have a 'core' set of files that can be included in several different systems using 'use' statements.

Use statements function similarly to 'include' statements in other languages.

A file referred to by a use statement must be in the same directory as the file that has the use statement, or in a parent directory, or in a subdirectory named lib.

Umple also ships built-in library files for common modeling patterns. Load one with the `lib:` prefix. The file is read from inside the Umple compiler, so it does not need to be in your project directory:

```
use lib:AbstractionOccurrence.ump;
```

The built-in pattern libraries are:

- Abstraction Occurrence (`use lib:AbstractionOccurrence.ump;`)
- Player Role (`use lib:PlayerRole.ump;`)
- Transportation Pattern A (`use lib:TransportationPatternA.ump;`)

Each of those pages shows the pattern definition and an example of applying it in a model.

A use statement can also load a `.ump` file from the web. The argument must be an `https` URL that ends in `.ump`. A raw file on GitHub is a typical source:

```
use https://raw.githubusercontent.com/umple/umple/master/umpleonline/ur
```

Example

view plain

```
01. use Core.ump;
02.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

// Use statements allow incorporation of other Umple files. See UseStatements
useStatement : use [use] (, [extraUse])*

Namespace Directives

Namespaces allow you to group similar entities to promote cohesion, as well as reduce the possibility of name collision. Sub-namespaces are separated using a period (.).

In the first example, the classes Faculty and Student will be in namespace school.admin. In Java they will be generated into the admin package (directory) within the school package. Similarly, class Building will be in the elevator.structure namespace.

A previously defined namespace for an entity can be redefined using the "--redefine" option. If the "--redefine" option is not used, the namespace will not change and a warning will be issued (example 2).

Entities declared before any namespace or after "namespace default;" will be in the default package. Entities declared after "namespace -;" will not be in the last declared namespace. Instead, they will be in the default package. If declared after a non-default namespace, the namespace of an entity in the default namespace will be redefined (example 3).

There are some cases where an entity should be imported thus cannot be in the default package and will automatically be placed in another package (example 4). However, automatic import code generation in interfaces that extend interfaces in other packages is not supported yet.

Example 1: Sub-namespaces

view plain

```
01. namespace school.admin;
02.
03. class Faculty{}
04. class Student{}
05.
06. namespace elevator.structure;
07.
08. class Building
09. {
10.   1 -- * Classroom;
11. }
12.
13. class Classroom{}
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 2: Redefining a namespace

view plain

```
01. // Entities A, B and C are currently in
02. // namespace m.
03.
04. namespace m;
05. class A{}
06. interface B{}
07. class C{}
08.
09.
10. // To redefine the namespace of an
11. // entity, use the --redefine option.
12. // B is now in namespace n.
13.
14. namespace n --redefine;
15. interface B{}
16.
17.
18. // If --redefine is not used, a warning will
19. // be issued when trying to redefine the
20. // namespace of an entity
21.
22. namespace p;
23. class C{}
24.
25.
26. // Using namepace -; will deactivate the last
27. // declared namespace. There will not be an
28. // attempt to redefine the namespace of interface
29. // B and a warning will not be issued
30.
31. namespace -;
32. interface B{}
33.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 3: Default namespace

view plain

```
01. // Class A will be in the default namespace
02.
03. class A{}
04. class B{}
05.
06.
07. // Class B was in the default namespace
08. // But now will be in namespace m
09.
10. namespace m;
11. class B{}
12. class C{}
13. namespace -;
14.
15.
16. // Because namespace -; was used, namespace m
17. // is no longer active, and class D will be
18. // in the default namespace
19.
20. class D{}
21.
22.
23. // Class C is in namespace m, but can
24. // be placed in the default namespace
25.
26. namespace default --redefine;
27. class C{}
28.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 4: Automatically redefined namespace

view plain

```
01. // Classes A is in the
02. // default namespace. Class B is in
03. // namespace m. Both are
```

```
04. // linked to each other via an association,
05. // therefore, and because they are not all
06. // in the same namespace, an import text
07. // should be generated. However, many programming
08. // languages do not support import from
09. // the default namespace so the namespace of
10. // A will become m because B is in m.
11.
12. class A{}
13. namespace m;
14. class B{*--* A;}
15. namespace -;
16.
17.
18. // The same issue occurs when an
19. // entity in a non-default namespace
20. // extends or implements an entity
21. // in the default namespace
22.
23. interface E{}
24. interface F{}
25. namespace n;
26. interface J{isA E;}
27. class K{isA F;}
28.
29.
30. // Here an import text will be generated for A in X
31. // because A is in a different namespace
32. // This feature is not yet supported for interfaces
33. // that extend interfaces in other namespaces
34. // and the import text will not be generated for K in Y
35.
36. namespace p;
37. class X{isA A;}
38. // interface Y{isA E;}
39. // @@@skipcppcompile
40.
41.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
// Namespaces divide the code into logical groups. See NamespaceDirectives
namespace- : namespace [namespace] [[namespaceoption]]? ;

namespaceoption : [=option:--redefine]
```

Strictness Directive

The strictness directive is used to control certain messages that the Umple compiler may issue. It has five subdirectives that are specified by a second keyword following 'strictness':

The first two are 'modelOnly' or 'noExtraCode'. These are used when the programmer or modeller intends not to include base language code, and wants a warning to appear if base language code is found. Base language code is code in a language like Java or PHP that is discovered by Umple but not interpreted in any way. One example is the code in the bodies of methods; however, when parsing a class, any time Umple cannot parse what it finds, it assumes it must be base language code. It just emits the base language code for the base language compiler to deal with. However there are circumstances when the developer does not want this: The developer may be creating a pure model or may want that the only base language code would be in the body of methods. It is advantageous therefore to tell the compiler to raise a warning if it thinks it has found base language code in some other context, since otherwise, an ordinary Umple syntax error may go undetected, until the base language compiler is run on the code.

- **strictness modelOnly;** means be very strict and issue a warning if base language code is found.
- **strictness noExtraCode;** means that base language code is allowed in method bodies, but if found anywhere else to issue a warning.

The second set of subdirectives are 'expect', 'allow' and 'disallow'. These are used to control the effect of certain messages. They are followed by a message number.

- **strictness expect n** declares that message number n *should occur*; it is an error if the message does not. This is used in testing to create example cases of message n; an error would be triggered if message n does *not* appear.
- **strictness allow n** is used in the case of errors, to tell the compiler to not actually 'fail', but to report that error as if it was a warning. In the case of n being a warning, the warning is not emitted. This is also used in testing to include cases that give message n, without the compiler reporting that it has failed.
- **strictness disallow n**, where n is a warning, tells the compiler to treat n as if it was an error, and fail the compilation.

Some of the above are still under development.

The strictness directives take effect on the entire system being built. The code

of all Ump files of that system will be subject to the strictness subdirectives.

Example

```
view plain

// Tell the compiler that error 25
// will appear (and fail if it does not)
strictness expect 25;

// Tell the compiler that if error 22
// occurs, not to fail the compile.
// However, this does not mean that
// code can be generated
strictness allow 22;

// UNDER DEVELOPMENT: The following
// is not operational yet
// Tell the compiler that  base language
// code should not appear
strictness modelOnly;

// Tell the compiler that the only base
// language code could be in method bodies
// Any other unparsable 'extra code' in classes
// will be rejected.
strictness noExtraCode;
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

// A high level of strictness will cause warnings to be issued when base language code
// is found, where it might not have been intended. Strictness can also be used
// To either suppress certain messages, or to declare that they should be present.
// NOTE: This is currently under development
strictness- : strictness

```
( [=strictness:modelOnly|noExtraCode|none]
  | [[strictnessMessage]]
  | [[strictnessDisableAuto]]
) ;

strictnessMessage : [=message:allow|ignore|expect|disallow] [messageNumber]

strictnessDisableAuto : disable [**expression]
```

Class Definition

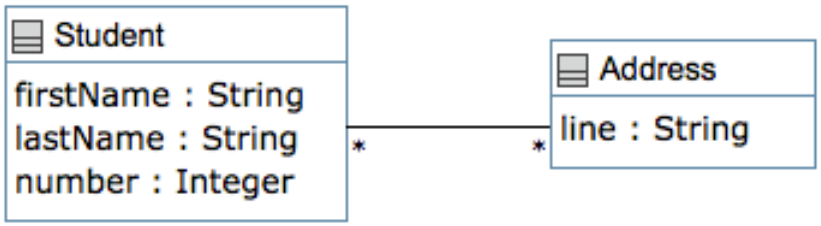
A class definition defines an object-oriented class available for use as a type in the system you are developing. Objects (chunks of data) are created as *instances* of the class. A class describes the structure of that data in terms of attributes (simple data like strings and numbers), associations (links to and from other objects), state machines, as well as many other things described in this manual.

To define a class in Umple, specify the keyword 'class', followed by the name of the class (starting in a capital letter in order to respect naming conventions as well as to avoid a warning) and then the body of the class within curly brackets.

The body can contain various elements that are listed in the Class Content page.

Traits can be used to add the same set of items to several unrelated classes..

The following UML diagram shows two classes: a Student class and an Address class, linked by an association. The corresponding Umple is below.



Example Showing two Classes (Class Diagram is Above)

view plain

```
01. class Student
02. {
03.   firstName;   // attribute - defaults to String
04.   lastName;
05.   Integer number;  // attribute with type Integer
06.   * -- * Address;  // Many-to-many association
07.
08.   // Method, whose content is not processed by Umple
09.   public String fullName()
10.   {
11.     return getFirstName() + " " + getLastName();
12.   }
13. }
14.
```

```
15. class Address
16. {
17.   String[] line;  // Multi-valued attribute
18. }
19.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

YouTube Video with Additional Explanation

Timothy Lethbridge





Watch on

Syntax

// Classes are the most common elements in Umple.
// See user manual page ClassDefinition
classDefinition : class [name] { [[classContent]]* }

// The following items can be found inside the body of classes or association classes
classContent- : [[comment]]

| [[reqImplementation]]
| [[innerClass]]
| [[mixsetDefinition]]
| [[distributable]]
| [[proxyPattern]]
| [[strictness]]
| [[classDefinition]]
| [[trace]]
| [[emitMethod]]
| [[templateAttributeDefinition]]
| [[primitiveDefinition]]
| [[portDefinition]]
| [[portBindingDefinition]]
| [[position]]
| [[displayColor]]
| [[abstract]]
| [[keyDefinition]]
| [[softwarePattern]]
| [[depend]]
| [[symmetricReflexiveAssociation]]
| [[attribute]]
| [[testCase]]
| [[genericTestCase]]
| [[testSequence]]
| [[testClassInit]]
| [[stateMachine]]
| [[activeMethodDefinition]]
| [[inlineAssociation]]
| [[concreteMethodDeclaration]]
| [[constantDeclaration]]
| [[modelConstraint]]
| [[invariant]]
| ;
| [[enumerationDefinition]]

| [[exception]]
| [[extraCode]]

interface Definition

An interface defines a list of abstract methods that are to be implemented in one or more classes. An interface can be composed of the following elements:

- Dependencies
- Method declarations (methods with no body)
- isA clauses (placing interfaces in an inheritance hierarchy)

To implement an interface in a class, or to create subinterfaces, use an isA clause.

Umple also supports a feature called traits, that has some similarities to interfaces. However, traits concrete methods, attributes and state machines to also be contained in them.

In the following example, RegisterCapable is an interface that defines a single abstract method registerForCourse(). This is implemented concretely by CorporateClient and IndividualStudent.

Example

view plain

```
01. interface RegisterCapable
02. {
03.   depend school.util.*;
04.
05.   boolean registerForCourse(Course aCourse);
06. }
07.
08. class Person {
09.   name;
10. }
11.
12.
13. class CorporateClient {
14.   isA RegisterCapable;
15.   boolean registerForCourse(Course aCourse) {
16.     // write code here
17.   }
18.   0..1 <- * Person employees;
19. }
20.
21. class IndividualStudent {
```

```
22.   isA Person, RegisterCapable;
23.   boolean registerForCourse(Course aCourse) {
24.     // write code here
25.   }
26. }
27.
28.
29. class Course
30. {
31.   name;
32.   description;
33.   * -- * Person registrants;
34. }
35.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

// An Interface can only contain method. See interfaceDefinition
interfaceDefinition : interface [name] { [[depend]]* [[interfaceBody]] }

// Interfaces: Note that if the format of an abstractMethodDeclaration is not
// followed, then the body will extraCode and passed to the base language
// See user manual page interfaceDefinition
interfaceBody- : [[interfaceMemberDeclaration]]*

interfaceMemberDeclaration : [[comment]]
| [[reqImplementation]]
| [[constantDeclaration]]
| [[constantDeclarationDeprecated]]
| [[abstractMethodDeclaration]]
| [[position]]
| [[displayColor]]
| [[isA]]
| [[interfaceTest]]
| [[distributableInterface]]

| [[exception]]
| [[extraCode]]

Class Content

A class can contain any of the following items. Interfaces are limited to those indicated as [Allowed in interfaces]. A trait can contain most of the items, indicated as indicated as [Allowed in traits].

- Comments describe the intent of your class (or interface), and any other elements in the class.[Allowed in interfaces] [Allowed in traits]
- isA directives that specify a superclass to your class (e.g. a Supervisor could be described as "isA Person"). A class can only have one isA clause referring to another class, but can have many referring to traits or interfaces. [More than one allowed in interfaces] [More than one allowed in traits]
- An **abstract;** keyword, specifying that the class is abstract (cannot be instantiated, must have at least one concrete subclass).
- Depend directives that describe external dependencies that might be used by your class within methods [Allowed in interfaces] [Allowed in traits].
- Attributes describing simple data that can be found in any element of the class (i.e. name, dob, amount) [Allowed in traits]. These can be preceded by the keyword **const** to declare constants.
- Inline associations that define links to instances of other classes (or to other instances of this class). [Allowed in traits] (can also be specified outside classes)
- Methods that provide behaviour for the class. [Allowed in interfaces, but with no method body] [Allowed in traits] (state events are also methods, and methods can also be specified in states)
- Enumerations for use inside this class (can also be specified outside classes)
- State machine definitions that describe declaratively attributes that change value in certain behaviour patterns in response to actions such as method calls. [Allowed in traits]
- A singleton directive that limits the system to only one instance of this class (e.g. there may only be one Bank in a banking system)[Allowed in traits]

- An immutable directive that ensures that once constructed, an instance cannot be modified (under development).[Allowed in traits]
- Key directives indicating which attributes (or in the future, associations) are going to be used to define when one element is equal to another. [Allowed in traits]
- **Generation templates** and **emit methods** for generating string output.
- **Invariant constraints**. These are Boolean expressions that limit the values of attributes and other elements. [Allowed in traits] (other types of constraints can be specified in methods)
- Mixset definitions that are named blocks containing any of the other items that can appear in classes (i.e. those listed on this page). These allow for conditional compilation because the content of the mixset an either be active or not active. Mixsets are activated by use statements or command line arguments.[Allowed in traits]
- **Class definitions nested inside this class** that define subclass of this class as an alternative to using the isA clause. (these are not inner classes; those are defined as below)
- Trace directives to direct generation of traces from the code. [Allowed in traits]
- An inner class, specified either by the keyword **inner** or **static**, followed by a class definition. These are only relevant to Java generation.
- Several other items to be documented later including the ability to colour classes in diagrams, and to specify test cases.

Syntax

```
// The following items can be found inside the body of classes or association classes
classContent- : [[comment]]
   | [[reqImplementation]]
   | [[innerClass]]
   | [[mixsetDefinition]]
   | [[distributable]]
   | [[proxyPattern]]
```

| [[strictness]]
| [[classDefinition]]
| [[trace]]
| [[emitMethod]]
| [[templateAttributeDefinition]]
| [[primitiveDefinition]]
| [[portDefinition]]
| [[portBindingDefinition]]
| [[position]]
| [[displayColor]]
| [[abstract]]
| [[keyDefinition]]
| [[softwarePattern]]
| [[depend]]
| [[symmetricReflexiveAssociation]]
| [[attribute]]
| [[testCase]]
| [[genericTestCase]]
| [[testSequence]]
| [[testClassInit]]
| [[stateMachine]]
| [[activeMethodDefinition]]
| [[inlineAssociation]]
| [[concreteMethodDeclaration]]
| [[constantDeclaration]]
| [[modelConstraint]]
| [[invariant]]
| ;
| [[enumerationDefinition]]
| [[exception]]
| [[extraCode]]

traitContent- : [[mixsetDefinition]]
| [[testCase]]

| [[requiredModelElements]]
| [[comment]]
| [[reqImplementation]]
| [[traitDefinition]]
| [[trace]]
| [[position]]
| [[displayColor]]
| [[abstract]]
| [[keyDefinition]]
| [[softwarePattern]]
| [[depend]]
| [[symmetricReflexiveAssociation]]
| [[attribute]]
| [[stateMachine]]
| [[inlineAssociation]]
| [[concreteMethodDeclaration]]
| [[abstractMethodDeclaration]]
| [[constantDeclaration]]
| [[invariant]]
| ;
| [[exception]]
| [[extraCode]]

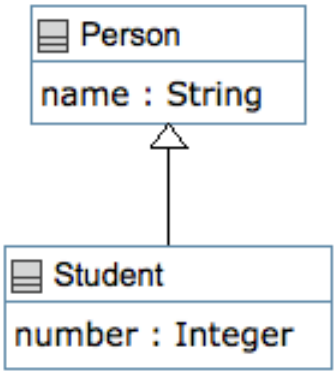
isA clause

The isA keyword is used to denote an inheritance relationship (generalization) between two classes, or a class and the interfaces it implements, or a class and the traits it includes.

This corresponds to keywords such as 'extends', 'subclass', etc. in other languages. The isA keyword was chosen so as to be independent of other languages, and due to the strong conceptual similarity between interfaces, classes and traits.

Note that it is possible to avoid using the isA keyword for class generalization, by directly embedding a subclass inside a superclass. Note that this does *not* create an inner class in the Java sense, but instead creates a subclass. The two examples below give identical results.

The following is how a generalization appears in UML. The corresponding Umple is below. Note that in UmpleOnline, the expected layout for generalizations places superclasses above subclasses.



Example Showing Student as a Subclass of Person

view plain

```
01. // A superclass-subclass relationship defined using the isA keyword
02. class Person
03. {
04.   name;
05. }
06.
07. class Student
08. {
09.   isA Person;
10.   Integer number;
```

```
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

The Same Model as Above but Using Nesting Instead of IsA

view plain

```
01. // A superclass-subclass relationship defined
02. // by embedding the subclass in the superclass
03. class Person
04. {
05.   name;
06.
07.   class Student
08.   {
09.     Integer number;
10.   }
11. }
12.
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

// Generalization and inheritance isAclause
isA- : [[singleIsA]] | [[multipleIsA]]

singleIsA- : isA [[isAName]] (, isA [[isAName]])* ;

multipleIsA- : isA [[isAName]] (, [[isAName]])* ;

Depend clause

The depend clause is used to indicate a dependency to another namespace, class or interface.

See also this example of how to use the depend clause to arrange for Umple to work with external code.

Example

view plain

```
01. namespace sports.core;
02.
03. class Equipment
04. {
05.   name;
06.   Integer weight;
07. }
08.
09. namespace sports.baseball;
10.
11. class Baseball
12. {
13.   depend sports.core.*;
14.
15.   public boolean isRequired(Equipment eq)
16.   {
17.     return eq.getName().equals("bat");
18.   }
19. }
20.
21.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
// Depend clause. See user manual page Dependclause
depend- : depend [depend] ;
```

Custom Constructor

Custom constructors should normally not be provided in Umple. Consider using before and after keywords to adjust what the automatically-generated constructor does, or adjusting constructor parameters using keywords like lazy.

Umple generates its own constructors: The arguments are composed from the list of attributes in the class (in the order provided), as well as any associations with a 1 multiplicity at the other end. To avoid having an argument in the generated constructor, designate an attribute as 'lazy' or specify association multiplicity to be 0..1 instead of 1. Umple also generates code to constrain the value of what appears in each constructor if a constraint is specified.

Attribute Definition

An attribute represents some information held by a class.

The attribute can have many properties, and can be defaulted to a certain value.

It is important to distinguish an attribute from the concept of a 'field' or 'instance variable' in Java or another programming language. An attribute is a UML/Umple entity that represents simple data. In Java it will become a field, but there will also be methods associated with the Umple attribute to get it, set it, and constrain its value. An attribute may also automatically be added to the argument list of the constructor and constructed in the constructor. In addition to representing attributes, Java fields also represent the ends of associations. Both attributes and associations should be considered more abstract than fields.

A summary of the generated methods for each attribute can be found on the API summary page.

Umple state machines are a special kind of attribute, and Umple also allows you to inject code that will constrain or alter the values of attributes using aspect-oriented techniques.

The example below shows the basic properties of attributes.

Example

view plain

```
01. class Group
02. {
03.   // Simple Umple Integer. Note that code generation
04.   // in different languages will use the simplest
05.   // native type in that language. In Java it will
06.   // use int.
07.   // The initial value must be supplied through a
08.   // constructor argument.
09.   // The value can later be accessed through set and
10.   // get methods (here setI and getI).
11.   Integer i;
12.
13.   // const: Declares a constant (static final in Java).
14.   const Integer Max = 100;
15.
16.   // immutable: A constructor argument is required so
17.   // it can be set at construction time; cannot be
18.   // changed after that since no set method is
```

```
19.   // generated.
20.   immutable String str;
21.
22.   // lazy: A constructor argument is not required.
23.   // Numbers are initialized to zero.
24.   // Objects (including Strings) are initialized to null,
25.   // Booleans are initialized to false.
26.   lazy Time t;
27.   lazy Boolean b;
28.   lazy s;
29.
30.   // settable: Set using a constructor argument and
31.   // can be changed after that. This is the default,
32.   // so the settable keyword can be omitted.
33.   settable Date d;
34.
35.   // internal: No getter and setter methods created.
36.   // Only for private use in internal methods. However
37.   // in this case it is initalized using =  The code
38.   // following = is in the 'base' language, here Java.
39.   internal Time t2 = new Time(
40.     System.currentTimeMillis());
41.
42.   // unique: Every new object created must have a
43.   // unique new value assigned.
44.   // You can get the value. You can set it as well,
45.   // but it has to be unique.
46.   unique u;
47.
48.   // autounique: Every new object created will have a
49.   // new integer assigned.
50.   // You can get the value (an integer) but not set it.
51.   autounique x;
52.
53.   // The value is initialized as shown in the constructor.
54.   // There is no constructor argument generated.
55.   String q = "chicken";
56.
57.   // defaulted: Set in the constructor to the default,
58.   // and can be reset to the default any time by
59.   // calling a reset method (here resetP()).
60.   // Can also be set to any other value using setP().
61.   // The default can be queried by calling getDefaultP().
62.   defaulted String p = "robot";
63.
```

```
64.   // Similar to the above, except this shows that if
65.   // no type is given, then the default type is String.
66.   defaulted r = "";
67. }
68. // @@@skipcppcompile
69.
70.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
// Details of attributes. See user manual page AttributeDefinition
attribute : [[simpleAttribute]]
   | [[autouniqueAttribute]]
   | [[multivaluedAttribute]]
   | [[derivedAttribute]]
   | [[complexAttribute]]

simpleAttribute- : [~name] ;

autouniqueAttribute- : [=autounique] [~name] ;

multivaluedAttribute- : [=unique]? [=lazy]? [=ivar]? [=modifier:immutable
   |settable
   |internal
   |defaulted
   |const
   |fixml]? [type]? [[list]] [~name]
  (= { [**value] }
  )? ;

derivedAttribute- : [=modifier:immutable
   |settable
   |internal
   |defaulted
```

```
  |const
  |fixml]? [[typedName]] =
( [[moreCode]]
)+

complexAttribute- : [=unique]? [=lazy]? [=ivar]? [=modifier:immutable
  |settable
  |internal
  |defaulted
  |const
  |fixml]? [[typedName]]
(= [**value]
)? ;
```

Umple Builtin Data Types

The following example illustrates the data types available for Umple attributes. Except as specified, they will generate primitive datatypes in Java.

Initialization of values can be performed in a manner similar to Java or C++. Initialization of Dates uses yyyy-mm-dd format, and initialization of Times use hh:mm:dd format.

Example

view plain

```
01. class Demo
02. {
03.   Integer i;
04.   Integer i2 = 5;
05.
06.   String str;  // the default if no type is specified
07.   str2;
08.   str3 = "";
09.
10.   Float flt;
11.
12.   Double dbl;
13.   Double dbl2 = 8.0;
14.
15.   Boolean bln;
16.   Boolean bln2 = false;
17.
18.   Date dte; // In Java. uses the java.sql.Date class
19.   Date dte2 = "2018-09-25";
20.
21.   Time tme; // In Java. uses the java.sql.Time class
22.   Time tme2 = "14:56:50";
23. }
24.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example Using a Native Java Type Rather than a Builtin Umple Type

view plain

```
01. class DemoNonUmpleType
02. {
03.   // If you use a non-umple type that is specific to a base
04.   // programming language, then code will be generated
05.   // consistent with that type, but generation of code
06.   // independent of base language is no longer possible
07.   int demoJavaInt;
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Classes as Attribute Types

You can declare a class as a data type of an attribute. This allows for declaration of a 'containment' relationship.

If the data type you are using for an attribute is an Umple class, users should also consider using an association, and particularly a composition instead. This allows drawing of diagrams and more consistency checks by the Umple compiler.

If you use the methods in the Umple-generated API to access the object stored in such an attribute, and pass this contained object to some remote subsystem, then that remote subsystem will be able to affect the containing object in a backhanded sort of way. This is considered *content coupling* in software engineering, and should be carefully controlled or avoided.

In the examples below, we show how using attribute notation is very similar to using a directed association, except that with attribute notation, you can set the value to null.

Example

view plain

```
01. // This first example uses attribute notation
02. // The address is required on the constructor, but
03. // can be null. An instance of Address can be
04. // considered to be contained within the
05. // Person class.
06. //
07. // Although there is nothing currently preventing
08. // the same address instance from being re-used in
09. // multiple classes, it is strongly suggested to
10. // avoid that. If you want to reuse the same
11. // Address, then use association notation instead.
12. class Person {
13.   name;
14.
15.   // The following uses a class as the attribute type
16.   Address address;
17. }
18.
19. class Address {
20.   street;
21.   city;
22.   postalCode;
```

```
  country;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Alternative to the Above that Uses an Association

view plain

```
// Contrast this example with the previous one
// Here we use a directed association instead of
// an attribute.
// Note that the multiplicity on the left is of no
// relevance. It is conventional to show it as *
// to indicate that the value could be attached to
// several objects.
//
// Here when you construct a Person, the address
// cannot be null.
class Person {
  name;

  * -> 1 Address;
}

class Address {
  street;
  city;
  postalCode;
  country;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Alternative with an Association Having Multiplicity Indicating Optional

view plain

```
// In this example, we have made the address
```

```
// optional. Now, it does not appear on the
// constructor. However when you add an address it
// still cannot be null.
//
// Note also that in this example, the role name
// 'address' is left off to show  it is optional.

class Person {
  name;

  * -> 0..1 Address;
}

class Address {
  street;
  city;
  postalCode;
  country;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

MultiValued Attributes

As in UML, Umple allows one to specify an attribute with multiple values.

We encourage the use of association notation in this context, however the attribute notation can be useful sometimes.

Initialization of multivalued attributes is also allowed, as shown in the example.

Example

view plain

```
01. class Office {
02.   Integer number;
03.   Phone[] installedTelephones;
04.   String[] emails = {"abc@umple.org", "def@umple.org"};
05.   Integer [] incomingNumbers = {765432, 987654};
06. }
07.
08. class Phone {
09.   String digits;
10.   String callerID;
11. }
12. // @@@skipcppcompile
13.
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
multivaluedAttribute- : [=unique]? [=lazy]? [=ivar]? [=modifier:immutable
   |settable
   |internal
   |defaulted
   |const
   |fixml]? [type]? [[list]] [~name]
 (= { [**value] }
 )? ;
```

Lazy Immutable Attributes

If you want to avoid having an attribute change after it is initially set, but do not want to have an argument in the constructor, then use the combination of keywords 'lazy immutable'.

You can call the set method just once on such an attribute. The set method will return false if you try again. This is useful for interacting with architectures where objects are constructed for you, so you have no ability to specify constructor arguments.

Note that if the lazy keyword is omitted, then there will be no set method and an argument will be present in the constructor to initialize the attribute. See also the immutable pattern.

Example

```
view plain

01. class A
02. {
03.     lazy immutable z;
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
complexAttribute- : [=unique]? [=lazy]? [=ivar]? [=modifier:immutable
   |settable
   |internal
   |defaulted
   |const
   |fixml]? [[typedName]]
  (= [**value]
  )? ;
```

Derived Attributes

When declaring an attribute in Umple, you can specify an arbitrary expression after the equals sign to create an attribute that will be computed. There will be no *set* method on such an attribute.

Note that unless you use the simplest of expressions, you will be limited to only being able to generate code for the language of the expression.

You should make sure you call the *get* methods provided in the Umple-generated API (rather than directly accessing variables) and avoid having any side-effects in your expressions. Currently this is not enforced, but may be in the future.

For other examples of derived attributes see the sections on the Delegation pattern and sorted associations.

Example

view plain

```
01. class Point
02. {
03.    // Cartesian coordinates
04.    Float x;
05.    Float y;
06.
07.    // Polar coordinates
08.    Float rho = {
09.      Math.sqrt(Math.pow(getX(), 2)
10.      + Math.pow(getY(), 2))
11.    }
12.    Float theta =
13.      {Math.toDegrees(Math.atan2(getY(),getX()))}
14. }
15. // @@@skipcppcompile
16.
17.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example with the Language of the Expression Being Specified

```
view plain

01. class Rectangle {
02.   Double height;
03.   Double width;
04.   Double area =
05.     Java {height*width} Php {$height*$width}
06. }
07. // @@@skipcppcompile
08.
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

derivedAttribute- : [=modifier:immutable
|settable
|internal
|defaulted
|const
|fixml]? [[typedName]] =
([[moreCode]]
)+

Unique Attributes

Attributes in Umple can be **unique**. Applying the unique keyword to an attribute ensures that each instance of the class in the currently running program has a unique value for that attribute.

In the first example, each player has a unique ranking. The ranking of a player can change but two players cannot have the same ranking.

The **autounique** keyword can be used to automatically assign a unique Integer to every new instance created (example 2).

Certain runtime errors can occur if attempts are made to violate uniqueness.

Example 1

view plain

```
01. // Every player has a ranking.
02. // Rankings can be modified,
03. // but two different players
04. // can not have the same ranking.
05.
06. class Player
07. {
08.   unique Integer ranking;
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 2

view plain

```
01. // Unique registration numbers (Integer)
02. // are automatically assigned to every
03. // new registration created
04.
05. class Registration
06. {
07.     autounique registrationNumber;
08. }
```

```
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
autouniqueAttribute- : [=autounique] [~name] ;

complexAttribute- : [=unique]? [=lazy]? [=ivar]? [=modifier:immutable
   |settable
   |internal
   |defaulted
   |const
   |fixml]? [[typedName]]
  (= [**value]
  )? ;
```

Enumeration Definition

The Umple language allows developers to define enumerations for either a single class, or to share between classes. An enumeration is a user-defined data type that is a set of constants.

As shown in the examples below, enumerations require the keyword "enum" followed by the name of the enumeration, and its values. If duplicate enumerations are detected, E095 Duplicate Enumerations will be raised. If enumerations outside of class bodies have the same name as the enumerations within class bodies, the enumerations within class bodies will be used in the generated code.

Model-level enumerations will be added to classes when they do not already have enumerations defined with the same name, and the enumeration is detected in attribute types, method return types, method parameter types, and/or event parameter types.

Example

view plain

```
01. // In this example, the "Status" enumeration
02. // is defined for the "Student" class
03.
04. class Student {
05.   enum Status { FullTime, PartTime }
06.   Status status;
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example of Enumeration Shared by Two Different Classes

view plain

```
01. // In this example, the "Status" enumeration
02. // is shared by the "GradStudent"
03. // and "UndergradStudent" class
04.
05. enum Status { FullTime, PartTime }
06.
```

```
07. class GradStudent {
08.   Status status;
09. }
10.
11. class UndergradStudent {
12.   Status status;
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example of Enumerations Defined for a Single Class or Shared between Different Classes

view plain

```
01. // In this example, the "Status" enumeration
02. // is shared by the "GradStudent"
03. // and "UndergradStudent" class
04. // The "Semester" enumeration is only defined
05. // for the "UndergradStudent" class
06.
07. enum Status { FullTime, PartTime }
08.
09. class GradStudent {
10.   Status status;
11. }
12.
13. class UndergradStudent {
14.   enum Semester { Spring, Summer, Fall, Winter }
15.   Status status;
16.   Semester semester;
17. }
18.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example with Enumerations Defined Within and Outside of Class Bodies

view plain

```
01. // In this case, the "Month" enumeration
02. // defined in class A will be used in
03. // the generated code
04.
05. namespace example;
06.
07. enum Month {x,y,z}
08.
09. class A{
10.   enum Month {o,p,q}
11.   Month m;
12.   Month p;
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

// Enumerations can be declared outside the body of classes
// See user manual page EnumerationDefinition
enumerationDefinition : enum [name] { [enumValue](, [enumValue])* }

Association Definition

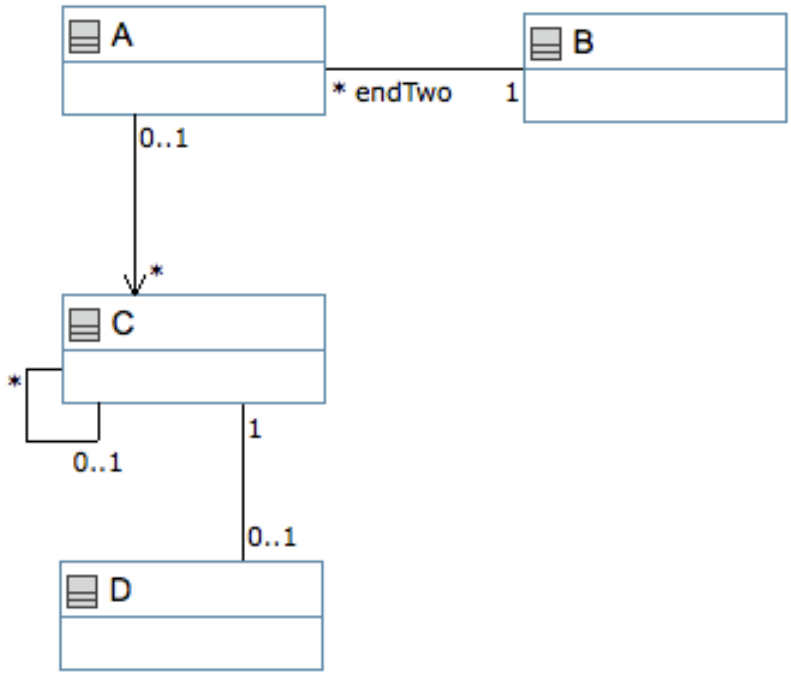
An association defines a relationship from a class to another class. More specifically, it defines which links (i.e. references or pointers) may exist at run time between instances of the classes.

Umple supports binary associations (associations with just two ends). This definition includes reflexive associations, in which both ends are the same class.

An association can specify the following information:

- The classes involved (i.e. one or two classes).
- Navigability: -- means both ways (i.e. each class can access the linked objects of the other class) and -> means one way.
- The restrictions on the number of objects allowed in the relationship (multiplicity).
- Role names on association ends (to clarify the relationship and avoid collision if two classes are associated in multiple ways).

The following is a UML diagram showing associations. The corresponding Umple is at the end of the page.



Associations can be presented in two ways in Umple.

- Inline - Defined within one of the associated classes. See line 7 of the example below; the association is defined between A and B.
- Independently - defined as a first-class entity in the model. See line 18 of

the example below; the association is defined between A and C.

There are several special kinds of associations in Umple.

- Reflexive - A class that is associated with itself. See line 12 of the example below; the association is defined between C and itself.
- Compositions - A composition is a sub-type of association where there is mandatory delete on the composition end, regardless of the multiplicity.
- External - To associate a class to class in non-umple code. See lines 13 and 27 of the example below; the association is defined between C and a class D that doesn't appear in this model and is shown as an external reference. Class D must be separately linked in; it could also be generated by Umple or could be hand-written. For more on the use of the external keyword, see this example.
- Association Class - A shorthand for two one-to-many associations to a class that contains data that relates the two classes. The two related classes are logically in a many-many relationship (this is currently under development).

Umple will report an error if an association refers to a non-existent class.

Example

view plain

```
01. // Example code illustrating various
02. // kinds of associations
03. class A {}
04.
05. // Class with inline association having role name
06. class B {
07.   1 -- * A endTwo;
08. }
09.
10. // Class with reflexive association
11. class C {
12.   0..1 -- * C;
13.   1 -- 0..1 D; // D is external
14. }
15.
16. // Independently defined and directed association
17. association {
18.   0..1 A -> * C;
19. }
20.
```

```
21. // Class with composition
22. class E {
23.   0..1 <@>- * A;
24. }
25. 
26. // Reference to a class defined elsewhere
27. external D {}
28. 
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Multiplicity

Multiplicity describes the allowable number of entities that can participate in one end of an association. In most cases you provide both a lower and upper bound, but the "many" (or "*") case assumes a zero lower bound.

The most common cases are:

- * meaning 'any number' or 'many'; i.e. from 0 to whatever number the computer can handle.
- 1 meaning 'mandatory', i.e. exactly one.
- 0..1 meaning 'optional', i.e. a lower bound of zero, and an upper bound of one.
- 1..* meaning 'mandatory many', i.e. a lower bound of 1 and an indeterminate upper bound.

Multiplicity must be specified for both ends of an association. At run time, code generated by Umple will ensure that the lower bounds and upper bounds are respected. Also the multiplicity determines which methods will be generated in the API for the model. For example, when * is specified, methods are generated to access all the associated objects, and to access an object at a specific index. The number of objects linked at run-time is called 'cardinality'.

If multiplicity is specified incorrectly, the compiler will generate an error message highlighting the line with the multiplicity error.

'Mandatory many' example

view plain

```
01. // When multiplicity is given as *, which is the
02. // same as 0..* there can be any number of links
03. // from an instance at the other end of the
04. // association to instances at this end
05. //
06. // The lower bound is zero and the upper bound is
07. // 'many'
08. class A
09. {
10.   // an instance of A has many B's
11.   1 -- * B;
12.
13.   // An instance of C has many A's
```

```
  * -- 1 C;
}

class B {} class C {}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example with lower and upper bounds

view plain

```
// When the mutiplicity is shown as two integers
// separated by .. then the first integer is the
// lower bound, and the second integer is the
// upper bound.
//
// Here, at one end of the association
// the lower bound is 0..1 (which means 'optional'
// and at the other end of the association
// the lower bound is 3 and the upper bound is 5
class D {
  0..1 -- 3..5 E;
}

class E{}
// @@@skipcppcompile

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example with single integer multiplicity

view plain

```
// When the multiplicity is a single integer there
// must be exactly that number of objects linked
// at all times (including when the object at the
// other end is first created). Except for '1',
// such multiplicities are rare.
```

```
06. //
07. // Here, there must be exactly two objects (lower
08. // and upper bound are both 2)
09. class F {
10.   0..1 -- 2 G;
11. }
12.
13. class G{}
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

multiplicity- : [!lowerBound:\d+|[**]] .. [!upperBound:\d+|[**]] | [!bound:\d+|[**]]

Inline Associations

An association represents a relationship between two classes. Associations can be defined within a class, in a similar manner as you would define an attribute.

Contrast this with the same model defined using independently defined associations.

Example

view plain

```
01. class Group
02. {
03.   // a many-to-many association
04.   // An item has zero or more groups and a group
05.   // has zero or more items
06.   * -- * Item;
07.
08.   // An item has an optional description
09.   // The association is directed, so descriptions
10.   // do not know which groups link to them
11.   1 -> 0..1 Description;
12. }
13.
14. class Item
15. {}
16.
17. class Description
18. {}
19.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

inlineAssociation : [=modifier:immutable]? [[inlineAssociationEnd]] [=arrow:--
|->
|<-
|><
|<@>-

|-<@>] [[associationEnd]] ;

inlineAssociationEnd : [[multiplicity]] [~roleName]? [[isSorted]]?

associationEnd : [[multiplicity]] [type] [~roleName]? [[isSorted]]?

Independently Defined Associations

An association can be defined separately from any class. Contrast this with the Umple code showing the same model with inline associations.

Specifying an association independently of the two associated classes, as in the example below, can sometimes make code clearer. Specifying the association inline in one of the two classes can be clearer in other cases, particularly when it is defined in the more important class. But the decision regarding which alternative to use is left to the designer.

Example

view plain

```
01. class Group { }
02. class Item {}
03. class Description {}
04.
05. association {
06.   * Group -- * Item;
07. }
08.
09. association {
10.   1 Group -> 0..1 Description;
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
// Associations can be declared outside the body of classes.
// See user manual page IndependentlyDefinedAssociations
associationDefinition : association [name]? {
  ([[comment]]
    | [[reqImplementation]]
    | [[mixsetDefinition]]
    | [[association]]
  )* }
```

```
association : [=modifier:immutable]? [[associationEnd]] [=arrow:--
   |-> 
   |<-
   |><
   |<@>-
   |-<@>] [[associationEnd]] ;

associationEnd : [[multiplicity]] [type] [~roleName]? [[isSorted]]?

association : association [associationNum];
```

Role Names

A role name is an additional name attached to an association. In the following example, the word 'supervisor' could be omitted, but it clarifies that the graduate student's professor is called his or her supervisor.

Example

view plain

```
01. class Person {
02.   name;
03. }
04.
05. class GraduateStudent {
06.   isA Person;
07.   Integer id;
08.   * -- 0..2 Professor supervisor;
09. }
10.
11. class Professor {
12.   isA Person;
13.   rank;
14. }
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

One-to-One Associations

The examples of associations in the previous user manual pages all showed cases where the lower-bound of the multiplicity at both ends is zero. For example there can be zero graduate students supervised by a professor, and zero professors might be assigned to supervise a graduate student initially.

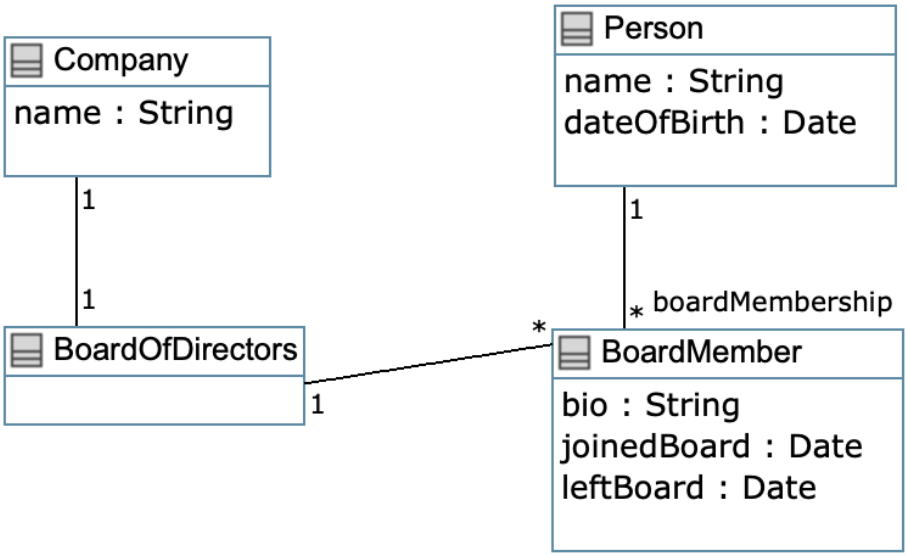
Allowing a lower bound of zero means that one object can be created without having to first create the object(s) it is to be linked to. The vast majority of associations have a lower bound of zero at at least one of the two ends.

However there are a few situations where the lower bound at both ends of the association ought to be greater than zero. Here we will consider the one-to-one case. In the example below, a Company must always have one BoardOfDirectors and a BoardOfDirectors must always belong to one Company. But which should be created first? If we create the Company first, then a multiplicity constraint would be violated since for a period of time it would lack a Board. One solution would be to relax the constraint and decide that the association should not be 1-1. However Umple *does* allow 1-1 associations and creates a special API to allow both objects to be created at the same time.

The generated code has a constructor with the arguments normally found in the constructors of *both* associated classes. So in the following main program, the call to the constructor of Company with just a single argument (company name) will construct both the Company and the BoardOfDirectors and link them (the BoardOfDirectors class normally takes no argument on its constructor). If the constructor of BoardOfDirectors had had any arguments they could have been provided to the constructor of Company. The constructor of Board of Directors also could have been used to create both objects simultaneously.

It is worth noting that when a one-to-one association exists, there must always exist in the running system an equal number of both instances of both associated classes, and they must be linked in pairs.

Another observation is that modellers tend to over-use one-to-one associations. A superior modeling solution might be to merge the two classes; in this case just have a Company class.



An Executable Example Showing the Use of a One-to-One Association

view plain

```
01. // Core class of the example
02. // A company is a corporation
03. class Company {
04.   name;
05.
06.   // A company must have a BoardOfDirectors
07.   // that must be created at the same time
08.   // as the Company itself
09.   1 -- 1 BoardOfDirectors;
10. }
11.
12. // BoardOfDirectors is in a 1-1 relationship
13. // with Company meaning that theoretically it
14. // could just be merged into the Company class
15. // However to facilitate separation of concerns
16. // it is sometimes best to keep such classes
17. // separate
18. class BoardOfDirectors {
19.   // A BoardMember is just a member of one
20.   // board (but a person can be on multiple
21.   // boards using multiple BoardMembers)
22.   1 -- * BoardMember;
23. }
24.
25. // A board must have a set of members
```

```
// Former board members tracked
class BoardMember {
  bio; // Brief description of background
  lazy Date joinedBoard;
  lazy Date leftBoard;

  // One of the roles a person could play would
  // be as a Board member
  * boardMembership -- 1 Person;
}

// Generic Person class
class Person {
  name;
  Date dateOfBirth;
}

// Mixin with a main program demonstrating
// manipulation of 1--1 and other associations
class Company {
  depend java.sql.Date;

  public static void main(String [] args) {

    // First create some instances of Person
    // They are initially not on any boards
    Person p1 = new Person("Alice",
      Date.valueOf("1990-01-01"));
    Person p2 = new Person("Bob",
      Date.valueOf("1991-02-02"));

    // Create a Company using the simpler of its
    // two constructors that doesn't require a
    // board to already exist. This will actually
    // create the BoardOfDirectors at the same time
    Company c = new Company("UmpleCorp");

    BoardOfDirectors b = c.getBoardOfDirectors();

    b.addBoardMember("Largest Shareholder", p1);
    b.addBoardMember("Founder", p2);

    // Output the results to prove that this works
    System.out.println("Key company info: "+c);
    System.out.println("Board members: "+
```

```
71.         b.getBoardMembers());
72.
73.     }
74.   }
75.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

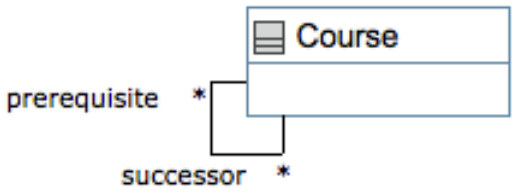
Reflexive Associations

A reflexive association is an association from a class to itself. There are two main types: Symmetric and Asymmetric.

Asymmetric Reflexive Associations: The ends of the association are semantically different from each other, even though the associated class is the same. Examples include parent-child, supervisor-subordinate and predecessor-successor.

The first and second examples below show courses having prerequisites that are other courses. It is necessary to specify a role name at one or both ends of the association to distinguish the ends. Also it is necessary to ensure that the lower bound is always zero on both ends. The reason for this is that otherwise an illogical situation would arise: For example, if you said that each course must have one or more pre-requisites, then what about the very first courses in the hierarchy? The compiler will report an error if the lower bound on a multiplicity is one or higher.

The following is how an asymmetric reflexive association appears in UML, as generated by UmpleOnline. The corresponding Umple code is in the first example below.



Symmetric Reflexive Associations: There is no logical difference in the semantics of each association end. The fourth example shows one of these: A set of courses that are mutually exclusive with each other essentially make up a set. If course A is mutually exclusive with B, then course B is mutually exclusive with A. In other words, students who have taken one course cannot take another in the set. Umple uses the keyword 'self' to identify this case. Note that at the current time the asociaton itself does not appear in UmpleOnline; this will be fixed.

Asymmetric association with role names on both ends

view plain

```
01. // Example asymmetric association with role names
```

```
// on both ends
class Course {
  * successor -- * Course prerequisite;
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Asymmetric association with role names on one end

view plain

```
// Example asymmetric association with role name
// on one end
class Course {
  * -- * Course prerequisite;
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Asymmetric association with role names on the other end

view plain

```
// Example asymmetric association with role name
// on the other end
class Person {
  0..2 parents -- * Person;
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Symmetric association

```
view plain

01. // Example symmetric association. Note the use of
02. // the keyword self
03. class Course {
04.   * self  isMutuallyExclusiveWith;
05. }
06.
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

symmetricReflexiveAssociation : [[multiplicity]] self [roleName] ;

Compositions

A composition is a subtype of association where the delete is mandatory regardless of the multiplicity (enforced delete).

Compositions are used for associations where once the objects are associated, it makes sense for the deletion of one to cause the deletion of the other. This differs from the pre-existing associations, where the delete is dependent on the multiplicity.

For example, consider the case of a Vehicle and Wheel classes. With a regular association, the code would be as follows (where a Wheel can be associated to at most 1 Vehicles, and a Vehicle can have between 2 and 4 Wheels).

See **Example 1**

Here, the delete code for Vehicle does not delete the associated Wheels, since the multiplicity on Wheel is 0..1 which means the Wheel can exist without a Vehicle. However, on deleting a Vehicle it would make more sense for the associated Wheels to be deleted.

This is where compositions are useful. Rewriting the above example so that a Vehicle is composed with Wheel:

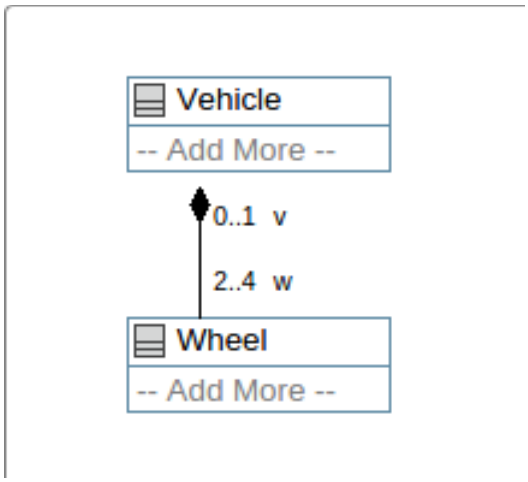
See **Example 2**

With the composition code, notice how the delete code for the Vehicle deletes all the Wheels associated, although the multiplicity is unchanged.

Compositions can be specified inline, or as separate associations (the same way as regular associations). The following two examples are equivalent to the Wheel-Vehicle example specified above.

See **Examples 3 and 4**

The following diagram is a representation of how the composition in the included examples is represented in UML, as generated in an editable class diagram by UmpleOnline.



Syntax

The syntax for compositions is the same as for regular associations, except for the arrow used. While in regular associations, the syntax is **a -- b** for "a associated with b", for compositions it is **a <@>- b** for "a composed with b", or **a -<@> b** for "b composed with a". In the class diagrams generated for compositions, the symbol is a filled-in diamond arrow.

Example 1

```
view plain
01. class Vehicle {}
02.
03. class Wheel {}
04.
05. association {
06.    0..1 Vehicle v -- 2..4 Wheel w;
07. }
08. // @@@skipcppcompile
09.
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 2

```
view plain

01. class Vehicle {}
02.
03. class Wheel {}
04.
05. // vehicle composed with wheels
06. association {
07.    0..1 Vehicle v <@>- 2..4 Wheel w;
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 3

```
view plain

01. class Vehicle {
02.   0..1 v <@>- 2..4 Wheel w;
03. }
04.
05. class Wheel {}
06. // @@@skipcppcompile
07.
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 4

```
view plain

01. class Vehicle {}
02.
03. class Wheel {
04.   2..4 w -<@> 0..1 Vehicle v;
05. }
06. // @@@skipcppcompile
07.
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

association : [=modifier:immutable]? [[associationEnd]] [=arrow:--
|->
|<-
|><
|<@>-
|-<@>] [[associationEnd]] ;

inlineAssociation : [=modifier:immutable]? [[inlineAssociationEnd]] [=arrow:--
|->
|<-
|><
|<@>-
|-<@>] [[associationEnd]] ;

association : association [associationNum];

Association Specializations

A specialization is a subset of duplicate associations where:

- the role names are identical;
- the associated types form a hierarchical structure (i.e. the specialized types are subclasses);
- and the multiplicities are more specific than that of the previously defined association.

For example, consider the case of Vehicle and Wheel classes and an Optional One to Many association between them. A Wheel can only belong to one Vehicle, but the concept of a Vehicle can include any number of Wheels. Say, for instance, we wanted to create a new Bicycle class that extends Vehicle, but we would like to make use of the association that already exists with a stricter multiplicity (say, a Bicycle can have up to 2 Wheels).

Without specializations, we need to create a new association between Bicycle and Wheel and thus generate some unneeded code and duplicate data fields (both Vehicle and Bicycle would have a list of Wheels, which behave more or less the same). With specializations, simply stating that we want a new association between Bicycle and Wheel (as long as the names match up to the Vehicle to Wheel association) is enough to generate more appropriate code.

Example

view plain

```
01. // Here we define our base classes
02. class Vehicle {}
03. class Wheel {}
04.
05. // Here we define our subclasses
06. class Bicycle { isA Vehicle; }
07. class Unicycle { isA Vehicle; }
08.
09. // This is the "parent association" so to speak.
10. // It defines the Vehicle to Wheel relationship
11. // with as much abstraction as possible.
12. association {
13.   0..1 Vehicle -- 0..* Wheel vehicleWheel;
14. }
```

```
15.
16.  // Here, we'd like to extend the functionality of
17.  // the previous association to work with
18.  // subclasses of Vehicle, while utilizing as much
19.  // of the existing code as possible.
20.  // Specializations allow us to do this, since:
21.  // -> Bicycle and Unicycle extend Vehicle;
22.  // -> Wheel is part of the Wheel hierarchy;
23.  // -> The bounds on the left are not less specific
24.  //    than the left bound of the parent
25.  //    association;
26.  // -> The bounds on the right are not less
27.  //    specific than the right bound of the parent
28.  //    association;
29.  // -> Finally, the role names are the same.
30.  association
31.    { 0..1 Bicycle vehicle -- 0..2 Wheel vehicleWheel; }
32.  association
33.    { 0..1 Unicycle vehicle -- 0..1 Wheel vehicleWheel; }
34.
35.
36.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Association Class Definition

An association class defines an object-oriented pattern to manage * -- * (many to many) associations when data needs to be stored about each link of that association.

An association class is a class and can have the items found in an ordinary class (attributes, state machines, etc.).

However it always has two or more 1..many associations to other classes. Each of these is written as:

- A multiplicity (usually *, but 1..* and similar cases are allowed)
- (optional) A role name, i.e. another name for association class in the context of this association.
- The name of the participating class
- (optional) Participating class role name.

The first role name is generally omitted. It is only required if both participating class names are the same (i.e. a reflexive association class). There is an example of this below.

The first example below shows classic use of association classes. A Ticket can be sold to Person in order to attend a Seminar. A Seminar can have many Persons, and a Person can attend many Seminars, so one might imagine simply creating a * -- * association between these classes. However, we want to record the ticket number and price for each Person attending each Seminar. We encode these attributes in the Ticket association class.

For most purposes, an association class behaves in the same manner as though it was an ordinary class with two * -- 1 associations to the participating classes (Person and Seminar in this first example). This is shown in the second example below. However, we want to restrict the possibility of having *more than one* Ticket sold to the same Person for the same Seminar. Association classes add this constraint.

The third example below shows that it is possible to have more than two classes participate in an association class.

The fourth and fifth examples show the use of the role name for the association class itself. The use of such role names is mandatory when the association class on a reflexive many-many association, i.e. the associated classes are the same. In this example there are Members (e.g. people in a club) and some are assigned to be mentors of others. We want to record the date of each assignment, and we do that using an association class. But since both associated classes are the same (Member), we need to use role names on both

ends to ensure all links can be navigated unambiguously. At the Assignment end we have menteeAssignments and mentorAssignments (note the plural) and at the Member end we have roles mentor and mentee.

If we do not use the role names in the 5th example, then error 19 will occur because it will not be possible to refer unambiguously to either side of the association.

Example 1

view plain

```
01. // A person can attend many seminars,
02. // and a seminar can be attended by many people
03. // The relationship between Person and Seminar is
04. // thus many-to-many.
05. //
06. // There is, however, data to record about each
07. // ticket. This can be  recorded as an
08. // association class
09. //
10. // Note the following doesn't currently render
11. // in UmpleOnline using Correct UML
12. // association class notation.
13. // There are plans to fix this.
14. associationClass Ticket
15. {
16.   Integer ticketNumber;
17.   Double price = 0.0;
18. 
19.   * Person attendee;
20.   * Seminar;
21. }
22. 
23. class Person
24. {
25.   name;
26. }
27. 
28. class Seminar
29. {
30.   Date when;
31.   address;
32. }
33. 
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 2

view plain

```
01. // The following shows how we might model sales of
02. // Ticket for Seminars to Persons without using an
03. // association class. Note, however, that in this
04. // model, it would be possible to sell more than
05. // one ticket to a Person for the same Seminar.
06. // Using an association class would hence be
07. // better than this.
08. class Ticket
09. {
10.   Integer ticketNumber;
11.   Double price = 0.0;
12.
13.   * -- 1 Person attendee;
14.   * -- 1 Seminar;
15. }
16.
17. class Person
18. {
19.   name;
20. }
21.
22. class Seminar
23. {
24.   Date when;
25.   address;
26. }
27.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 3

view plain

```
01. // The following shows a 'quaternary' association,
```

```
// where the association class represents data in
// an association that links four classes.
class SportsPlayer {
  name;
}

class Season {
  year;
}

// e.g. goalie, forward etc.
class PlayingPosition {
  description;
}

class Team {
  name;
}

// This gathers the number of points a player
// gained on a particular team in a particular
// season while playing in a particular position
// To get the total points in any one category,
// you would have to add the points several
// instances
associationClass PlayerInPosition {
  Integer points;
  * SportsPlayer player;
  * Season;
  * Team;
  * PlayingPosition position;
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 4

view plain

```
// The following example is a simple class that
// shows an association class using both a
// single role name for one associated class (a1)
```

```
// and two role names to make links to the
// other associated class.
class A{
   f;
}


associationClass B{
   * A a1;
   * b2 C c1;
}


class C {
   d;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 5

view plain

```
// This shows a reflexive association class, where
// the associated classes are the same (Member).
// In this case we must use role names at both
// ends to ensure navigation to the correct sets
// (mentors, mentees)
class Member {
  name;
}

associationClass Assignment {
  Date dateEstablished;
  * menteeAssignments Member mentor;
  * mentorAssignments Member mentee;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
// Associations that would be many-many can also become full-fledged classes too
// See user manual page AssociationClassDefinition
associationClassDefinition : associationClass [name] {
[[associationClassContent]]* }

associationClassContent- : [[comment]]
   | [[reqImplementation]]
   | [[classDefinition]]
   | [[position]]
   | [[displayColor]]
   | [[invariant]]
   | [[softwarePattern]]
   | [[depend]]
   | [[association]]
   | [[inlineAssociation]]
   | [[singleAssociationEnd]]
   | [[attribute]]
   | [[stateMachine]]
   | [[enumerationDefinition]]
   | ;
   | [[extraCode]]

singleAssociationEnd : [[multiplicity]]
  (
  ( [otherEndroleName] [type] [roleName]
  )
    |
  ( [type] [~roleName]?
  )
  ) ;
```

Sorted Associations

It is possible to arrange for associations to maintain themselves in sorted order according to the value of an attribute. Create a derived attribute in order to define a complex sort order or to sort on the value of an attribute of an associated class (the first example below does this in one case).

To declare an association as sorted do the following: after specifying the multiplicity, and any role name, give the keyword 'sorted' followed by the name of an attribute (sort key) in curly brackets. Whenever an add method is called to add an item to the association, Umple ensures the association is maintained in sorted order. .

Note that Umple associations are always 'ordered' by default; i.e. the user controls the position of items. Declaring an association as sorted makes this automatic. The presence of 'sorted' adds a 'sort' function to the API to allow for resorting (if the attributes on which sorting is based are changed, for example) and removes the API for manually ordering the association using addXAt().

Errors are generated if the attribute specified as the sort key attribute does not exist in the class, or the sort key attribute is not of a recognized type.

Java serialization of sorted associations is possible without the need to modify the Java code generated by Umple. The two ends of a sorted association implement the Serializable interface by default. Association items can be serialized and deserialized. After deserialization, more items can be added to the association and the association is maintained in sorted order as demonstrated by the second example.

Example

view plain

```
01. // This example demonstrates two cases of sorted
02. // associations. The main program adds items out
03. // of order. But when they are printed the output
04. // will be sorted
05. class Academy {
06.   1 -- * Course;
07.   1 -- * Student registrants sorted {id};
08. }
09.
10. class Student {
11.   Integer id;
12.   name;
```

```
}

class Course {
  code;
}

class Registration {
  * -- 1 Student;

  // In each course, sort registrations by name
  * sorted {name} -- 1 Course;

  // Derived delegated attribute used for sorting
  // printing
  name = {getStudent().getName()}

  // Derived delegated attribute used for sorting
  // printing
  code = {getCourse().getCode()}


  public String toString() {
    return "Registration: " + getName()
      + ":" + getCode();
  }
}

// Mixin with main program and toString method
class Academy {
  public static void main(String [ ] args) {
    Academy ac = new Academy();
    Student j = ac.addRegistrant(12,"Jim");
    Student a = ac.addRegistrant(4,"Ali");
    Student m = ac.addRegistrant(8,"Mary");
    Student f = ac.addRegistrant(3,"Francois");
    Course c = ac.addCourse("CS191");
    Course c2 = ac.addCourse("AN234");
    j.addRegistration(c);
    a.addRegistration(c);
    m.addRegistration(c);
    f.addRegistration(c);
    m.addRegistration(c2);
    f.addRegistration(c2);
    System.out.println(ac);
  }
```

```
58.   public String toString() {
59.     String result="Students:\n";
60.     for (Student s: getRegistrants()) {
61.       result +=s + "\n";
62.     }
63.     result +="Courses:\n";
64.     for (Course c: getCourses()) {
65.       result +=c + "\n";
66.     }
67.
68.     return result;
69.   }
70. }
71.
72. class Student {
73.   public String toString() {
74.     String result="Student=" + id + "["
75.       + name + "\n";
76.     for (Registration r: getRegistrations()) {
77.       result +="  In: " + r + "\n";
78.     }
79.     return result;
80.   }
81. }
82.
83. class Course {
84.   public String toString() {
85.     String result ="Course=" + code + "\n";
86.     for (Registration r: getRegistrations()) {
87.       result +="  Has: " + r + "\n";
88.     }
89.     return result;
90.   }
91. }
92.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example of sorted associations serialisation in Java

```
view plain
01. // This example demonstrates serialisation of
```

```
// sorted associations in Java. The main program
// adds items to an object out of order, serialize the
// object, deserialize it and then add more items
// to it. When they are printed after deserialisation,
// all items will appear in the right order.

class Academy {
  1 -- * Student registrants sorted {id};
}

class Student {
  Integer id;
  name;
}

// Mixin with main program, toString method
// and serialization test

class Academy {
  depend java.nio.file.*;
  depend java.io.*;

  public static void main(String [ ] args) {
    String filename = "SortedSerializableAssociation.ser";
    Academy ac = new Academy();
    Student j = ac.addRegistrant(12,"Jim");
    Student a = ac.addRegistrant(4,"Ali");

    System.out.println("Before Serialization");
    System.out.println(ac);

    //serialization
    try
    {
      FileOutputStream file = new FileOutputStream(filename);
      ObjectOutputStream out = new ObjectOutputStream(file);
      out.writeObject(ac);
      out.close();
      file.close();
    }
    catch(Exception ex)
    {
      System.out.println(ex.getMessage());
    }

```

```
47.     Academy ac2 = null;
48.
49.     // Deserialization
50.     try
51.     {
52.       FileInputStream file = new FileInputStream(filename);
53.       ObjectInputStream in = new ObjectInputStream(file);
54.       ac2 = (Academy)in.readObject();
55.       in.close();
56.       file.close();
57.     }
58.     catch(Exception ex)
59.     {
60.       System.out.println(ex.getMessage());
61.     }
62.
63.     // Adding new elements and comparing
64.     try
65.     {
66.     Student m = ac2.addRegistrant(8,"Mary");
67.     Student f = ac2.addRegistrant(3,"Francois");
68.     }
69.     catch(Exception ex)
70.     {
71.       System.out.println(ex.getMessage());
72.     }
73.
74.     System.out.println("After Serialization");
75.     System.out.println(ac2);
76.
77.   }
78.
79.   public String toString() {
80.     String result="Students:\n";
81.     for (Student s: getRegistrants()) {
82.       result +=s + "\n";
83.     }
84.     return result;
85.   }
86. }
87.
88. class Student {
89.   public String toString() {
90.     String result="Student=" + id + "["
91.       + name +"]"+ "\n";
```

```
92.     return result;
93.   }
94. }
95.
96.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

inlineAssociationEnd : [[multiplicity]] [~roleName]? [[isSorted]]?

isSorted- : sorted { [priority] }

Simple Constraints

Umple supports basic OCL-type Boolean constraints that limit what umple-generated mutator methods (set, add, etc.) and constructors can do. If a constraint is violated set methods will return false, and constructors will throw an exception.

Constraints are specified within square brackets. Simple constraints can refer to attributes and literals (numbers, strings). These can be compared using standard comparison operators such as < and >. Expressions can be conjoined by the and operator && or the or operator ||. An exclamation mark is the not operator. Parentheses can be used to adjust the standard order of operations.

Guards on state machines are also Boolean expressions and, like simple constraints, are also enclosed within square brackets.

Additional capabilities are being developed in Umple to allow other types of constraints.

Example with simple constraints

view plain

```
01. // The following demonstrates two simple
02. // constraints limiting the range of age
03. // The test code demonstrates that this works
04. // as expected.
05. class Client {
06.   const Integer MinAge =18;
07.   Integer age;
08.   [age >= MinAge]
09.   [age <= 120]
10.
11.   public static void main(String [ ] args) Java {
12.     Client c = new Client(40);
13.     for (int i: new int[]
14.       {-1,10,17,18,19,50,119,120,122,1000})
15.     {
16.       System.out.println(
17.         "Trying to initialize age to "+i);
18.       System.out.println(c.setAge(i)
19.         ? "  Success"
20.         : "  FAILURE");
21.     }
22.   }
```

```
23.
24.   public static void main(String [ ] args) Python {
25.     import Client
26.     c = Client.Client(40)
27.     for i in [-1, 10, 17, 18, 19, 50, 119, 120, 122, 1000]:
28.       print("Trying to initialize age to", i)
29.       print("  Success" if c.setAge(i) else "  FAILURE")
30.   }
31. }
32.
33.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example with more operators

view plain

```
01. // Example with a few more operators
02. class X {
03.   Integer i;
04.   Integer j;
05.   [i > j]
06.   [i < 5 && j > 0]
07.   [! (i == 10)]
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

// A constraint is an expression optionally be surrounded in round brackets or negated

constraint- : [[constraintBody]] [[linkingOp]]? | [[constraintExpr]] [[linkingOp]]?

//negativeConstraint : (! | not) ([[constraintName]] | [[constraint]]) fixed issue1536 'Umple does not parse properly the guards'

```
negativeConstraint :
  ( ! | [!constraintNegSymbol:not\s+]
  )
  ( [[constraintName]] | [[constraint]]
  )

// A constraint body is a constraint expression (possibly with a linking operator such as && or ||).
constraintBody : ( [[constraint]] )

linkingOp- :
  ( [=andOp:and|&&|&]
    | [[equalityOperator]]
    | [!orOp:(or|\Q||\E)]
  ) [[linkingOpBody]]

constraintExpr- : [[statemachineExpr]]
    | [[stringExpr]]
    | [[boolExpr]]
    | [[numExpr]]
    | [[associationExpr]]
    | [[genExpr]]
    | [[loneBoolean]]
    | { [!fakeContraint:[^\n]+] }

//must be string
stringExpr : [[stringExprOperator]] | [[stringExprPlain]]

//basically the "other" catagory, contains everything that can be equal to something else
genExpr : [[constraintVariable]] [[equalityOperator]] [[constraintVariable]]

//for floats, doubles and ints
numExpr : [[numberExpression1]]
    | [[numberExpression2]]
    | [[numberExpression3]]
```

```
   | [[numberExpression4]]

ordinalOp- : [=greaterOp:greater|>=|=>|=>=]
   | [=lessOp:less|<=|=<|=<=]
   | [=moreOp:larger|>]
   | [=smallerOp:smaller|<]
```

Preconditions

You can specify basic preconditions in methods as shown below. Like other constraints, they are specified in square brackets. Precondition appears as the first lines of a method and must start with 'pre:'. A precondition can refer to one or more attributes or method arguments. If a precondition is not satisfied when the method is run an exception will be thrown; in Java a RuntimeException is used.

Example

view plain

```
01. class PotentialVoter {
02.   Integer age;
03.
04.   int vote(int candidate) {
05.     [pre: age >= 18]
06.     [pre: candidate > 0]
07.     // rest of stuff that we do not interpret
08.      return 0;
09.   }
10. }
11.
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
// Constraints in Umple.
// This is currently under development. Constraint capability is being
// developed in agile increments. The first step, described below,
// allows limiting the values of attributes. Code generation is not currently enabled.
// Constraints may appear in classes (including association classes)
// as well as in states.
precondition : [ [name]? pre : [[constraint]] ]

//queuedMethodDeclaration- : queued [[methodDeclarator]]
[[methodThrowsExceptions]]? [[methodBody]]
```

```
methodBody- :
 ( [[codeLangs]] {
 ( [[precondition]]
  | [[postcondition]]
  | [[assertion]]|[[testCase]]
 )* [**code] }
 )+
```

Model Constraints

Constraints can be used to assert that certain properties of the model are as expected, as shown in the examples below. Associations, attributes and generalizations can be checked.

Why would model constraints be useful, given that a programmer can just inspect the code to see if there are statements declaring the facts in question? The answer is that Umple has sophisticated separation of concerns mechanisms including traits, mixins and mixsets, and the developer may not be completely sure that a given model constraint will always be true. An attribute, for example, might be only introduced into the class in a completely different file or mixset, and the developer may not be certain that the file or mixset is being included. Using a model constraint allows a developer who wants to use that attribute in a method to cause an Umple compilation error if it does not in fact exist, rather than waiting to see if Java code fails to compile.

Below is an example of a model constraint, but there are other examples in the manual pages that give help about the error messages raised by these constraints:

- Assertion that an attribute of a certain name (defined elsewhere) must exist in this class
- Assertion that an attribute of a certain type must exist
- Assertion that this class is a subclass of another (with the isA statement defined elsewhere)
- Assertion that this class is a superclass of another class (with the isA statement defined in that class)
- Assertion that an association exists between two classes

Example

view plain

```
01. class X {
02.   // Checks that this class has an association
03.   // with class Z.
04.   [model: -- Z]
05. }
06.
07. class Y {
08.   isA X;
09.   // Verifies that this is a subclass of X
10.   [model: isA X]
11. }
```

```
12.
13.  class Z {
14.   a;
15.   1 -- * X;
16.   // Verifies that there is an association a
17.   [model: has a]
18.  }
19.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

modelConstraint- : [model : [[modelConstraintBody]]]

modelConstraintBody : ([[modelConstraintBody]]) [[modelLinkingOp]]?
  | [[modelExpr]] [[modelLinkingOp]]?

modelLinkingOp :
 ([=and:&|&&|and|,] | [!or:
 ([|][|]?|or|;
)]
) [[modelConstraintBody]]

modelExpr : [~source]? [[modelRelationOp]] [~target]

modelRelationOp- : [[modelRelationOpAssociation]]
  | [[modelRelationOpAttribute]]
  | [[modelRelationOpInheritance]]

modelRelationOpInheritance : [=subclass:isA]
  | [=subclass:subclass|child|>>]
 (of
)?
  | [=subclass:inherits]
 (from

```
 )?
  | [=superclass:parent|superclass|<<]
 ( of
 )?

modelRelationOpAttribute : [=op:has]
 ( attribute | attr
 )? [=classification:named|of]?

modelRelationOpAssociation : [[modelRelationAssociationEnd]]? [=op:--
   |->
   |<-
   |-<@>
   |<@>-] [[modelRelationAssociationEnd]]?

modelRelationAssociationEnd : [!bound:(\d+|[**])] ( .. [!bound:(\d+|[**])] )?
```

Singleton Pattern

Use the Singleton keyword to mark a class as a singleton class. The code ensures that only one object of the class is instantiated at runtime.

For more details on the Singleton pattern, see this Wikipedia page.

Example

```
view plain

class Airline
{
  singleton;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

softwarePattern- : [[isA]] | [[singleton]] | [[immutable]] | [[keyDefinition]] | [[codeInjection]]

// A class that can have only one instance SingletonPattern
singleton- : [=singleton] ;

Immutable Pattern

Mark a class as immutable to force all its contents to be immutable. The code ensures all attributes and associations can only be set in the constructor, and cannot be modified again subsequently.

The only associations allowed to be immutable are directed associations to other immutable classes.

A class can only be immutable if all its superclasses are also immutable. Declaring a superclass immutable forces its subclasses to be immutable; in other words, immutability is inherited. If the subclasses break immutability constraints (such as the type of attributes allowed), then errors will be raised.

Individual attributes and associations can be marked as immutable instead of the entire class. An attribute can be marked as lazy immutable if it needs to be initialized after the constructor has finished. An immutable class cannot have an association to a non-immutable class.

For more details on the Immutable pattern, see this Wikipedia page.

Immutable class

view plain

```
01. // A simple example of an immutable class
02. class Point2D
03. {
04.   immutable;
05.   Float x;
06.   Float y;
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Immutable class with reference to another class

view plain

```
01. // An example of one immutable class making reference to another
```

```
02.
03.  class Path
04.  {
05.    immutable;
06.    1 -> * Point2D pathElements;
07.  }
08.
09.  class Point2D
10.  {
11.    immutable;
12.    Float x;
13.    Float y;
14.  }
15.
16.  strictness allow 36; // We are OK with this warning
17.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Immutable association

view plain

```
01.  // Example of the declaration of an association to be immutable
02.  // Note that this can be set only the first time
03.  class X {
04.    Integer id;
05.    immutable * -> 0..1 Y;
06.  }
07.
08.  class Y {
09.    immutable;
10.    someInfo;
11.  }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
softwarePattern- : [[isA]] | [[singleton]] | [[immutable]] | [[keyDefinition]] |
[[codeInjection]]

// A class that can't be modified when created ImmutablePattern
immutable- : [=immutable] ;

association : [=modifier:immutable]? [[associationEnd]] [=arrow:--
   |->
   |<-
   |><
   |<@>-
   |-<@>] [[associationEnd]] ;

inlineAssociation : [=modifier:immutable]? [[inlineAssociationEnd]] [=arrow:--
   |->
   |<-
   |><
   |<@>-
   |-<@>] [[associationEnd]] ;

derivedAttribute- : [=modifier:immutable
   |settable
   |internal
   |defaulted
   |const
   |fixml]? [[typedName]] =
  ( [[moreCode]]
  )+

complexAttribute- : [=unique]? [=lazy]? [=ivar]? [=modifier:immutable
   |settable
   |internal
   |defaulted
   |const
   |fixml]? [[typedName]]
  (= [**value]
```

```
)? ;

association : association [associationNum];
```

Delegation Pattern

Delegation is one of the most fundamental patterns, that underpins many more sophisticated patterns found in the Gang of Four book. To simplify code, reduce coupling, and adhere to the 'Law of Demeter', it encourages you to create one-line methods that do nothing other than call methods in neighbouring classes. Other code in your class then calls the delegation methods, rather than also calling the methods in the neighbouring classes. That way, very few methods (just the delegation methods) actually need to traverse associations

In Umple there is no special syntax for delegation, however the notation for derived attributes has the effect of creating delegation methods in the manner shown in the example below.

Delegation methods can also be used to help build occurrences of other patterns such as

- Adapter: You create a delegation method that calls a method with a different name in a class you are trying to adapt for use in your system.
- Facade: You create a class that does a lot of its work by calling other classes in the system.

Note that delegation methods as shown here are 'read only'. They generate a get method with the name of the derived attribute, not a set method.

Example

view plain

```
01. // This system has many delegation methods so that information in one
02. // be obtained in a neighbouring class easily. The delegation is created
03. // derived attributes, which generate get methods for use by other code
04. class Airline {
05.   code;
06.   name;
07.   1 -- * RegularFlight;
08. }
09.
10. // A regular flight flies every day and has a flight number and other fixe
11. class RegularFlight {
12.   Integer number;
13.   Time depTime;
14.   * outgoing -- 1 Airport origin;
```

```
15.   * incoming-- 1 Airport destination;
16.   fullNumber = {getAirline().getCode()+getNumber()}
17.  }
18.
19.  // A specific flight actually has passengers
20.  class SpecificFlight {
21.   * -- 1 RegularFlight;
22.   Date depDate;
23.
24.   // Four delegation attributes to save multiple methods in this
25.   // class from having to traverse the association
26.   flightNumber = {getRegularFlight().getFullNumber()}
27.   Airport origin = {getRegularFlight().getOrigin()}
28.   Airport destination = {getRegularFlight().getDestination()}
29.   Time plannedDepTime = {getRegularFlight().getDepTime()}
30.  }
31.
32.  // A passenger on a flight
33.  class Booking {
34.   lazy seat;
35.   * -- 1 SpecificFlight;
36.   * -- 1 Passenger;
37.
38.   // Second-level delegation - data comes from RegularFlight
39.   flightNumber = {getSpecificFlight().getFlightNumber()}
40.   Airport destination = {getSpecificFlight().getDestination()}
41.
42.   // Simple delegation
43.   passengerName = {getPassenger().getName()}
44.   Time plannedDepTime = {getSpecificFlight().getPlannedDepTime()}
45.
46.   //Simple method using the data gathered by delegation
47.   public String toString() {
48.    return getFlightNumber()+" "+getPlannedDepTime()+" "+
49.     getDestination().getName()+" "+getPassengerName()+" "+getSeat
50.   }
51.  }
52.
53.  class Passenger {
54.   lazy Integer freqFlyerNumber;
55.   name;
56.  }
57.
58.  class Airport {
59.   code;
```

```
60.    name;
61.   }
62.
63.
64.  // Mixin with main program to test the above
65.  class Airline {
66.    public static void main(String [ ] args)  {
67.      Airport ott = new Airport("YOW","Ottawa");
68.      Airport tor = new Airport("YYZ","Toronto Pearson");
69.      Airline a = new Airline("AO","Air Ottawa");
70.      RegularFlight r = a.addRegularFlight(100,java.sql.Time.valueOf("13:
71.      SpecificFlight f = r.addSpecificFlight(java.sql.Date.valueOf("2017-
     03-15"));
72.      Passenger p = new Passenger("John");
73.      Booking b = new Booking(f,p);
74.      System.out.println(b);
75.    }
76.  }
77.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Keys for Equality and Hashing

In any class you can specify a set of attributes and associations as keys. The associations must have a multiplicity of 1 at the other end. This allows Umple to generate code that defines what objects of the class are equal (they have the same values for the key).

Umple will also generate a hashCode method in any class that has a defined key. This helps when looking up an object in a set.

Place the comma-separated list of key elements in curly brackets, after the word key. If you have multiple key statements, a warning is generated.

Example

view plain

```
01. class Sport {
02.   name;
03.   description;
04.   code;
05.   key { code }
06. }
07.
08. class League {
09.   name;
10.   id;
11.   geographicalArea;
12.   * -- 1 Sport;
13.   Date seasonStart;
14.   Date seasonEnd;
15.   key { id }
16. }
17.
18. class Team {
19.   name;
20.   * -- 1 League;
21. }
22.
23. class Player {
24.   name;
25.   Integer id;
26.   key { id }
27.  }
28.
```

```
29. class PlayerOnTeam {
30.   Integer year;
31.   * -- 1 Player;
32.   * -- 1 Team;
33.   key { year, player, team }
34. }
35.
36.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

softwarePattern- : [[isA]] | [[singleton]] | [[immutable]] | [[keyDefinition]] | [[codeInjection]]

// For equality and hashing KeysforEqualityandHashing
keyDefinition- : [[defaultKey]] | [[key]]

// Keys are used to define quality and hash codes, plus Sql keys.
// See user manual page KeysforEqualityandHashing
defaultKey : key { }

key : key { [keyId] (, [keyId])* }

Abstraction Occurrence

When modeling, it is common to encounter situations where there will be objects (abstractions) that need to store information that will not vary among many other objects (occurrences). We want to avoid duplicating the data in each occurrence.

Examples include:

- Books (abstraction) have a title and an author; but there are many copies (occurrence) in the library that are separately borrowed by clients. We do not want to duplicate the title and author in each copy object.
- TV series (abstraction) have stars, a name and a studio; but there are many episodes (occurrences).

It is an *antipattern* (i.e. poor design) to use the same class for both abstraction and occurrence. It is also an antipattern to use generalization (i.e. erroneously making the occurence a subclass of the abstraction) because that would result in duplication of data in the attributes of the occurrences. This is often not well-understood by inexperienced modelers.

The solution is to create a one-to-many association between the abstraction class and the association class. Umple provides a built-in pattern to make abstraction-occurrence relationships explicit, although using an ordinary association would be a valid alternative.

To use this pattern, add a use statement to incorporate the built-in AbstractionOccurrence.ump file. Then use isA statements to declare which class is an Abstraction, and which is an occurrence (of which Abstraction). Follow what is show in the second block of code below.

This is one of several patterns in Umple Umple that are built into the language as of 1.32 via a use statement. See the code below.

The Abstraction-Occurrence pattern code that is imported by specifying use lib:AbstractionOccurrence.ump

view plain

```
01. // Generic pattern for abstractions and occurrences
02. // In any Umple program say 'use lib:AbstractionOccurrence.ump;' to ir
03.
04. // Something that has many occurrences
05. // Examples include a TV show that has episodes, where the show has a
06. // Or a route that has runs, or a published book in a library where there
07. trait Abstraction {}
```

```
08.
09.  // Something representing occurrences of an abstraction
10.  // where both the abstraction and the occurrences need their own insta
11.  // It is common for modellers to make mistakes and just have one class
12.  // Or to consider the Occurrences as instances of the abstractions, but t
13.  trait Occurrence <Abstraction> {
14.   * -- 1 Abstraction;
15.  }
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example of the Abstraction-Occurrence pattern to specify a library

```
view plain
01.  /* Example of using builtin Umple
02.   * Abstraction Occurrence pattern file
03.  */
04.
05.  use lib:AbstractionOccurrence.ump;
06.
07.  class Author {
08.   name;
09.  }
10.
11.  class Book {
12.    isA Abstraction;
13.    title;
14.    * -- * Author;
15.  }
16.
17.  class CopyOfBook {
18.    isA Occurrence <Abstraction = Book>;
19.    Integer barcode;
20.  }
21.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Player Role

When modeling, it is common to encounter situations where one wants to make multiple subclasses, but can not or should not for various reasons. One reason not to create a subclass is because instances may need to change class as their roles in the system change. Another reason not to create a subclass is because an instance may need to play two or more roles smultaneously.

The solution is to create a Role class and to link it by a many-to-one association to the main class (which we will call the Player class).

To use this pattern, add a use statement to incorporate the built-in PlayerRole.ump file. Then use isA statements to declare which class is a Player, and which is an Role (of which Player). Follow what is show in the second block of code below.

This is one of several patterns in Umple Umple that are built into the language as of 1.32 via a use statement. See the code below.

The Player Role pattern code that is imported by specifying use lib:PlayerRole.ump

view plain

```
01. // Generic pattern for players and roles
02. // In any Umple program say 'use lib:PlayerRole.ump;' to include this p
03.
04. // A player is an object that can have many roles
05. // A person can be an employee and a student
06. trait Player {}
07.
08. // A captures some functionality that a player may have
09. // either temporarily or in conjunction with other roles
10. trait Role <Player> {
11.  * -- 1 Player;
12. }
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example of the Player-Role pattern to specify a system for a child sports team

view plain

```
01. /* Example of using builtin Umple
02.  * Player Role pattern
03. */
04.
05. use lib:PlayerRole.ump;
06.
07. class Person {
08.  isA Player;
09.  name;
10. }
11.
12. class TeamRole {
13.   isA Role<Player=Person>;
14. }
15.
16. class Athlete {
17.   isA TeamRole;
18. }
19.
20. class Coach {
21.   isA TeamRole;
22. }
23.
24. class ParentOfAthlete
25. {
26.   isA TeamRole;
27.   * -- * Athlete;
28. }
29.
30.
31. class SportsTeam {
32.   nickname;
33.   * -- * TeamRole;
34. }
35.
36.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Transportation Pattern A

Transportation systems for airlines, bus networks and train networks have many things in common. This is a high level pattern to capture one way of representing common features of those models.

The pattern consists of traits representing

- Generic Routes with an ID
- RegularlyScheduledRun on those routes
- ActualRun (occurrences) of the above
- ScheduledStop
- Vehicles used

This is only one possible design for a transportation system, but it illustrates some possibilities. The two examples show how the design has been applied to create an airline system and a bus system.

Experiment with the diagrams in UmpleOnline. You can use control-G and Control-E to switch layout; you can use control-R to toggle between viewing traits or collapsed classes.

Transportation pattern code that is imported by specifying use lib:TransportationPatternA.ump

view plain

```
01. // This is a library of traits that can be used
02. // to build the core of transportation systems
03.
04. use lib:AbstractionOccurrence.ump;
05.
06. // A route such as a flight or bus route
07. trait Route <Stop> {
08.  routeID; // generic ID for all runs on this route, e.g. bus route nu
09.  * -- * Stop;
10. }
11.
12. // A run pattern at a particular time of day that
13. // is repeated on multiple days
14. trait RegularlyScheduledRun <Route, ScheduledStop> {
15.  isA Abstraction;
16.  runId; // e.g. a flight number / Bus systems tend to hide this fro
```

```
17.   String plannedDaysOfWeek; // e.g. "135" for Mon, Wed, Fri
18.   * -- 1 Route;
19.   Boolean forwardDirection; // only relevent if routeID is same in b
20.   1 --
      * ScheduledStop; // contains the times of the stops in order; not
21.  }
22.
23.  // A stop on a particular regular run
24.  trait ScheduledStop <Stop> {
25.   * -- 1 Stop;
26.   Time expectedArrivalTime; // null for departure only
27.   Time expectedDepartureTime; // May be same as arrival time for
28.  }
29.
30.  trait Stop {
31.    stopName; // e.g. Airport, bus station etc.
32.  }
33.
34.  // People can book on this for flights. It can be late or cancelled
35.  trait ActualRun <RegularlyScheduledRun, Vehicle>  {
36.    isA Occurrence <Abstraction = RegularlyScheduledRun>;
37.    Date runDate;
38.    Boolean isCancelled;
39.    * -- 1 Vehicle vehicleUsed;
40.  }
41.
42.  // Specific plane, bus, etc.
43.  trait Vehicle {
44.    vehicleID;
45.  }
46.
47.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example of use of the above transportation pattern for an airline

view plain

```
01.  /* Example of using builtin Umple pattern files
02.  */
03.
04.  use lib:AbstractionOccurrence.ump;
```

```
use lib:TransportationPatternA.ump

// This shows how a general pattern for
// transportation can be adapted to
// describe an airline


class AirlineRoute {
  isA Route <Stop = Airport>;
}

class RegularFlight {
  isA RegularlyScheduledRun <
    Route = AirlineRoute,
    ScheduledStop = AirportStop
   >;
}

class AirportStop {
  isA ScheduledStop <Stop = Airport>;
}

class Airport {
  isA Stop;
}

class SpecificFlight {
  isA ActualRun <
    RegularlyScheduledRun = RegularFlight,
    Vehicle = Airplane
  >;
}

class Airplane {
  isA Vehicle;
}


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example of use of the above transportation pattern for a bus system

view plain

```
01. /* Example of using builtin Umple pattern files
02. to create a bus system
03. */
04.
05. use lib:AbstractionOccurrence.ump;
06. use lib:TransportationPatternA.ump
07.
08. // This shows how a general pattern for
09. // transportation can be adapted to
10. // describe a bus system
11.
12.
13. class BusRoute {
14.   Integer routeNumber;
15.   isA Route <Stop = BusStop>;
16. }
17.
18. class RegularlyTimedBusRun {
19.   isA RegularlyScheduledRun <
20.     Route = BusRoute,
21.     ScheduledStop = TimedStopAtBusStop
22.    >;
23. }
24.
25. class TimedStopAtBusStop {
26.   isA ScheduledStop <Stop = BusStop>;
27. }
28.
29. class BusStop {
30.   isA Stop;
31. }
32.
33. class BusRun {
34.   isA ActualRun <
35.     RegularlyScheduledRun = RegularlyTimedBusRun,
36.     Vehicle = Bus
37.   >;
38. }
39.
40. class Bus {
41.   isA Vehicle;
42. }
43.
```

44.

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Basic State Machines

A state machine has a fixed set of of values (called states) The state machine transitions from state to state by the occurrence of events. State machines are very useful for quickly defining a program's behaviour.

In Umple, a state machine is modeled as a special type of attribute. In any class, simply specify the state machine name, and follow the name by a block starting with '{' and ending with '}'. This indicates to Umple that you are defining a state machine, and not an ordinary attribute.

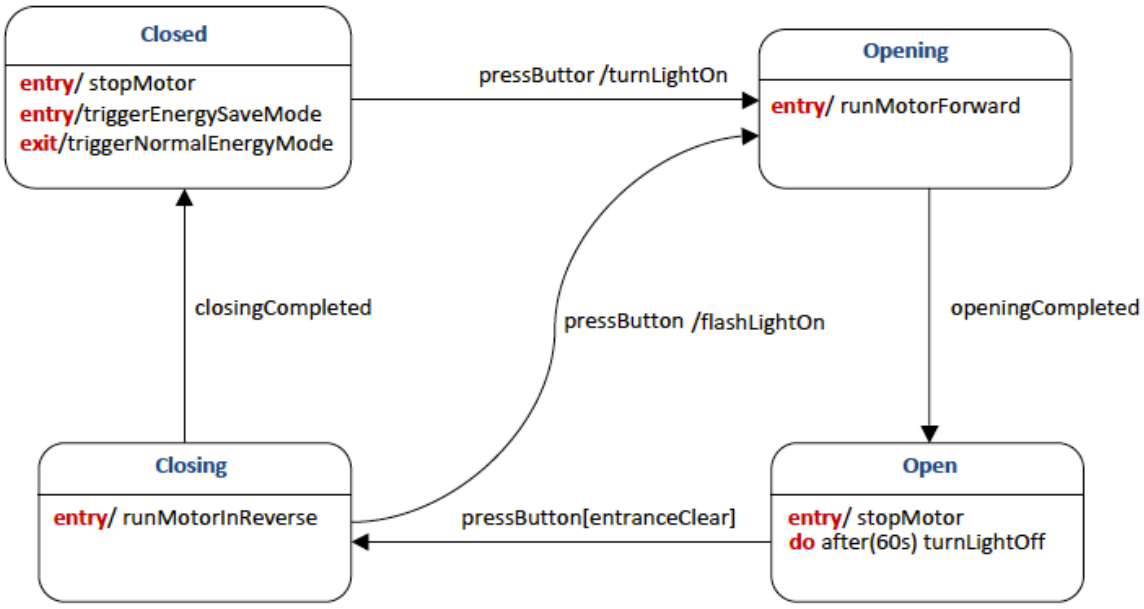
Within the block, you list the names of the various states. Each state is followed by a block, again indicated using '{' to start and '} to end. This block defines the details of a state.

Details of each state can include:

- **Transitions**. A transition has an **event** name, the symbol '->', and then the name of a destination state. The event name will become the name of a generated method. To trigger the transition, you simply call the method. When the event occurs while the state machine is in the origin state, the state machine will change to the destination state.
  Transitions can also have:
  - **transition actions**, specified as '/' followed by a block of code to execute.
  - **guards**, which are boolean expressions in square brackets. Even if the event occurs, the transition only takes place if the guard evaluates to true.
- Entry actions, exit actions and do activities. See the separate manual page.
- Nested states. See the separate manual page. You can nest state machines to arbitrary levels of depth.
- Final states. See the separate manual page.

Note that, in addition, you can specify code to be executed whenever an event is processed by using before or after directives.

The following diagram shows a garage door state machine as a UML diagram. The Umple code for this is in the second example shown, and is further explained in the video below.



Basic Garage Door Example

view plain

```
01. // This example shows a simple state machine
02. // without any actions or guards
03. //
04. // In the following, status is a state machine,
05. // and acts like an attribute, whose value is set
06. // by various events.
07. //
08. // Open, Closing, Closed, Opening and HalfOpen are
09. // the possible values, or states, of status.
10. //
11. // buttonOrObstacle, reachBottom and reachTop are
12. // events. These become generated methods that can
13. // be called to cause a state change.
14. //
15. // To make the state diagram appear in
16. // UmpleOnline, select 'Options' then choose
17. // 'GraphViz State Diagram'
18. class GarageDoor
19. {
20.   status {
21.     Open { buttonOrObstacle -> Closing;  }
22.     Closing {
23.         buttonOrObstacle -> Opening;
24.         reachBottom -> Closed;
```

```
      }
      Closed { buttonOrObstacle -> Opening; }
      Opening {
         buttonOrObstacle -> HalfOpen;
         reachTop -> Open;
      }
      HalfOpen { buttonOrObstacle -> Opening; }
  }
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Garage Door Example with Actions

view plain

```
// This is a more fully-featured state machine for
// a garage door corresponding to the diagram
// above
class Garage {
  Boolean entranceClear=true;
  GarageDoor {
    Closed {
      entry/{stopMotor();}
      entry/{triggerEnergySaveMode();}
      exit/ {triggerNormalEnergyMode();}
      pressButton -> /{turnLightOn();} Opening;
    }
    Opening {
      entry/{runMotorForward();}
      openingCompleted -> Open;
    }
    Open {
      entry/{stopMotor();}
      // do {wait(60000); turnLightOff();}
      pressButton [getEntranceClear()] -> Closing;
    }
    Closing {
      entry/{runMotorInReverse();}
      closingCompleted -> Closed;
      pressButton -> /{flashLightOn();} Opening;
    }
```

```
27.   }
28.
29.   boolean runMotorInReverse() Java {
30.     System.out.println(
31.       "Running motor in reverse");
32.    return true;
33.   }
34.
35.   boolean runMotorInReverse() Python {
36.    print("Running motor in reverse")
37.    return True
38.   }
39.
40.   boolean flashLightOn() Java {
41.     System.out.println("Flashing light on");
42.    return true;
43.   }
44.
45.   boolean flashLightOn() Python {
46.    print("Flashing light on")
47.    return True;
48.   }
49.
50.   boolean turnLightOn() Java {
51.     System.out.println("Turning light on");
52.    return true;
53.   }
54.
55.   boolean turnLightOn() Python {
56.    print("Turning light on")
57.    return True
58.   }
59.
60.    boolean turnLightOff() Java {
61.     System.out.println(
62.       "Turning light off");
63.    return true;
64.   }
65.
66.    boolean turnLightOff() Python {
67.    print("Turning light off")
68.    return True
69.   }
70.
71.   boolean runMotorForward() Java {
```

```
72.      System.out.println(
73.        "Running motor forwards");
74.      return true;
75.    }
76.
77.    boolean runMotorForward() Python {
78.      print("Running motor forwards")
79.      return True
80.    }
81.
82.    boolean triggerEnergySaveMode() Java {
83.       System.out.println(
84.         "Triggering Energy Saving Mode");
85.      return true;
86.    }
87.
88.    boolean triggerEnergySaveMode() Python {
89.      print("Triggering Energy Saving Mode")
90.      return True
91.    }
92.
93.    boolean stopMotor() Java {
94.       System.out.println("Stopping motor");
95.      return true;
96.    }
97.
98.    boolean stopMotor() Python {
99.      print("Stopping motor")
100.     return True
101.   }
102.
103.   boolean triggerNormalEnergyMode() Java {
104.      System.out.println(
105.        "Triggering Normal Energy Mode");
106.         return true;
107.   }
108.
109.   boolean triggerNormalEnergyMode() Python {
110.     print("Triggering Normal Energy Mode")
111.     return True
112.   }
113.
114.   boolean waitawhile() Java {
115.     System.out.println("Waiting");
116.     return true;
```

```
117.    }
118.
119.    boolean waitawhile() Python {
120.      print("Waiting")
121.      return True
122.    }
123.
124.    boolean test() Java {
125.           System.out.println("Testing");
126.          return true;
127.    }
128.
129.    boolean test() Python {
130.      print("Testing")
131.      return True
132.    }
133.  }
134.  // @@@skipcppcompile
135.
136.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

YouTube Video with Additional Explanation

Timothy Lethbridge





Watch on

Syntax

```
stateMachine : [[enum]]
    | [[inlineStateMachine]]
    | [[referencedStateMachine]]
    | [[activeDefinition]]

//Issue 148
inlineStateMachine : [=queued]? [=pooled]? [~name] {
  ( [[comment]]
    | [[reqImplementation]]
    | [[state]]
    | [[trace]]
    | [[mixsetDefinition]]
    | [=||]
    | [[standAloneTransition]]
  )* }

// An enum is a state machine that has no events
// stateName is prefixed with ~ to match alphanumeric names only.
// This is needed to solve issue 399, which is cause when a terminating }
is parsed as part of the statename.
```

```
enum : [~name:key] { [~stateName]? (, [~stateName])* }

state : [=final]? [stateName] { [[stateInternal]]* }

stateEntity- : [=||]
    | [[mixsetDefinition]]
    | [[entryOrExitAction]]
    | [[autoTransition]]
    | [[transition]]
    | [[activity]]
    | [[state]]
    | [[displayColor]]
    | [[trace]]
    | ;

autoTransition : [[activity]]? [[autoTransitionBlock]]

// Autotransitions have no event. The transition is immediately taken
// or taken after the do activity ends[[guard]]? -> [[action]]?
// The action can come before or after the arrow
autoTransitionBlock- : [[guard]]?
  ( [[action]] -> | -> [[action]] | ->
  ) [stateName] ;

// A transition guard can come before or after the arrow
// The order of guard and event definition can also be interchanged
transition :
  ( [[eventDefinition]] [[guard]]
    | [[guard]] [[eventDefinition]]
    | [=unspecified]? [[guard]]
    | [[eventDefinition]]
  )?
  ( [[action]] ->
    | -> [[action]]
    | ->
  ) [stateName] ;

eventDefinition- : [[afterEveryEvent]] | [[afterEvent]] | [~event]
  ( [[parameterList]]
  )?

// The timer in a timed event can be an number, variable, function() or
function(num)
afterEveryEvent- : afterEvery ( [!timer:[+*/a-zA-
Z0-9_\.-]+(\([0-9\.]*\))?] )
```

```
afterEvent- : after ( [!timer:[+*/a-zA-Z0-9_\.-]+(\([0-9\.]*\))?] )

// An action can be executed on a transition, or on entry or exit
action : / [[moreCode]]+

entryOrExitAction : [=type:entry|exit] [[guardCode]]? / [[moreCode]]+

// A do activity is long-lasting and can be interrupted
activity : do [[moreCode]]+

guard : [ [[constraint]] ]
```

State Machine Actions and Do Activities

Transition actions: When a transition is taken, an action can occur. This is indicated using a slash "/" followed by arbitrary code in braces. The arbitrary code may be placed after the event name or before the destination state name. In other words the arrow -> can be placed either directly after the event (before the action), or directly before the destination state (after the action)

Entry and exit actions: Similarly, when entering or exiting a state, an action can occur. This is indicated using the keywords entry or exit, followed by a slash, followed by code.

The actions described above should be programmed such that they take negligible time to execute; in other words they shouldn't contain significant loops..

Do activities: If a longer-running computation (activity) needs to be done while in a state, encode this using the keyword do, followed by a block of code in curly brackets. In languages such as Java that support it, a thread will be started to invoke the code. This allows the state machine to 'stay live' and be able to respond to other events, even while the do activity is running. A do activity thread can be interrupted and cancelled when a transition is taken out of the state. To enable this in Java, the code it needs to catch InterruptedException as shown in the first example.

Example

view plain

```
01. namespace example;
02.
03. class LightFixture
04. {
05.   bulb
06.   {
07.     On {
08.       entry / { doEntry(); }
09.       exit / { doExit(); }
10.       push -> /{ doTransition(); } Off;
11.       do { doThisContinuouslyWhileOn(); }
12.     }
13.     Off {}
14.   }
15.
16.   void doEntry() {System.out.println("Entry");}
```

```
  void doExit() {System.out.println("Exit");}
  void doTransition() {
    System.out.println("Transition");}
  void doThisContinuouslyWhileOn() {
    while (true) {
      System.out.println("Still on");
      try {
        Thread.sleep(1000);
      }
      catch (InterruptedException e) {}
    }
  }
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example with different target languages

view plain

```
// This demonstrates that actions can be different
// in different target languages
class DemoSM
{
  Integer id;
  sm {
    s1 {
      e1 / Java {id=5;} Php {$id=5;} -> s2;
    }
    s2 {
    }
  }
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
// An action can be executed on a transition, or on entry or exit
action : / [[moreCode]]+

entryOrExitAction : [=type:entry|exit] [[guardCode]]? / [[moreCode]]+

// A do activity is long-lasting and can be interrupted
activity : do [[moreCode]]+
```

Nested State Machines

A state machine can be nested inside another. This often allows for simpler Umple (or UML) notation, since events that need to cause effects in every substate of the outer state do not have to be repeated in each of the substates.

The first example below illustrates nesting abstractly. The next example shows nesting in a concrete example. The third example is the same as the second, but with no nesting, to illustrate the difference.

Abstract example of nested state machines

view plain

```
01. // Illustration of nested states. Event e1 will
02. // take the system into state s2, and its first
03. // substrate s2a. Event e2 directly goes to s2b.
04. // Event e3 toggles between s2b and s2a.
05. // Event e1 when in s2, goes to s1
06.
07. class A {
08.   sm {
09.     s1 {
10.       e1 -> s2;
11.       e2 -> s2b;
12.     }
13.     s2 {
14.       e1 -> s1;
15.       s2a {
16.         e3 -> s2b;
17.       }
18.       s2b {
19.         e3 -> s2a;
20.       }
21.     }
22.   }
23. }
24. // @@@skipcppcompile
25.
26.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Concrete example of nested state machines

view plain

```
01. // Example showing nested states.
02.
03. class Course {
04.   code;
05.   description;
06.   1 -- * CourseSection;
07. }
08.
09. class CourseSection
10. {
11.   // Example from Lethbridge and Laganiere: Object
12.   // Oriented Software Engineering: Practical
13.   // Software Development using UML and Java
14.   // McGraw Hill, 2005
15.   // https://www.site.uottawa.ca/school/research/lloseng/
16.
17.   sectionId;
18.   Integer classSize = 0;
19.   Integer minimumClassSize = 10;
20.   Integer maximumClassSize = 100;
21.
22.   // State machine controlling status
23.   status
24.   {
25.     Planned {
26.       openRegistration -> NotEnoughStudents;
27.     }
28.     Open {
29.       cancel -> Cancelled;
30.       NotEnoughStudents {
31.         closeRegistration -> Cancelled;
32.         register
33.           [getClassSize() > getMinimumClassSize()]
34.           -> EnoughStudents;
35.       }
36.       EnoughStudents {
37.         closeRegistration -> Closed;
38.         register
39.           [getClassSize() > getMaximumClassSize()]
40.           -> Closed;
```

```
41.       }
42.     }
43.     Cancelled {}
44.     Closed {}
45.   }
46.
47.   boolean requestToRegister(Student aStudent)
48.   {
49.     register();
50.     return setClassSize(getClassSize()+1);
51.   }
52. }
53.
54. class Student {
55.   //Remainder of this class defined elsewhere
56. }
57.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Concrete example of state machines without nesting

view plain

```
01. // Example with an opportunity to nest states, to
02. // avoid repeating transitions
03.
04. class Course {
05.   code;
06.   description;
07.   1 -- * CourseSection;
08. }
09.
10. class CourseSection
11. {
12.   // Example from Lethbridge and Laganiere: Object
13.   // Oriented Software Engineering: Practical
14.   // Software Development using UML and Java
15.   // McGraw Hill, 2005
16.   // https://www.site.uottawa.ca/school/research/lloseng/
17.
18.   sectionId;
19.   Integer classSize = 0;
```

```
20.   Integer minimumClassSize = 10;
21.   Integer maximumClassSize = 100;
22.
23.   // State machine controlling status
24.   status
25.   {
26.     Planned {
27.       openRegistration -> OpenNotEnoughStudents;
28.     }
29.     OpenNotEnoughStudents {
30.       closeRegistration -> Cancelled;
31.       cancel -> Cancelled;
32.       register
33.         [getClassSize() > getMinimumClassSize()]
34.         -> OpenEnoughStudents;
35.     }
36.     OpenEnoughStudents {
37.       closeRegistration -> Closed;
38.       cancel -> Cancelled;
39.       register
40.         [getClassSize() > getMaximumClassSize()]
41.         -> Closed;
42.     }
43.     Cancelled {}
44.     Closed {}
45.   }
46.
47.   boolean requestToRegister(Student aStudent)
48.   {
49.     register();
50.     return setClassSize(getClassSize()+1);
51.   }
52. }
53.
54. class Student {
55.   // Remainder of class defined elsewhere
56. }
57.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

State Machine Regions

It is possible for a system to be in two states at once. This can be accomplished by dividing any state (we will call it the parent state) into regions separated by || symbols. Each region is like a small state machine, and the system is in one state of each of the regions until it exits the parent state.

In the example below the TrackShuttler starts off initializing then transferringLoad. But after that it starts shuttling (going to the other end of some kind of track). During shuttling, it independently controls its moving in one region, and the lights in the other region. Here shuttling is the parent state. While shuttling the system can be in lightsOn+accelerating, lightsOn+coasting, lightsOff+coasting, lightsOff+braking or in any of the 2*3=6 total configurations of substates of the two regions. When an exit from shuttling occurs due to the reachEnd event, the state machine exits both controllingLights and moving.

In general there can be any number of regions, and regions can be nested in arbitrary ways.

Example

view plain

```
01. class TrackShuttler {
02.   sm {
03.     initializing {
04.       readyToGo -> transferringLoad;
05.     }
06.     transferringLoad {
07.       loaded -> shuttling;
08.     }
09.     shuttling {
10.       reachEnd -> transferringLoad;
11.       moving {
12.         nearEnd -> braking;
13.         accelerating {
14.           reachedMaxSpeed -> coasting;
15.         }
16.         coasting {
17.           tooSlow -> accelerating;
18.         }
19.         braking {
```

```
20.           tooSlow -> coasting;
21.         }
22.       }
23.       ||
24.       controllingLights {
25.         lightsOn {
26.           daylightDetected -> lightsOff;
27.         }
28.         lightsOff {
29.           darknessDetected -> lightsOn;
30.         }
31.       }
32.     }
33.   }
34. }
35. // @@@skipcppcompile
36.
37.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Final States

In Umple, there are two ways to define final states.

The first way is to use the "Final" keyword (with a capital F) as the destination of a transition. This automatically defines a final state for the top-level state machine. Once the state machine is in this state, it is considered to be complete, and it is terminated. The "Final" keyword can be used as shown in the example below. Note, if a user defines a state named "Final", error E074 User Defined State Cannot be Named Final will be thrown.

The second way is to use the "final" keyword (lower case) before a state name to tag a state as final. Using this keyword allows users to define multiple final states as shown in the second example below. In the case of concurrent regions (indicated using the || in the example), all state machines within the same concurrent region must be in their final states before the enclosing region is considered to be complete. In the case of non-concurrent state machines, as soon as a final state is entered, the state machine is considered to be complete.

Example with the "Final" keyword

view plain

```
01. // Using the keyword "Final" automatically defines the
02. // final state for the top-level state machine "sm"
03.
04. class X {
05.   sm {
06.     s1 {
07.       goToS2 -> s2;
08.     }
09.     s2 {
10.       goToFinal -> Final;
11.     }
12.   }
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example of concurrent state machines with "final" states

view plain

```
// "s1" is an orthogonal region, and it is
// considered to be complete when all of the
// concurrent state machines are in their final
// states (i.e. s2 is in state "s4", s5 is in
// state "s7", and s8 is in state "s10")

class X {
  sm {
    s1 {
      s2 {
        s3 {
         goToS4 -> s4;
        }
        final s4 { }
      }
      ||
      s5 {
        s6 {
          goToS7 -> s7;
        }
        final s7 { }
      }
      ||
      s8 {
        s9 {
          goToS10 -> s10;
        }
        final s10 { }
      }
    }
  }
}
// @@@skipcppcompile


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example of non-concurrent state machines with "final" states

view plain

```
01. // "sm" will be considered as complete when
02. // it is in either state "s3" or "s5"
03.
04. class X {
05.   sm {
06.     s1 {
07.       s2 {
08.         goToS3 -> s3;
09.         goToS4 -> s4;
10.       }
11.       final s3 { }
12.     }
13.     s4 {
14.       goToS5 -> s5;
15.       final s5 { }
16.     }
17.   }
18. }
19. // @@@skipcppcompile
20.
21.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Auto-Transitions

It is possible to arrange for a state machine to transition automatically from one state to the next immediately after completing entry actions or upon completion of a do activity. This is specified using a transition without any preceding event. Such transitions may have guards and transition actions.

Example

view plain

```
01. // In this example, the system will transition to s2 automatically
02. // when the do activity ends
03. class X {
04.   sm {
05.     s1 {
06.       entry / {
07.         System.out.println("Starting first sleep");
08.       }
09.       do {
10.         Thread.sleep(2000);
11.         System.out.println("Ending first sleep");
12.       }
13.       -> s2;
14.     }
15.     s2 {
16.       entry / {
17.         System.out.println("Starting second sleep");
18.       }
19.       do {
20.         Thread.sleep(2000);
21.         System.out.println("Ending second sleep");
22.       }
23.     }
24.   }
25.   public static void main(String [ ] args) {
26.     X x = new X();
27.   }
28. }
29.
30.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
autoTransition : [[activity]]? [[autoTransitionBlock]]

// Autotransitions have no event. The transition is immediately taken
// or taken after the do activity ends[[guard]]? -> [[action]]?
// The action can come before or after the arrow
autoTransitionBlock- : [[guard]]?
  ( [[action]] -> | -> [[action]] | ->
  ) [stateName] ;
```

Unspecified Events

In Umple, the keyword 'unspecified' can be used to handle events that have no handlers in the current state (and would normally be ignored) in a queued or regular state machine. If an 'unspecified' transition is present, no event in that state will be ignored. The keyword can be placed at any level of nesting of the state machine, and specifies a transition, using the same syntax as a regular event name (hence it can have a guard, and and action) as well as a destination state.

For example, in the first example below, if finishedClosing occurs while in the closed state, this event would be ignored if 'unspecified' were not present. But the 'unspecified' keyword causes a transition to be taken for any unexpected event while in closed, opening, open or closing states.

Note that tf the 'unspecified' keyword is used in a pooled state machine, it will simply be treated as a regular event since pooled state machines do not ignore events without handlers, but leave them at the head of the queue.

Unspecified transitions are particilarly useful to transition to error-handling states. In the examples below, the system transitions back to normal functioning after handing the error. The use of transitions to history can also be useful in this situation.

Example

view plain

```
01. //A garage door can be modeled
02. //using unspecified events for
03. //error states
04.
05. class GarageDoor {
06.   status {
07.     functionning {
08.       unspecified -> error;
09.
10.       closed {
11.         pressButton -> opening;
12.       }
13.       open {
14.         pressButton -> closing;
15.       }
16.       closing {
17.         finishedClosing -> closed;
```

```
      }
      opening {
        finishedOpening -> open;
      }
    }

    error {
      entry / {
        /* attempt to solve the issue */
      }
      ready -> functionning;
    }
  }
}


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
//The state machine sm makes use
//of the unspecified event to reach
//specific error states to complete
//auto-transitions

//If the machine is idle and receives
//a non-handled event, the destination
//state is error 1, while if the sm was
//in state active, the machines reaches
//error 2

class AutomatedTellerMachine{

  queued sm{

    idle {
      cardInserted -> active;
      maintain -> maintenance;
      unspecified -> error1;
    }
```

```
21.
22.     maintenance {
23.       isMaintained -> idle;
24.     }
25.
26.     active {
27.       entry /{readCard();}
28.       exit /{ejectCard();}
29.       validating {
30.         validated -> selecting;
31.         unspecified -> error2;
32.       }
33.
34.       selecting {
35.         select -> processing;
36.       }
37.
38.       processing {
39.         selectAnotherTransiction -> selecting;
40.         finish -> printing;
41.       }
42.
43.       printing {
44.         receiptPrinted -> idle;
45.       }
46.       cancel -> idle;
47.     }
48.
49.     error1{
50.       ->idle;
51.     }
52.
53.     error2{
54.       ->validating;
55.     }
56.
57.   }
58.   void readCard() {/*Code would be written here*/}
59.   void ejectCard() {/*Code would be written here*/}
60.
61. }
62. // @@@skipcppcompile
63.
64.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
// A transition guard can come before or after the arrow
// The order of guard and event definition can also be interchanged
transition :
  ( [[eventDefinition]] [[guard]]
    | [[guard]] [[eventDefinition]]
    | [=unspecified]? [[guard]]
    | [[eventDefinition]]
  )?
  ( [[action]] ->
    | -> [[action]]
    | ->
  ) [stateName] ;
```

Timed Transitions

A transition can be triggered after a specified delay (a floating point value, given in seconds). The transition will be taken after the delay provided the object is still in the state and any guard is true. This is accomplished using the 'after' keyword.

The 'afterEvery' keyword, tries to trigger a transition repeatedly at a specified interval while the object remains in the state.

Example

view plain

```
01. // In this example, the system will transition to state b
02. // after a 1-second delay
03. class X {
04.   sm {
05.     a {
06.       entry / {System.out.println("entering a");}
07.       after(1) -> b;
08.     }
09.     b {
10.       entry / {System.out.println("entering b");}
11.     }
12.   }
13.   public static void main(String [ ] args) Java {
14.     X x = new X();
15.   }
16.
17.   public static void main(String [ ] args) Python {
18.     x = X()
19.   }
20. }
21.
22.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // In this example, the system will transition to state b
```

```
02. // when the value of the attribute count drops below 5
03. // An attempt to make this transition is taken every second
04. // The do activity slowly drops the value of count
05. class X
06. {
07.   Integer count = 10;
08.   sm
09.   {
10.     a
11.     {
12.       entry / { System.out.println("entering a"); }
13.       do
14.       {
15.         while (count > 0)
16.         {
17.           System.out.println("Count = " + count);
18.           Thread.sleep(1000);
19.           count --;
20.         }
21.       }
22.       afterEvery(1) [count<5] -> b;
23.     }
24.     b
25.     {
26.       entry / { System.out.println("entering b"); }
27.     }
28.   }
29.
30.   public static void main(String [] args)
31.   {
32.     X x = new X();
33.   }
34.
35. }
36.
37.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

// The timer in a timed event can be an number, variable, function() or

function(num)
afterEveryEvent- : afterEvery ([!timer:[+*/a-zA-Z0-9_\.-]+(\([0-9\.]*\))?])

afterEvent- : after ([!timer:[+*/a-zA-Z0-9_\.-]+(\([0-9\.]*\))?])

History states

The history state of any state is its last-visited substate. This is recorded in order to be able to transition to that substate again when needed.

There are two types of history states: Deep history states and regular history states. Deep history stores the specific substate at the deepest level of nesting, while the regular history stores only the first-level substate.

A transition to a history states is indicated using dot notation, by referring to the state to go back to, followed by a dot and either the symbol H to transition to the regular (first-level) history state, or HStar to transition to deep history. When Umple draws a state machine diagram, the deep history appears as H*.

Transitions to history are most commonly used when the system has had to leave normal operation temporarily and needs to later return to exactly what it was doing. Deep history is often appropriate when returning to normal operation, but sometimes regular history is needed in order to allow some form of reinitialization of a substate. When transitioning to history, if there is more than one level of nesting (i.e. the history state has substates), the state machine will enter the start substate of the history state.

Example

view plain

```
01. //The class Machine uses both deep history
02. //and regular history states.
03.
04. //When the event emergency occurs, the
05. //state of normalOperation is stored.
06. //When the issue is resolved, depending
07. //on the intended behaviour, the state
08. //machine goes back to the specific
09. //substate it was previously in (HStar)
10. //or it restarts the phase it was in (H)
11.
12. class Machine {
13.   sm {
14.     initializing {readyForOperation -> normalOperation;}
15.     normalOperation {
16.       phase1 { completePhase1 -> phase2;}
17.       phase2 {
```

```
        subphase2a { complete2a -> subphase2b;}
        subphase2b {}
        completePhase2 -> phase3;
      }
      phase3 {}
      emergency -> handleEmergency;
    }
    handleEmergency {
      resolveEmergency -> normalOperation.HStar;
      resolveButCleanup2a -> normalOperation.H;
    }
  }
}
// @@@skipphpcompile Php does not generate proper
// history state code (see issue 1338)
// @@@skipcppcompile


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
//The class DrawingTool uses a history
//state to get back to the state it was
//previously in after a selection

class DrawingTool {
  sm {
    drawing {
      idle {
        penPress -> drawingStroke;
        colorChange -> changingColor;
      }
      drawingStroke {
        entry / {
          /*deal with the stroke*/
        }
        endStroke -> idle;
      }
      changingColor {
```

```
19.         colorSelected -> idle;
20.       }
21.       changeMode -> selecting;
22.     }
23.     selecting {
24.       selectionCompleted -> drawing.H;
25.     }
26.   }
27. }
28.
29.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Events With Parameters

An event can have a parameter (of any valid type that the generated language can accept). The generated event method will have this parameter. The value of the parameter can be used in guards or transition actions.

It is absolutely necessary that all events with the same name have the same parameter type, or else an error will be raised.

Example

view plain

```
01. // The do activity of the first state machine is a thread that
02. // communicates with the second state machine.
03. // It calls the event method e with different arguments.
04. // The second state machine only changes state when the
05. // guard detects that the argument is valid
06. // The argument is also used in the transition action code.
07. class X {
08.   stateMachine1 {
09.     s1a {
10.       do {
11.         // This do activity sends events to stateMachine2
12.         e(5);
13.         Thread.sleep(1000);
14.         e(6);
15.         Thread.sleep(1000);
16.         e(7);
17.         Thread.sleep(1000);
18.         e(8);
19.       }
20.       -> s1b;
21.     }
22.     s1b {}
23.   }
24.   stateMachine2 {
25.     s2a {
26.       entry / {System.out.println("s2a");}
27.       e(int a) [a > 6] / {System.out.println("e"+a);} -> s2b;
28.     }
29.     s2b {
30.       entry / {System.out.println("s2b");}
31.       e(int a) / {System.out.println("e"+a);} -> s2a;
```

```
32.     }
33.   }
34.   public static void main(String [ ] args) {
35.     X x = new X();
36.   }
37. }
38.
39.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

Queued State Machines

Standard state machines operate in a single thread. When an event method is called, the state machine code that runs continues the same thread of the caller. This can be satisfactory for simple applications, but it doesn't work in multi-threaded environments, and can also result in deadlocks.

To overcome the above problems, a state machine may be declared as 'queued' by just placing the keyword queued before the declaration of the state machine. In a queued state machine, the calls to the event methods simply add a record to the queue, and then return. The calling thread can then continue and do other activities. A separate thread exists in each state machine to take events off the queue and process them, in the order they are received. The thread will only process the event at the head of the queue when in a state that has a transition corresponding to that event; otherwise it will wait until the state machine enters such a state.

A queued state machine will process events in the same order as a regular state machine. See also pooled state machines for a variation on this semantics.

Example

view plain

```
01. // Test of queued state machines.
02. // The word queued results in a thread being
03. // created to process events.
04.
05. class TestSM {
06.   queued sm{
07.     s1 {
08.       e1 /{System.out.println("e1");} ->s2;
09.     }
10.     s2 {
11.       e2 /{System.out.println("e2");} ->s3;
12.     }
13.     s3 {
14.       e3 /{System.out.println("e3");} ->s4;
15.     }
16.     s4 {
17.       e4 /{System.out.println("e4");} ->s1;
18.     }
19.   }
```

```
20.
21.   public static  void main(String [] ags){
22.    TestSM test=new TestSM();
23.    test.e1(); // processed s2
24.    test.e2(); // processed s3
25.    test.e3(); // processed s4
26.    test.e4(); // processed s1
27.
28.    test.e1(); // processed s2
29.
30.            // queued: ignored
31.            // pooled: left in queue, until we
32.    test.e3(); //   are next in s3
33.
34.            // pooled: left in queue, until
35.    test.e4(); //   next in s4
36.
37.            // pooled: processed goes to s3
38.            // pooled: process e3 from queue
39.            //   goes to s4
40.            // pooled: process s4 from queue
41.    test.e2(); //   goes to s1
42.
43.    test.e1();
44.    test.e3();
45.    test.e2();
46.    test.e4();
47.
48.    test.e1();
49.    test.e2();
50.    test.e3();
51.    test.e4();
52.
53.    test.e1();
54.    test.e2();
55.    test.e4();
56.    test.e3();
57.    test.e4();
58.
59.    test.e1();
60.    test.e3();
61.    test.e2();
62.    test.e3();
63.    test.e1();
64.    test.e4();
```

```
65.
66.     test.e2();
67.     test.e1();
68.     test.e2();
69.
70.   }
71. }
72.
73.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
//Issue 148
inlineStateMachine : [=queued]? [=pooled]? [~name] {
  ( [[comment]]
    | [[reqImplementation]]
    | [[state]]
    | [[trace]]
    | [[mixsetDefinition]]
    | [=||]
    | [[standAloneTransition]]
  )* }
```

Pooled State Machines

Queued and regular state machines always process events in the order they arrive, and ignore events that cannot be handled in the current state. Sometimes, however, you want to 'save' events that arrive out of sequence, and process them as soon you enter a state that can handle them. This is accomplished by adding the word 'pooled' before a state machine definition.

In a pooled state machine there is a queue in a separate thread, just like in a queued state machine, however, if the next event to be processed has no handler in the current state, then it remains at the front of the queue, and the queue processor looks further back in the queue for the first event that can be handled.

A pooled state machine will therefore often process events in a different order from a regular or queued state machine.

Example

view plain

```
01. // Test of pooled state machines.
02. // The word pooled results in a thread being
03. // created to process events, and 'pooled'
04. // semantics being employed. Events which cannot
05. // be handled are kept waiting until entry into a
06. // state where they can be handled.
07.
08. class TestSM {
09.   String ev="";
10.   pooled sm{
11.    s1 {
12.      e1 /{ev="e1";} ->s2;
13.      e5 /{ev="e5";} ->s2;
14.    }
15.    s2 {
16.      e2 /{ev="e2";} ->s3;
17.    }
18.    s3 {
19.      e3 /{ev="e3";} ->s4;
20.    }
21.    s4 {
22.      e4 /{ev="e4";} ->s1;
23.    }
```

```
24.   }
25.   after e* {
26.     if(wasEventProcessed)
27.       System.out.println(ev);
28.   }
29.
30.   public static  void main(String [] ags){
31.     TestSM test=new TestSM();
32.     test.e1(); // processed s2
33.
34.             // Will only be processed in the
35.     test.e5(); // pooled case (eventually)
36.
37.     test.e2(); // processed s3
38.     test.e3(); // processed s4
39.     test.e4(); // processed s1
40.
41.     test.e1(); // processed s2
42.
43.             // queued: ignored
44.             // pooled: left in queue, until we
45.     test.e3(); //   are next in s3
46.
47.             // pooled: left in queue, until
48.     test.e4(); //   next in s4
49.
50.             // pooled: processed goes to s3
51.             // pooled: process e3 from queue
52.             //   goes to s4
53.             // pooled: process s4 from queue
54.     test.e2(); //   goes to s1
55.
56.
57.     test.e1();
58.     test.e3();
59.     test.e2();
60.     test.e4();
61.
62.     test.e1();
63.     test.e2();
64.     test.e3();
65.     test.e4();
66.
67.     test.e1();
68.     test.e2();
```

```
69.     test.e4();
70.     test.e3();
71.     test.e4();
72.
73.     test.e1();
74.     test.e3();
75.     test.e2();
76.     test.e3();
77.     test.e1();
78.     test.e4();
79.
80.     test.e2();
81.     test.e1();
82.     test.e2();
83.
84.   }
85. }
86.
87.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
//Issue 148
inlineStateMachine : [=queued]? [=pooled]? [~name] {
  ( [[comment]]
    | [[reqImplementation]]
    | [[state]]
    | [[trace]]
    | [[mixsetDefinition]]
    | [=||]
    | [[standAloneTransition]]
  )* }
```

Standalone State Machines

State machines can be defined at the top level of a model. This feature allows the definition of reusable state machines, to simplify code, or to facilitate separation of concerns.

The example below shows that the *statemachine* keyword is used to specify a standalone state machine. Such a state machine, by itself, must be syntactically correct, and error messages will be generated, enabling debugging. However, the state machine is not displayed as a diagram and will not result in any code generation unless it is incorporated in one or more classes. The *as* keyword is used to do incorporate a standalone state machine in a class, as shown in the example.

An alternative, and equivalent, way to define state machines outside of classes is to use traits. A model equivalent to the example below is shown in Example 5 of the page on state machines in traits.

Standalone state machines can be used to make code appear simpler than it does when the state machine is directly incorporated in a class or trait, as there is less indentation, and the statemachine keyword makes it more explicit that what is being described is a state machine.

Standalone state machine used in two classes

view plain

```
01. // Class Diagram illustrating use of top-
    level state machine with
02. // usage in separate classes.
03. // Note that this could also be accomplished by putting the
04. // state machine in a trait, and then having each class use the tra
05.
06. class MotorController {
07.   motorStatus as deviceStatus;
08. }
09.
10. class BrakeController {
11.   brakeStatus as deviceStatus;
12. }
13.
14. // By itself, the following will not generate any code
15. // but it can be debugged by itself outside of any class
16. // and can be incorporated in multiple classes, as above
17. statemachine deviceStatus {
18.   inactive {
```

```
    activate -> booting;
  }
  booting {
    completedStartupChecks -> active;
    startupCriticalErrorDetected -> outOfOrder;
  }
  active {
    runtimeCriticalErrorDetected -> outOfOrder;
    deactivate -> shuttingDown;
  }
  shuttingDown {
    shutdownComplete -> inactive;
  }
  outOfOrder {
    repaired -> inactive;
  }
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Standalone state machine modified when reused

view plain

```
// Class Diagram illustrating standalone state machine being
// modified when it is reused, to add additional events
// states and entry/exit actions

class MotorController {
  Boolean warmup = false;
  motorStatus as deviceStatus {
    // Add a transition to the reused state machine
    cancel booting-> inactive;
    completedStartupWhilewarning  booting -> warm ;

    // Add transition with action
    warmSoReady inactive -> / {warmup=true;} booting ;

    // Add an entirely new state
    warm {
      go -> active;
```

```
    }

    // add an entry action to an existing state
    inactive {
      entry / {warmup=false;};
    }
  };
}

// Reusable standalone state machine
statemachine deviceStatus {
  inactive {
    activate -> booting;
  }
  booting {
    completedStartupChecks -> active;
    startupCriticalErrorDetected -> outOfOrder;
  }
  active {
    runtimeCriticalErrorDetected -> outOfOrder;
    deactivate -> shuttingDown;
  }
  shuttingDown {
    shutdownComplete -> inactive;
  }
  outOfOrder {
    repaired -> inactive;
  }
}
// @@@skipphpcompile - contains java code
// @@@skippythoncompile - Contains Java Code


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

// State machine elements in Umple. See user manual page: BasicStateMachines
stateMachineDefinition : statemachine [=queued]? [=pooled]? [name]

```
{ [[state]]* }

referencedStateMachine : [name] as [definitionName]
  ( { [[extendedStateMachine]] } | ;
  )

//Issue 547 and 148
extendedStateMachine# :
  ( [[comment]]
    | [=changeType:+| - |- |*]? [[state]]
    | [[standAloneTransition]]
  )*
```

Composing State Machines

State machines can be built from parts. Elements of state machines can be added in several ways:

- By repeating the state machine definition in the same class, adding more detail. In the example below, the state machine sm has part of its definition starting at line 2, and part of its definition starting at line 26.
- By using standalone transitions. Line 20 shows that outside the context of any state, a transition can be defined. Just add the origin state before the arrow. This one indicates that when e3 occurs, if in state b, then go to state a. This notation can be used to help separate concerns, and place focus on a particular transition. Another example of this is at line 31, where event e5 can cause a transition from state bsub1 to state a. Note that line 34 shows another way of indicating a standalone transition, by adding more detail to state a (that had already beed defined starting at line 3).
- By using mixins where a more detail to a class (and its contained state machine) is added. Starting at line 26, there is a new definition adding more detail to class X, which in turn adds more detail to state machine sm starting at line 29.
- By using mixsets: Mixsets are like mixins but can be either excluded or included in the system. To include a mixset,add a use statement (as i line 49) or command line option. An example starts at line 39, where yet more detail is added to class X and its state machine sm. This is activated with the use statement at line 49.

Illustration of various ways of adding details to a state machine

view plain

```
01. class X {
02.   sm {
03.     a {
04.       // Ordinary transition
05.       e1-> bsub1;
06.     }
07.     b {
08.       bsub1 {
09.         // transition in substate
10.         e2 -> bsub2;
11.       }
```

```
12.       bsub2 {}
13.     }
14.     // Standalone transition
15.     // at to level
16.     e3 b -> a;
17.
18.     // Standalone transition
19.     // between substate
20.     e4 bsub2 -> bsub1;
21.   }
22. }
23.
24. // Mixin for class that mixes in
25. // more details into the state machine
26. class X {
27.   // Supplemental detail for same
28.   // state machine
29.   sm {
30.     // Standalone transition
31.     e5 bsub1 -> a;
32.
33.     // Another way of adding a transition
34.     a {e6 -> bsub2;}
35.   }
36. }
37.
38. // Mixset to optionally add more detail
39. mixset extra class X {
40.   sm {
41.     // Adding a new state
42.     c {}
43.     // Adding a transition
44.     e7 b -> c;
45.   }
46. }
47.
48. // Activating the above mixset
49. use extra;
50.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
// State machine elements in Umple. See user manual page:
BasicStateMachines
stateMachineDefinition : statemachine [=queued]? [=pooled]? [name]
{ [[state]]* }

state : [=final]? [stateName] { [[stateInternal]]* }

// A transition guard can come before or after the arrow
// The order of guard and event definition can also be interchanged
transition :
  ( [[eventDefinition]] [[guard]]
    | [[guard]] [[eventDefinition]]
    | [=unspecified]? [[guard]]
    | [[eventDefinition]]
  )?
  ( [[action]] ->
    | -> [[action]]
    | ->
  ) [stateName] ;

//Issue148
standAloneTransition :
  ( [[eventDefinition]] [[guard]]
    | [[guard]] [[eventDefinition]]
    | [=unspecified]? [[guard]]
    | [[eventDefinition]]
  )? [~fromState]
  ( [[action]] ->
    | -> [[action]]
    | ->
  ) [~toState] ;
```

Method Definition

The Umple language allows a developer to extend the functionality of a class with arbitrary methods written in the natively compiled language (e.g. Java, Php or Ruby).

Within these arbitrary methods, a developer may call generated methods that access the Umple attributes, associations and state machines. To determine what API methods are available to be called by methods, refer to the API reference or generate Javadoc from an Umple file using UmpleOnline.

A standard Umple method will specify the return type, then the name, then the argument list and finally the method body in curly brackets. The generated output for the method will use correct format for the generated language and will be public.

Example of a method with no arguments

view plain

```
generate Java;

// A class with a method displayName() that has no arguments
class Person
{
  name;

  String displayName()
  {
    return("Hello, my name is " + getName());
  }
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example of a method with arguments

view plain

```
generate Java;

// A class with a method that has arguments
```

```
04. class Person
05. {
06.   name;
07. 
08.   String displayName(String greeting, String property)
09.   {
10.     return(
11.       greeting +
12.       ", my " +
13.       property +
14.       " is " +
15.       getName());
16.   }
17. }
18. 
19. 
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example of a public method

view plain

```
01. generate Java;
02. 
03. // A class with a method that uses 'public'.
04. class Person
05. {
06.   name;
07. 
08.   public String displayName()
09.   {
10.     return("Hello, my name is " + getName());
11.   }
12. }
13. 
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
// Methods: The code in concrete methods is passed to the base language
// See user manual page MethodDefinition
concreteMethodDeclaration : [=modifier:public|protected|private]?
[=static]? [=synchronized]? [=queued]? [type]? [[methodDeclarator]]
[[methodThrowsExceptions]]? [[methodBody]]
    | [=modifier:public|protected]? [=abstract] [type]?
[[methodDeclarator]] [[methodThrowsExceptions]]? ;

methodDeclarator : [methodName] [[parameterList]]

parameterList : (
  ([[parameter]]
  ( , [[parameter]]
  )*
  )? )

parameter : [[typedName]]
```

Abstract Methods

An abstract method is a method declaration with no implementation. These methods are to be implemented in classes with the specific behaviour intended.

Abstract methods can be defined in classes using the keyword "abstract", followed by the method's signature. By declaring a method abstract, the class containing it is abstract as well, meaning no instances of it can be created (the new keyword cannot be used to create an object of the class). The abstract class must have one or more subclasses that have concrete implementations of the abstract method.

An interface can only contain abstract methods, and the keyword is not needed. The usage of the abstract keyword in a trait declares a required method.

Example

view plain

```
01. //The class Person contains abstract
02. //methods, which are inherited and
03. //implemented by its subclass Student
04.
05. //Person is abstract, but Student is a
06. //concrete class as it has no methods
07. //that are unimplemented.
08. class Person {
09.   abstract void method1();
10.   abstract void method2();
11.   void method3() Java {
12.     /* Implementation */
13.   }
14.
15.   void method3() Python {
16.     ''' Implementation '''
17.   }
18. }
19.
20. class Student {
21.   isA Person;
22.
23.   void method1() Java {
24.     /* Implementation */
```

```
  }
  void method2() Java {
    /* Implementation */
  }

 void method1() Python {
    ''' Implementation '''
  }
  void method2() Python {
    ''' Implementation '''
  }
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
//The class Shape contains an abstract
//method, area, as well as a concrete
//method move, which has its implementation
//shared across all subclasses

//Shape is an abstract class, because it
//contains an abstract method
class Shape {
  color;
  Integer x;
  Integer y;

  abstract Double area();
  void move() Java {
    /* implementation */
  }
}

//Both Square and Circle are concrete
//classes as they implement all inherited
//abstract methods
class Square {
  isA Shape;
```

```
  Double width;
  Double height;

  Double area() Java {
    return getWidth()*getHeight();
  }

  Double area() Python {
    return self.getWidth()*self.getHeight()
  }

}

class Circle {
  isA Shape;
  Double radius;
  const Double PI = 3.1416;

  Double area() Java {
    return PI*radius*radius;
  }

  Double area() Python {
    return self.PI*self.radius*self.radius
  }
}

class A {
  Square s;
  Circle c;

  void Test() Java {
    //The method implemented in Shape is
    //used
    s.move();
    c.move();

    //The methods implemented in their
    //respective classes are used
    s.area();
    c.area();
  }

  void Test() Python {
    # The method implemented in Shape is
```

```
69.     # used
70.     self._s.move()
71.     self._c.move()
72.
73.     # The methods implemented in their
74.     # respective classes are used
75.     self._s.area()
76.     self._c.area()
77.   }
78. }
79.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
// Instructs a class or method to be abstract
abstract- : [=abstract] ;
```

Alternative Languages

In Umple, the body of a method will by default be emitted unchanged. This ties the model to the particular programming language. However it is possible to provide alternative implementations for different programming languages by preceding the method body with the language name.

This multi-lingual capability also works for other places where native code is injected such as in state machine guards and actions, or in derived attributes.

Example

view plain

```
01. // Generating Java and Php the following will
02. // result in the appropriate
03. // method body being selected.
04. class LanguageSpecific {
05.   int m1() Java {
06.     // Return 1 if the model is compiled into Java
07.     return 1;
08.   }
09.   Php {
10.      // Return 0 if the model is compiled into Php
11.     return 0;
12.   }
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // A method body may work for several languages.
02. // If no language is specified it is taken as the
03. // default.
04. class LanguageSpecific {
05.   int m1()
06.   Java,Php {
07.     // Return 0 if the model is compiled into
08.     // Java or Php
```

```
09.     return 0;
10.   }
11.   {
12.     // Default implementation for other languages
13.     return 1;
14.   }
15. }
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

State-Dependent Methods

A state-dependent method is a partial method body declared within a state machine. Each of these partial definitions are subsequently aggregated into a single function within the Umple class as switch cases.

Note that while it is possible to define state-dependent methods across several state machines, the switched cases must be disjoint as the order within the cases is non-deterministic. Additionally, if a method of the same name and parameter set is provided in the same class then it is used as the default body for the function.

Example

view plain

```
01. class Mario {
02.   Modifiers {
03.     Small {
04.       Normal {
05.         void onHit() {
06.           Die();
07.         }
08.       }
09.
10.       Invulnerable {
11.         void onHit() {
12.           // took no damage
13.         }
14.       }
15.
16.       void render() {
17.         // draw small mario
18.       }
19.
20.       EatMushroom -> Large;
21.       Star -> Small.Invulnerable;
22.       Die -> Dead;
23.     }
24.
25.     Large {
26.       Normal {
27.         void onHit() {
28.           Shrink();
29.         }
```

```
30.       }
31.
32.       Invulnerable {
33.         void onHit() {
34.           // took no damage
35.         }
36.       }
37.
38.       void render() {
39.         // draw large mario
40.       }
41.
42.       Shrink -> Small;
43.       Star -> Large.Invulnerable;
44.     }
45.
46.     Dead {
47.       void render() {
48.         // draw dying animation
49.       }
50.     }
51.   }
52. }
53.
54. // @@@skipphpcompile - See #1351. State-
    dependent methods are exclusive to Java generation.
55. // @@@skipcppcompile
56.
57.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. external Thread {};
02.
03. class Philosopher {
04.
05.   isA Thread;
06.
07.   Fork leftFork;
```

```
08.   Fork rightFork;
09.   String fname;
10.   boolean isLeftHanded;
11.
12.   ForksInHand {
13.     NoForks {
14.       LookingForForks {
15.         void eat(){
16.           if (isLeftHanded) {
17.             if (leftFork.take()) {
18.               System.out.println(fname + " picked up first fork (left)"
19.               LeftForkAcquired();
20.             }
21.           }
22.           else if (rightFork.take()) {
23.             System.out.println(fname + " picked up first fork (right)"
24.             RightForkAcquired();
25.           }
26.         }
27.         AteFood -> Idling;
28.       }
29.       Idling {
30.         BellyGrumble -> LookingForForks;
31.       }
32.       LeftForkAcquired -> ForkInLeft;
33.       RightForkAcquired -> ForkInRight;
34.     }
35.     ForkInLeft {
36.       void eat() {
37.         if (rightFork.take()) {
38.           System.out.println(fname + " picked up second fork (right
39.           RightForkAcquired();
40.         }
41.       }
42.       RightForkAcquired -> ForkInBoth;
43.     }
44.
45.     ForkInRight {
46.       void eat() {
47.         if (leftFork.take()) {
48.           System.out.println(fname + " picked up second fork (left)"
49.           LeftForkAcquired();
50.         }
51.       }
52.       LeftForkAcquired -> ForkInBoth;
```

```
53.     }
54.
55.     ForkInBoth {
56.       void eat() {
57.         System.out.println(fname + " is eating.");
58.         try {
59.           sleep(1000);
60.         } catch (InterruptedException ex) { }
61.         AteFood();
62.         leftFork.putDown();
63.         rightFork.putDown();
64.       }
65.       AteFood -> NoForks;
66.     }
67.   }
68.
69.   Belly {
70.     IsHungry {
71.       boolean isHungry() {
72.         return true;
73.       }
74.       AteFood -> IsFull;
75.     }
76.     IsFull {
77.       void think() {
78.         System.out.println(fname + " is full. Now thinking...");
79.         try {
80.           sleep(1000);
81.         } catch (InterruptedException ex) { }
82.         BellyGrumble();
83.       }
84.       BellyGrumble -> IsHungry;
85.     }
86.   }
87.
88.   public void run() {
89.     while (true) {
90.       if (isHungry()) {
91.         eat();
92.       } else {
93.         think();
94.       }
95.     }
96.   }
97. }
```

```
98.
99.  class Fork {
100.   depend java.util.concurrent.*;
101.   Semaphore fork = new Semaphore(1);
102.
103.   boolean take() {
104.     try {
105.       Thread.sleep(100);
106.     } catch (InterruptedException ex) { }
107.     return fork.tryAcquire();
108.   }
109.
110.   void putDown() {
111.     fork.release();
112.   }
113. }
114.
115. class Main {
116.   public static void main(String[] args){
117.     String[] names = {"Plato", "Aristotle", "Cicero", "Confucius", "E
118.     int k = names.length;
119.     Philosopher[] philosophers = new Philosopher[k];
120.     Fork[] forks = new Fork[k];
121.
122.     for (int i = 0; i < k; ++i) {
123.       forks[i] = new Fork();
124.     }
125.
126.     for (int i = 0; i< k; ++ i) {
127.       Fork leftFork = forks[i];
128.       Fork rightFork = forks[(i + 1) % forks.length];
129.       philosophers[i] = new Philosopher(leftFork, rightFork, names
      1);
130.       philosophers[i].start();
131.     }
132.   }
133. }
134.
135. // @@@skipphpcompile - See #1351. State-
     dependent methods are exclusive to Java generation.
136. // @@@skipcppcompile
137.
138.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
// Methods: The code in concrete methods is passed to the base language
// See user manual page MethodDefinition
concreteMethodDeclaration : [=modifier:public|protected|private]?
[=static]? [=synchronized]? [=queued]? [type]? [[methodDeclarator]]
[[methodThrowsExceptions]]? [[methodBody]]
    | [=modifier:public|protected]? [=abstract] [type]?
[[methodDeclarator]] [[methodThrowsExceptions]]? ;
```

Traits

A trait is a group of class content items that serves as building blocks for classes.

A trait implements its main functionality through methods, state machines, and other modeling elements used for modeling behavior.

Clients of traits: Clients of traits can be classes and other traits. In other words, a trait is a partial description of a class that can be reused in several different classes, and traits can form hierarchies. The client uses the isA clause to include the trait content.

Enabling multiple inheritance: Traits can be used in place of standard inheritance where there is already a superclass, since multiple inheritance is not allowed in Umple to be consistent with Java and several other languages.

Separation of concerns: Traits can be used to inject attributes, associations, state machines, constraints, methods and many other elements. They are one of several approaches in Umple to separation of concerns. The others are the mixin ability (ability to specify a class several times and have the elements added together), mixsets, and the aspect oriented capabilities. Note that traits themselves are subject to being mixed in. You can declare two parts of the same trait in two different places in an Umple system.

Defining traits: Umple traits are defined through the keyword '*trait*' followed by a unique name and a pair of curly brackets. The name must be alphanumeric and start with an alpha character, or the symbol (underscore). We also recommend capitalizing the first letter of traits names, as is the case for classes and interfaces in Umple. All elements of traits are defined inside the curly brackets except template parameters defined between the name and the curly brackets. There are separate pages describing how to incorporate attributes, state machines, template parameters, required interfaces, and associations into traits.

Traits are not types: An Umple trait cannot be used as a type of a variable (whereas a class or interface can).

Required methods: Traits also can have missing functionality which is defined as a set of *required* (abstract) methods. These required methods must be provided by clients of traits, either directly or indirectly. Required methods are defined similarly to the way abstract methods are defined in classes. They have exactly the same syntax, but it is also possible in traits to define required methods without the keyword *abstract*. If a method is

defined like a normal method without a body (or implementation), the Umple compiler will consider that as a required method.

Provided methods: These are defined in the same way as concrete methods are defined in classes. Indeed, they have exactly the same syntax and semantics. Provided methods also support multiple code blocks, for generation of systems in different languages.

Diagramming: Traits are not part of UML. A UML class diagram drawn from an Umple file containing traits will 'flatten' the traits. Umple can, however, be made to generate a special diagram that shows traits; this can be requested via the Options menu in UmpleOnline and in the generate clause of the compiler.

Use with mixsets: Creating a mixset that defines both a trait and isA statements incorporating the trait into clients allows the specification of optional functionality that can be mixed in to several classes. This is how mixins work in languages such as Ruby.

Elements copied into client classes: The trait elements will appear in all classes that include that trait. In the first example below, the name and address attribute will appear in the class diagram of both Person and Company

The example 2 below shows a trait called TEquality. It has a required methods named isEqual and a provided method named isNotEqual. The provided methods uses the required method to satisfy part of its functionality.

When exploring the following examples in UmpleOnline, you can use the Options menu to control what is visible, or you can use control-R to flip back and forth between showing the diagram with the original traits, vs. the diagram collapsed into the classes to be compiled; or you can use control-M to show/hide methods.

Example

view plain

```
01. /*
02.  Example 1: use of a trait in Umple
03.  The trait has regular attributes, derived
04.  attribute and a method that are copied into
05.  Organization and Company
```

```
06.
07.   To see different diagram views in UmpleOnline:
08.     Use control-g for auto-layout
09.       (if not already showing)
10.     Use control-r to switch between trait view and
11.        plain classes resulting from applying traits
12.     Use control-m to show/hide methods
13.  */
14.  trait Identifiable {
15.    firstName;
16.    lastName;
17.    address;
18.    phoneNumber;
19.    fullName = {firstName + " " + lastName}
20.    Boolean isLongName() { return lastName.length() > 1;}
21.  }
22.  class Person {
23.    isA Identifiable;
24.  }
25.  class Organization {
26.    Integer registrationNumber;
27.  }
28.  class Company {
29.    isA Organization, Identifiable;
30.  }
31.  // @@@skipcppcompile
32.
33.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

```
view plain

01.  /*
02.     Example 2: showing a trait designed for
03.     comparison regarding if two objects are equal
04.     or not.
05.
06.     To see different diagram views in UmpleOnline:
07.       Use control-g for auto-layout
08.         (if not already showing)
```

```
09.       Use control-r to switch between trait view and
10.         plain classes resulting from applying traits
11.       Use control-m to show/hide methods
12.  */
13.  trait TEquality{
14.    Boolean isEqual(Object object);
15.    Boolean isNotEqual(Object object){
16.      return isEqual(object) ? true : false;
17.    }
18.  }
19.  // @@@skipcompile Java code not compilable no class
20.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

traitDefinition : trait [name] [[traitParameters]]? { [[traitContent]]* }

traitContent- : [[mixsetDefinition]]
| [[testCase]]
| [[requiredModelElements]]
| [[comment]]
| [[reqImplementation]]
| [[traitDefinition]]
| [[trace]]
| [[position]]
| [[displayColor]]
| [[abstract]]
| [[keyDefinition]]
| [[softwarePattern]]
| [[depend]]
| [[symmetricReflexiveAssociation]]
| [[attribute]]
| [[stateMachine]]
| [[inlineAssociation]]
| [[concreteMethodDeclaration]]
| [[abstractMethodDeclaration]]
| [[constantDeclaration]]
| [[invariant]]
| ;
| [[exception]]

| [[extraCode]]

Clients of Traits

Traits are used by clients (classes and traits) by specifying the keyword '*isA*' followed by their names and a semi-colon. Traits cannot use themselves (even through several uses in a cyclic manner). When clients specify names of traits for use, those traits must exist in the system. If a trait uses more than one trait, it can separate them by comma or use the keyword '*isA*' for each one.

In the example 1 below, trait T2 is a client that uses trait T1 (at line 11) and class C2 is a client that uses trait T2 (at line 21). Clients can use other traits if they satisfy their required methods. Satisfaction of required methods is performed by having them implemented in clients. For example, trait T2 uses trait T1 and so it is required to have the required methods of trait T1 implemented in trait T2 (or in clients of trait T2). Trait T2 achieves this through implementing two methods named method1() and method2() defined in lines 13 and 14.

Trait T2 is not a final client, therefore, it could use trait T1 without implementing those required methods, in that case, the required methods of trait T2 would be method1(), method2(), and method 3(). Class C2, which is a final client, uses trait T2 and therefore needs to implement the remaining required method method3(). However, there is no direct implementation for it. Instead, class C2 obtains such an implementation indirectly from its superclass, which is C1. Therefore, it satisfies the required method of trait T2.

When clients use traits, they obtain all provided methods defined in the traits. This includes all other provided methods those traits might obtain from their own used traits. For example, trait T2 gets the provided method method4() from trait T1. This provided method can be called by all other provided methods defined in trait T2. Therefore trait T2 provides three provided methods named method1(), method2(), and method4(). Class C2 uses trait T2 and so it obtains all provided methods.

Note that it is allowed for abstract classes to use traits without satisfying their required methods. Such required methods are then considered as abstract methods of the abstract class. Then, all concrete subclasses of the abstract class are required to implement those abstract methods. This process ensures that required methods will finally be implemented. It is worth noting that interfaces cannot use traits because they cannot have concrete methods in their definitions.

When exploring the following example in UmpleOnline, you can use the Options menu to control what is visible, or you can use control-R to flip back and forth between showing the diagram with the original traits, vs.

the diagram collapsed into the classes to be compiled; or you can use control-M to show/hide methods.

Example

view plain

```
/*
  Example 1: showing how traits can be used by
  their clients.

  To see different diagram views in UmpleOnline:
    Use control-g for auto-layout
      (if not already showing)
    Use control-r to switch between trait view and
       plain classes resulting from applying traits
    Use control-m to show/hide methods
*/
trait T1{
  abstract void method1();
  abstract void method2();
  void method4() Java {/*implementation*/ }
  void method4() Python { ''' implementation ''' }
}
trait T2{
  isA T1;
  void method3();
  void method1() Java {/*implementation*/ }
  void method2() Java {/*implementation*/ }
  void method1() Python { ''' implementation ''' }
  void method2() Python { ''' implementation ''' }
}
class C1{
  void method3() Java {/*implementation*/ }
  void method3() Python { ''' implementation ''' }
}
class C2{
  isA C1;
  isA T2;
  void method2() Java {/*implementation*/ }
  void method2() Python { ''' implementation ''' }
}

```

Load the above code into classic UmpleOnline

Load the above code into UmpleOnline V2

Attributes in Traits

Attributes in traits are defined in the same way they are defined for classes. Traits also support all modifiers that can be applied to attributes. The example 1 below shows the way attributes can be defined and used in traits.

As seen, there is a trait named Identifiable that has five attributes: first-Name, lastName, address, phoneNumber, and fullName. It also has a provided method named isLongName(). There are no required methods because the trait offers pure functionality to its clients. Class Person uses trait Identifiable and obtains the provided method and defined attributes from the trait. Class Company also uses the trait Identifiable and extends class Organization. It obtains both attributes coming from its superclass and used trait. All attributes are flattened similar to the way provided methods are flattened.

When clients use traits, a name conflict might happen because a client might have an attribute and obtains a new attribute with the same name from a trait. Modelers are responsible to resolve the conflict. The conflict is detected automatically. Unlike conflicts with other elements in traits, in the current implementation of our work, there is no operator to resolve an attribute name clash conflict. Our current recommendation is to change the name of attributes in the clients of traits and avoid changing names in traits because this could break other clients of those traits.

Another way to avoid conflicts is to use stateless traits. In that case, traits use required methods to have access to states (attributes). The example 2 below shows trait T1 uses two required methods getData() and setData(Integer) to have access and write to the attribute data of type Integer. It also has a provided method that uses the method getData() to obtain the value of the attribute data. Then, it performs some operation on it (in this case adding 2 to the value,) and finally uses the method setData(Integer) to save the new value into the attribute data. The class C1 uses trait T1 and has the attribute data. Since Umple automatically generates accessor and mutators for attributes, they satisfy the required methods defined in trait T1 and so there is no need to implement them manually in class C1.

When exploring the following examples in UmpleOnline, you can use the Options menu to control what is visible, or you can use control-R to flip back and forth between showing the diagram with the original traits, vs. the diagram collapsed into the classes to be compiled; or you can use control-M to show/hide methods. (NOTE: on some browsers, these

keyboard shortcuts may do other things that override umple. For instance, control-R reloads tabs in some browsers, which may take precedence over umple's built in keyboard commands)

Example showing how to use a trait in Umple

view plain

```
/*
 Example 1: use of a trait in Umple
 The trait has regular attributes, derived
 attribute and a method that are copied into
 Organization and Company

  To see different diagram views in UmpleOnline:
    Use control-g for auto-layout
      (if not already showing)
    Use control-r to switch between trait view and
       plain classes resulting from applying traits
    Use control-m to show/hide methods
*/
trait Identifiable {
  firstName;
  lastName;
  address;
  phoneNumber;
  fullName = {firstName + " " + lastName}
  Boolean isLongName() { return lastName.length() > 1;}
}
class Person {
  isA Identifiable;
}
class Organization {
  Integer registrationNumber;
}
class Company {
  isA Organization, Identifiable;
}
// @@@skipcppcompile


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example showing how methods can obtain states in traits

view plain

```
01. /*
02.  Example 2: showing how states in traits can be
03.  obtained by required methods.
04.
05.   To see different diagram views in UmpleOnline:
06.     Use control-g for auto-layout
07.       (if not already showing)
08.     Use control-r to switch between trait view and
09.        plain classes resulting from applying traits
10.     Use control-m to show/hide methods
11. */
12. trait T1{
13.   int getData();
14.   int processData() {
15.     int data = getData();
16.     data=data+2;
17.     setData(data);
18.     return(data);
19.   }
20. }
21. class C1{
22.   Integer data;
23.   isA T1;
24. }
25. // @@@skipjavacompile @@@skipphpcompile @@@skippythoncc
26.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Trait Template Parameters

Template parameters at the modeling level are combined with other modeling elements like associations. This combination increases modularity and reusability to an extent that is not achievable at the implementation level. Template parameters can be referred to in required and provided methods and attributes. Traits can have template parameters with generic or primitive data types. The difference in their use is that it is possible to put restrictions on bound types of generically-typed parameters. Such, restrictions might include a declaration that the interfaces or classes must be extended or implemented by other specific classes. These restrictions are only available for generic-type parameters because primitive types cannot implement or extend any other types and so there is no way of imposing such constraints on them.

Template parameters are defined after the name of a traits inside a pair of angle brackets. Each parameter has a name and they are separated by a comma. Restrictions on the templates applied in the same manner Umple allows extending and implementing interfaces and classes respectively. In other words, the keyword '*isA*' is used after the name of a template parameter followed by the name of interfaces or a class. If there are more than one interface or one interface and one class they are separated by the symbol '&'.

When clients use traits, they must bind types to their parameters and also types must satisfy their restrictions. Values are bound through the symbol '=', in the manner values are generally assigned to attributes. The bindings are performed inside a pair of angle brackets, each one separated by a comma. Therefore, when a client uses a trait with template parameters, they need to extend the normal way of using traits by having angle brackets appearing after the name of traits.

The **example 1** below shows how template parameters can be defined and used. It depicts a trait called T1 (line 4) with one template parameter named TP. The template parameter is restricted to implement the interface I1, defined in line 8. The restriction is applied to make sure a correct type will be bound to the template parameter. Trait T1 has a required method named method2() with a return and a parameter of type TP (which is a template parameter). Furthermore, it has a provided method named method3() with a parameter of type TP.

Class C1 defined in line 11 uses trait T1 and assigns class C1 as a binding type to TP. Since class C1 has already implemented interface I1 (line 12), the type is acceptable. Class C1 needs to implement the required method of trait T1, but it must be performed based on the type assigned to the

template parameter TP. Therefore, class C1 implements method2() with the correct data type (which is C1) for the parameter and return types. Line 15 shows the exact signature of the implemented method. In the same manner, the provided method obtained is also adapted based on the binding value. The same process is adopted for class C2, but it binds C2 to the parameter TP and so the required method method2() needs to be implemented with type C2. The provided method obtained is based on type C2. There is a method called method1() for classes C1 and C2 that implement interface I1. They have the same signature but different implementations.

Since a trait can use other traits, a trait with template parameters can use other traits with template parameters. Therefore, traits can bind a template parameter to another template parameter to achieve better flexibility. This is performed in the same manner a type is bound to a template parameter. Furthermore, template parameters are modulated for each trait so it is possible for a trait with template parameters to use other traits with the same names for their template parameters.

The **example 2** below shows how one template parameter is bound to another one. Trait T1 has the template parameter TP used to define the type of parameter for the provided method method2(). Trait T2 has the template parameter TP used to define the type of parameter for the provided method method3(). It also uses trait T1 and binds its own template parameter to traits T1's template parameter (line 9). Note that trait T2 can also bind any other types to the template parameter TP, if it is required. Finally, class C1 uses trait T2 and binds String to the template parameter TP (line 13)

Umple introduces a special syntax to allow having template parameters that can be substituted in code blocks. For template parameters to operate on code in code blocks, they need to be encompassed within a pair of the symbol '#'. This ensures that the dedicated Umple scanner (i.e. not a full code parser) will be able to detect strings matching the template parameters correctly and replace them with binding values.

The **example 3** shows how a template parameter is used in the body of a method. As seen, trait T1 has a template parameter named TP (line 4). The provided method method2() needs to return a string which is a combination of calling the required method method1() and a method named process() from the template parameter TP. In the body of the provided method method2(), an instance of the template parameter needs to be created so as to call the method process(). This is achieved by having the name of template parameter encompassed by '#' in places that require types (line 7). The rest needs to follow the syntax of the language

used to implement the code blocks, in this case, Java. Class C2 uses trait T1 and binds class C1 to the template parameter TP (line 15). It also implements the required method method1() in order to able to use the trait.

When exploring the following examples in UmpleOnline, you can use the Options menu to control what is visible, or you can use control-R to flip back and forth between showing the diagram with the original traits, vs. the diagram collapsed into the classes to be compiled; or you can use control-M to show/hide methods.

Example 1

view plain

```
01. /*
02.   Example 1: showing how template parameters in
03.   traits are defined and used.
04.
05.   To see different diagram views in UmpleOnline:
06.     Use control-g for auto-layout
07.       (if not already showing)
08.     Use control-r to switch between trait view and
09.        plain classes resulting from applying traits
10.     Use control-m to show/hide methods
11. */
12. trait T1< TP isA I1 > {
13.   abstract TP method2(TP data);
14.   String method3(TP data){ /*implementation*/ }
15. }
16. interface I1{
17.   void method1();
18. }
19. class C1{
20.   isA I1;
21.   isA T1<TP = C1>;
22.   void method1(){/*implementation*/}
23.   C1 method2(C1 data){ /*implementation*/ }
24. }
25. class C2{
26.   isA I1;
27.   isA T1< TP = C2 >;
28.   void method1(){/*implementation*/}
29.   C2 method2(C2 data){ /*implementation*/ }
```

```
}
// @@@skipcompile issue needs raising missing return statement

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 2

view plain

```
/*
   Example 2: showing how one template parameter
   in a trait is bound to another one.
*/
trait T1<TP>{
  void method1();
  void method2(TP data) Java {/*implementation*/}
  void method2(TP data) Python {''' implementation '''}
}
trait T2<TP>{
  isA T1< TP = TP >;
  void method3(TP Data) Java {/*implementation*/}
  void method3(TP Data) Python {'''implementation'''}
}
class C1{
  isA T2< TP = String >;
  void method1() Java {/*implementation*/}
  void method1() Python {'''implementation'''}
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 3

view plain

```
/*
   Example 3: showing how template parameters in
   traits are used in code blocks.
*/
```

```
05. trait T1 <TP>{
06.   String method1();
07.   String method2(){
08.     #TP# instance = new #TP#();
09.     return method1() +":"+instance.process();
10.   }
11. }
12. class C1{
13.   String process(){/*implementation*/}
14. }
15. class C2{
16.   isA T1< TP = C1 >;
17.   String method1(){/*implementation*/ }
18. }
19. // @@@skipcompile issue needs raising as Java
20. // compile fails missing return statement
21.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

traitParameters : < [[traitFullParameters]]
(, [[traitFullParameters]]
)* >

traitFullParameters : [~parameter]
([[traitParametersInterface]]
)?
(= [~defaultType]
)?

traitParametersInterface- : isA [~tInterface](& [~tInterface])*

Associations in Traits

An association is a useful mechanism to specify relationships among instances of classifiers. An association implies the presence of certain variables and provided methods in both associated classifiers.

Associations in traits are defined in the same way they are defined for classes. Traits can make associations with interfaces, classes, and template parameters. If one end of the association is a template parameter, the binding type must be checked to make sure it is compatible with the type of the association. For example, if a trait has a bidirectional association with a template parameter, the binding value cannot be an interface and it must be a class. This is an extra constraint applied to template parameters.

When an association is defined, APIs (set of methods) are generated in the class to allow such actions as adding, deleting and querying links of associations. Traits may use these APIs inside provided methods to achieve their needed implementation. The first example below depicts a simple version of the observer pattern implemented based on traits. As can be seen in line 7, the concept of the subject in the observer pattern has been implemented as trait Subject, which gets its observer as a template parameter. A direct association has been defined in trait Subject (Line 8), which has a multiplicity of zero or one on the Subject side and zero or many on the Observer side. This association lets each subject have many observers and it also applies the case in which observers do not need to know the subject. The trait has encapsulated the association between the subject and observers and then applies it to proper elements when it is used by a client.

As each subject must have a notification mechanism to let observers know about changes, there is a provided method notifyObservers() for this. This method obtains access to all observers through the association. Two classes Dashboard and Sensor play the roles of observer and subject. Class Dashboard has a method named update(Sensor) (line 2) used by the future subject to update it. Class Sensor obtains the feature of being a subject through using trait Subject and binding Dashboard to parameter Observer

The second example below is a concrete example of the use of traits to incorporate functionality (in this case dated addresses) that is needed in otherwise unrelated classes (Persons and Businesses). The third examples shows the use of traits that have associations along with other features of Umple.

When exploring the following examples in UmpleOnline, you can use the Options menu to control what is visible, or you can use control-R to flip back and forth between showing the diagram with the original traits, vs. the diagram collapsed into the classes to be compiled; or you can use control-M to show/hide methods.

Example of the Observable pattern using traits with associations

view plain

```
01. /*
02.   Example 1: implementing observable pattern
03.   with traits and associations.
04.
05.   To see different diagram views in UmpleOnline:
06.     Use control-g for auto-layout
07.       (if not already showing)
08.     Use control-r to switch between trait view and
09.        plain classes resulting from applying traits
10.     Use control-m to show/hide methods
11. */
12. class Dashboard{
13.   void update (Sensor sensor) Java { /* code */ }
14.   void update (Sensor sensor) Python { ''' code ''' }
15. }
16. class Sensor{
17.   isA Subject< Observer = Dashboard >;
18. }
19. trait Subject <Observer>{
20.   0..1  -> * Observer;
21.   void notifyObservers() Java { /* code */ }
22.   void notifyObservers() Python { ''' code ''' }
23. }
24.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example showing traits with associations to manage addresses

view plain

```
01. class Address { street; city; postalCode; country; }
02.
```

```
03. trait EntityWithValidityPeriod {
04.   Date startDate; Date endDate;
05. }
06.
07. trait PurposedEntity {
08.   purpose; // e.g. home, work, mobile
09. }
10.
11. class DatedAddress {
12.    isA EntityWithValidityPeriod, PurposedEntity, Address;
13. }
14.
15. class DatedPhoneNumber {
16.   isA EntityWithValidityPeriod, PurposedEntity;
17.   number;
18. }
19.
20. trait LocatableEntity {
21.   0..1 -> 1..* DatedAddress;
22.   0..1 -> 1..* DatedPhoneNumber;
23. }
24.
25. class Person {  isA LocatableEntity;}
26. class Business {  isA LocatableEntity;}
27.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example that also includes state machines and mixsets

```
view plain

01. interface I1 {
02.   void m1();
03. }
04.
05. trait SuperT {
06.   isA I1;
07.   mixset f5 {e;}
08.   void m1() {
09.     System.out.println("impl of m1");
10.   }
11. }
```

```
12.
13.  trait T1 {
14.    isA SuperT;
15.    Integer c;
16.    Integer d;
17.    mixset f6 {[c > d]}
18.    after setC {
19.       System.out.println("SetC has happened from T1");
20.    }
21.    mixset f1 {1 -- 1..* C4;}
22.  }
23.
24.  trait T2 {
25.    after setC {
26.      System.out.println("setC has happened from T2");
27.    }
28.  }
29.
30.  trait T3 {
31.    void m2() {
32.      System.out.println("M2 is executing");
33.    }
34.    sm1 {
35.      s1 {
36.        e -> s2;
37.      }
38.      s2 {
39.      }
40.    }
41.  }
42.
43.  trait T4 {
44.    isA T3;
45.  }
46.
47.  class C1 {
48.    isA T1, I1;
49.    mixset f1 {1 -- * C3;}
50.  }
51.
52.  mixset f5 class C2 {
53.    mixset f4 {1 -- 1..* C3;}
54.    isA C1, T2;
55.    void m1() {
56.      System.out.println("Overriding impl of m1");
```

```
57.   }
58. }
59.
60. class C3 {
61. }
62.
63. mixset f4 class C4 {
64.   isA T4;
65.   // Commented code bellow will cause an error when mixset f4 is
66.   // This because inline mixsets does not support apsect.
67.   //
    mixset f5 after custom m2() {System.out.println("m2 occurred");
68.   mixset f5 { after custom m2() {System.out.println("m2 occurr
69.   after e1 {System.out.println("event e1 occurred");}
70. }
71.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Required Interfaces of Traits

A required interface is an interface that a trait declares that it (or its clients) implement using the isA clause. A required interface groups together several required methods. The following two paragraphs explain this in more depth.

Required functionality of classic traits is defined in terms of required methods. However, there are shortcomings to just listing the required methods. The first shortcoming is that there is no way to reuse a *list* of required methods. For example, consider a case in which there are several traits that happen to have the same set of required methods but different provided methods. In this case, there is duplication due to repeated listing of the same methods. Furthermore, if there are several traits that must always have the same list of required methods, an inconsistency could be introduced in the design by changing just one of them and not all.

A required interface also allows putting a restriction on a trait's clients that specifies the interfaces they must implement. Such a restriction ensures that traits are not used in clients that just happen to have methods with the same signature.

So using required interfaces, traits can either put extra restrictions on clients (i.e. interfaces the clients must implement) or just manage their required methods in a more modular and reusable way. Traits may use already-existing interfaces or new interfaces may be written to accomplish the desired modularization. Furthermore, developers can create a hierarchy of interfaces to optimize the reusability.

Traits define their required interfaces through the keyword '*isA*' followed by the name of interfaces and a semi-colon. When a class uses traits, it needs to implement the required interfaces of those traits. If a trait uses other traits with required interfaces, those required interfaces are added to the set of required interfaces of the trait, and final clients are required to implement all of those required interfaces.

The example 1 below shows how required interfaces are used and applied. There is a hierarchical design for required methods in terms of interfaces, making it reusable and consistent. Traits T1 and T2 have the same required interface (lines 14 and 18) and if there is a modification in the required interface it will be applied to both. Classes C1 and C2 have to implement interfaces I1 (Line 29) and I2 (line 32) to be able to use trait T1 and T2.

When exploring the following example in UmpleOnline, you can use the

Options menu to control what is visible, or you can use control-R to flip back and forth between showing the diagram with the original traits, vs. the diagram collapsed into the classes to be compiled; or you can use control-M to show/hide methods.

Example

view plain

```
01. /*
02.   Example 1: showing how required interfaces
03.   in traits are defined and used.
04.
05.   To see different diagram views in UmpleOnline:
06.     Use control-g for auto-layout
07.       (if not already showing)
08.     Use control-r to switch between trait view and
09.        plain classes resulting from applying traits
10.     Use control-m to show/hide methods
11. */
12. interface I1{
13.   void method1();
14.   double method2();
15. }
16. interface I2 {
17.   isA I1;
18.   Boolean method3();
19. }
20. trait T1{
21.   isA I1;
22.   Float method3(){/*implementation*/ }
23. }
24. trait T2{
25.   isA I1;
26.   Float method4(){/*implementation*/ }
27. }
28. trait T3{
29.   isA T1,T2;
30. }
31. class C3{
32.   void method1(){/*implementation*/ }
33.   Double method2(){/*implementation*/ }
34. }
35. class C1{
36.   isA C3, I1, T1;
```

```
37. }
38. class C2 {
39.   isA C3, I2, T3;
40.   Boolean method3(){/*implementation*/ }
41. }
42. // @@@skipcompile Java code not compilable
43.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

State Machines in Traits

A trait can have zero or many state machines, each with a unique name. The definition of state machines in traits follows the same rules and constraints that exist for them in classes.

The first example shows a trait called T1 (lines 4-13) with two state machines sm1 (lines 5-8) and sm2 (lines 9-12). Use of trait T1 in class C1 results in class C1 having those two state machines as native state machines. In general, if a class already has state machines with completely distinct names to those being introduced via traits, the introduced state machines are just 'flattened' into the class, i.e. they are treated as though they were coded directly in the class.

Introduced machines that have names duplicating existing state machine names are composed (merged) with the existing machines, and the resulting composed machines are flattened into the class. The same concept is applied when traits are used by other traits. The second example below expresses the same model designed in example 1, in which class C1 has two state machines, but it uses trait T2 (line 18) to obtain the same state machine. Trait T2 provides state machine sm2 from its own definition (lines 12-15) and state machine sm1 from trait T1 (line 11). This use of traits by other traits allows building more complex traits (and hence more complex state machines) from simpler ones.

State machines in traits are considered as *provided* functionality. More concretely, any event in a state machine is considered as a *provided method*, and so can satisfy the required methods of used traits. State machines are supposed to encapsulate their own actions and guards so they can be reused as a piece of functionality. For example, a guard can be defined as a reference to an attribute in the trait or to a parameter of the event, so when the state machine is reused the attribute and parameter are reused as well. The same perspective is true for actions. In real world cases, state machines may need to obtain those conditions and actions from the context (e.g. clients) in which they are reused. State machines hence require those conditions and actions to behave correctly. This requirement is in alignment with the notion of required methods of traits. Therefore, required actions and guards of state machines can be expressed as required methods of traits. Indeed, if events of state machines are considered as provided methods and their guards and actions are considered as required methods, then state machines can define functional behavior and therefore the required behavior of them can again be satisfied by other state machines in the clients. It should be noted that if guards, actions, and activities of state machines are defined as methods (not inline code blocks directly in the state machine), they are

not anymore required methods. They will be treated as provided methods, and traits' rules related to provided methods will be applied to them

The third example below shows how required methods are satisfied by events of state machines. As seen, trait T1 has two required methods, m1(String) and m2(), called in actions of transitions e1(string) and e2 in states s1 and s2 of state machine sm1 respectively. Class C1 uses trait T1 and must satisfy those required methods. Class C1 does not have any concrete method to satisfy the required methods, but it has state machine sm2. State machine sm2 has two event m1(String) and m2 which satisfy the required methods of trait T1. If state machines are used to satisfy the required methods, there is a limitation in return types of required methods. All required methods must have Boolean as their return types, otherwise, events cannot satisfy them. The reason for this limitation is that all event automatically obtain Boolean as their return types by the Umple compiler.

Like the way template parameters defined in traits are used for required and provided methods, they can also be used in collaboration with state machines. Template parameters can be used to define the types of parameters for events. They can also be used in code blocks related to actions and activities. The example 4 illustrates how template parameters can be used with events of state machines. State machine sm in trait T1 has two events. Event e1 has a parameter of type TP (line 6) and event e2 has two parameters with types TP and String (line 7). Class C2 uses trait T1 (line 12) and binds class C1 to the template parameters TP.

The final example below shows how two classes can reuse the same state machine using a trait. This functionality is equivalent to that achieved using standalone state machines.

When exploring the following examples in UmpleOnline, you can use the Options menu to control what is visible, or you can use control-R to flip back and forth between showing the diagram with the original traits, vs. the diagram collapsed into the classes to be compiled; or you can use control-M to show/hide methods; or control-S and control-G to alternate between state and class diagrams.

When exploring the following examples in UmpleOnline, you can use the Options menu to control what is visible, or you can use control-R to flip back and forth between showing the diagram with the original traits, vs. the diagram collapsed into the classes to be compiled; or you can use control-M to show/hide methods.

Example 1: Building a class with two state machines

```
view plain

/*
  Example 1: showing how state machines are
  defined and used in traits.

  To see different diagram views in UmpleOnline:
    Use control-g for auto-layout as a class diagram
    Use control-r to switch between trait view and
       plain classes resulting from applying traits
    Use control-m to show/hide methods
    Use control-s to show the resulting state diagram
*/
trait T1 {
  sm1{
  s0 {e1-> s1;}
  s1 {e0-> s0;}
}
sm2{
  s0 {e1-> s1;}
  s1 {e0-> s0;}
}
}
class C1 {
  isA T1;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 2: Events as provided methods

```
view plain

/*
    Example 2: showing how state machines
    are defined through different traits.
*/
trait T1 {
  sm1{
  s0 {e1-> s1;}
  s1 {e0-> s0;}
  }
```

```
10. }
11. trait T2 {
12.   isA T1;
13.   sm2{
14.   s0 {e1-> s1;}
15.   s1 {e0-> s0;}
16.   }
17. }
18. class C1 {
19.   isA T2;
20. }
21.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 3: Satisfying required methods

view plain

```
01. /*
02.    Example 3:showing how required methods
03.    of traits are satisfied by events of
04.    state machines.
05. */
06. trait T1{
07.   Boolean m1(String input);
08.   Boolean m2();
09.   sm1{
10.     s1{ e1(String data) -> /{ m1(data); } s2;   }
11.     s2{ e2 -> /{ m2(); } s1; }
12.   }
13. }
14. class C1{
15.   isA T1;
16.   sm2{
17.     s1{ m1(String str) -> s2;}
18.     s2{ m2 ->  s1;}
19.   }
20. }
21.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 4: Use of template parameters

view plain

```
/*
   Example 4: showing how template parameters
   in traits are used inside state machines.
*/
trait T1<TP>{
  sm{
  s1{ e1(TP p1)-> s2; }
    s2{ e2(String p1, TP p2) -> s1; }
  }
}
class C1{/*implementation*/}
class C2{
  isA T1<TP=C1>;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 5: Equivalence to standalone state machines

view plain

```
// Class Diagram illustrating use of a trait
// to incorporate the same state machine in multiple
// classes. This is an alternative to using
// standalone state machines (see an equivalent model
// in the state machine section of the manual)
/*
  To see different diagram views in UmpleOnline:
    Use control-g for auto-layout as a class diagram
    Use control-r to switch between trait view and
       plain classes resulting from applying traits
    Use control-s to show the resulting state diagram
*/

class MotorController {
  isA DeviceWithStatus;
```

```
16. }
17.
18. class BrakeController {
19.   isA DeviceWithStatus;
20. }
21.
22. trait DeviceWithStatus {
23.   deviceStatus {
24.     inactive {
25.       activate -> booting;
26.     }
27.     booting {
28.       completedStartupChecks -> active;
29.       startupCriticalErrorDetected -> outOfOrder;
30.     }
31.     active {
32.       runtimeCriticalErrorDetected -> outOfOrder;
33.       deactivate -> shuttingDown;
34.     }
35.     shuttingDown {
36.       shutdownComplete -> inactive;
37.     }
38.     outOfOrder {
39.       repaired -> inactive;
40.     }
41.   }
42. }
43. // @@@skipcppcompile
44.
45.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Method Operators in Traits

Operators are capabilities that allow resolving the conflicts and also managing the granularity of traits. Operators are applied to traits when traits are used by clients. Clients can apply more than one operator to a specific trait, but those operators need to be compatible with each other. There is no sequence in the way operators are applied. Operators are defined inside angle brackets after the name of traits and can be mixed with binding types to template parameters. The general structure by which a client uses operators is as follows:

isA TName < Operator1, Operator2, …, Operatorn >

TName specifies the name of a trait to which each Operatori,i=1..n is applied. There is no limitation on the number of operations that can be applied to a trait. Two operators with the same functionality can only be applied to different elements. Errors resulting from violations of this are detected and reported to the modeler. Note that all processes related to the flattening and composition algorithm are performed after applying the operators.

If traits have template parameters and those template parameters have been used as types in provided methods (or event of state machines), their types do not affect the signature of provided methods referred to by operators. In other words, types of template parameters are applied to traits after operators are applied. The identification factor for selecting a provided method is its signature. However, the return type of provided methods is not used for identification because the name and list of types of parameters can uniquely differentiate each provided method from others. When parameters of provided methods are specified, there is no need to define the name of parameters

Removing/keeping provided methods

This operator allows removing or keeping provided methods of a trait. The syntax for this operator is as follows:

(+|-) methodName(argumentTypes)

The symbol – indicates removing while the symbol + indicates keeping. The symbol must precede the signature of the provided method. When the symbol – is applied to a provided method of a trait, it removes the method from the set of provided methods. However, when the symbol + is applied to a provided method of a trait, the provided method is kept and the rest are removed. Please consider that multiple keeping operators can be used together to keep several methods and throw away the rest.

The example 1 below shows how class C1 removes the provided methods method2() and method3() (line 12) while class C2 keeps the provided

method method5(), coming from trait T1 (line 16). As seen, class C1 obtains two provided methods method4() and method5() in addition to the method1() which satisfies the required methods of trait T1. Class C2 just has the provided method method5() in addition to method1().

Renaming (Aliasing)
The renaming operator allows changing the name of provided methods and also their visibilities. This operator can also be mixed partially with the keeping operator to provide better flexibility. The operator does not allow changing the types of parameters or number of parameters. The reason is that the provided methods might be used by other provided methods and so any change regarding types and numbers can break traits. The current implementation of the operator does not support renaming recursive methods. The syntax for this operator is as follows:

(+) methodName(argumentTypes) as newName

The example 2 shows this operator in action. Class C1 uses trait T1 (line 12) and renames its provided method method1() to function2. There is no need to specify parentheses for the new name. Class C2 uses trait T1 (line 16) and while it renames provided method3() to function3, it also changes the visibility of the provided method to be private. Finally, class C3 uses trait T1 and renames the provided method method5(Integer) to function5. However, it also forces other provided methods of trait T1 to be removed. This feature can be really useful if there is a utility trait and we are just interested in a provided method with the name that suits our domain.

When exploring the following examples in UmpleOnline, you can use the Options menu to control what is visible, or you can use control-R to flip back and forth between showing the diagram with the original traits, vs. the diagram collapsed into the classes to be compiled; or you can use control-M to show/hide methods.

Example

view plain

```
01. /*
02.     Example 1: showing how the operator
03.     "Removing/keeping provided methods" works.
04. */
05. trait T1{
06.   abstract method1();
07.   void method2() Java {/*implementation*/}
08.   void method3() Java {/*implementation*/}
```

```
09.   void method4() Java {/*implementation*/}
10.   void method5() Java {/*implementation*/}
11.
12.   void method1() Python { '''implementation''' }
13.   void method2() Python { '''implementation''' }
14.   void method3() Python { '''implementation''' }
15.   void method4() Python { '''implementation''' }
16.   void method5() Python { '''implementation''' }
17. }
18. class C1{
19.   isA T1<-method2() , -method3()>;
20.   void method1() Java {/*implementation for C1*/}
21.   void method1() Python {'''implementation for C1*'''}
22. }
23. class C2{
24.   isA T1<+method5()>;
25.   void method1() Java {/*implementation for C2*/}
26.   void method1() Python {'''implementation for C2'''}
27. }
28.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. /*
02.    Example 2: showing how the operator
03.    "Renaming (Aliasing) provided methods" works.
04. */
05. trait T1{
06.   abstract method1();
07.   void method2() Java {/*implementation*/}
08.   void method3() Java {/*implementation*/}
09.   void method4() Java {/*implementation*/}
10.   void method5(Integer data) Java {/* implementation*/}
11.
12.   void method2() Python {'''implementation'''}
13.   void method3() Python {'''implementation'''}
14.   void method4() Python {'''implementation'''}
15.   void method5(Integer data) Python {'''implementation'''}
16. }
```

```
17. class C1{
18.   isA T1< method2() as function2 >;
19.   void method1() Java {/*implementation related to C1*/}
20.   void method1() Python {'''implementation related to C1'''}
21. }
22. class C2{
23.   isA T1< method3() as private function3 >;
24.   void method1() Java {/*implementation related to C2*/}
25.   void method1() Python {'''implementation related to C2'''}
26. }
27. class C3{
28.   isA T1< +method5(Integer) as function5 >;
29.   void method1() Java {/*implementation related to C3*/}
30.   void method1() Python {'''implementation related to C3'''}
31. }
32.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

iEFunction : [=modifier:+|-] [~methodName] [[iEParameterList]]

functionAliasName : [=modifier:+]?
  (
  ([~smName]
    | [!smPattern:\d+|[**]]
) .
)? [~methodName] [[iEParameterList]] as [[IEVisibilityAlias]]

State Machine Operators in Traits

As with operators on methods, certain operators can be applied to traits' state machines when they are used in clients . These provide mechanisms to improve flexibility, assign state machines to specific states, and resolve conflicts caused by name collisions. These operators follow the same structure defined for operators on methods.

Changing the name of a state machine

This operator is used to change the name of a state machine when it is to be reused by a client. This operator can also be mixed partially with the keeping operator to provide better flexibility. The syntax for this operator is as follow:

(+) stateMachineName as newName

When a state machine with a given name is specified by the renaming operator, it must be available in the trait being operated on, either directly in the trait or another trait used by the trait. The example 1 shows how names of state machines in trait T1 can be changed to mach1 and mach2. As seen, in example 1 both state machines that are available directly in trait T1 have been renamed. The example 2 shows how class C1 uses the operator to rename the state machine sm1 in trait T1 and also automatically remove other state machines, which are just state machine sm2 in this case.

There are two main scenarios for this operator. The first is merging: when a client already has a state machine and will obtain another state machine coming from the used trait. The two machines might have some of the same states but different functionality. The client wants to have all functionality merged in one state machine (described as we progress). In this case, the operator is used to change the name of the incoming state machine from the trait, to match the name of the existing state machine. The result is that the new functionality will be merged into the existing state machine. The second scenario is for avoiding conflicts. This occurs when a client has an existing state machine and wants to incorporate another one with different functionality, but that happens to have the same name. In this case, the client can change the name of the incoming state machine to be different from the existing state machine. This second scenario can be needed in various conflict-resolution scenarios

Changing the Name of a State

This operator changes the name of a state inside a specific state machine. The operator covers both simple and composite states. The syntax used for this purpose is as follows:

stateMachineName.stateName.....stateName as newName

The state to be renamed is specified based on a series of names separated by dots, starting with the name of the state machine. If the state is a simple or composite state at the top level of a hierarchical state machine, then it comes directly after the name of the state machine. However, if it is deeper in the hierarchy, the chain of parents must also be specified. The last name in the series is always the name of the state to be renamed. In example 3, trait T1 has a state machine with a composite state named s0 (line 6). Composite state s0 has two internal states s11 and s12. Class C1 uses trait T1 and changes the name of state s0 to state0 and the name of s11 to state11 (line 15). In order to specify the state s0, it is preceded by the name of a state machine, which is sm. For state s11, the name of the state machine, the composite state, and the region name precede it. In Umple, the name of the single region inside a composite state is set automatically to the name of the composite state. The operator applies also the same rule when it changes a composite state.

The first scenario for this operator is to change the vocabulary used for the names of states. This adds flexibility when a trait is specified in a generalized context and there is a need to adapt names so as to be more domain-specific. E.g. 'tripEnded' in a general transportation state machine becomes 'landed' in an adaptation to the airline domain, or 'docked' in an adaptation to the water transport domain. The second scenario is when two states need to be merged, but they have different names. By changing the name of one so it matches the other, then the algorithm knows to merge them. The third scenario occurs when there are two states in a state machine to be composed, but we want to keep those two states separate and prevent merging.

Changing the name of regions

This operator allows renaming a specified region, an orthogonal part of a composite state or state machine. It is just like the operator used for changing the name of states. The difference is that the last name in the sequential series of names (separated by dots) is the name of a region to be changed. Since names of regions are set automatically by the Umple compiler and they are equal to the names of their initial states, renaming the name of a region must also be applied to its initial state. This is performed automatically by the operator. The applications for this operator are like those for changing the name of states. In particular, this operator is used when several regions are supposed to be merged or kept separate.

Change the name of events

This operator is used to change the name of events that will trigger transitions in state machines. The syntax used for this operator is as follows:

(* | stateMachineName).eventName(argumentTypes) as new_Name

Through this operator, it is possible to rename an event related to a specific state machine or all state machines in a trait. For the first case, the modeler specifies the name of the state machine (stateMachineName). For the second case, an asterisk (*) is specified. The event name (eventName) must end with a pair of parentheses including any needed argument types. This operator does not allow changing the argument types because that would break the implementation of the event method. The operator is used mostly to change the event names based on a new domain's requirements. It can also be used to keep an event from being overwritten by the client's state machine and vice versa.

In example 4, trait T1 has two state machines sm1 and sm2. These state machines have common events named e1(Integer) and e0(). Class C1 want to use trait T1 with some changes in the name of events. It is required to rename all event names e0() to event0() and just change the event name e1(Integer) to event1() in state machine s1. Line 15 depicts how class C1 achieves it. Since the change on event e0() is going to happen in both the state machines, the symbol * has been used. However, the name of state machine sm1 was used for the event e1(Integer) because we do not want to have it changed in state machine sm2.

When exploring the following examples in UmpleOnline, you can use the Options menu to control what is visible, or you can use control-R to flip back and forth between showing the diagram with the original traits, vs. the diagram collapsed into the classes to be compiled; or you can use control-M to show/hide methods; or control-S and control-G to alternate between state and class diagrams.

Example

view plain

```
01. /*
02.   Example 1: showing how the operator
03.   "Changing the name of a state machine" works.
04.
05.   To see different diagram views in UmpleOnline:
06.     Use control-g for auto-layout as a class diagram
07.     Use control-r to switch between trait view and
08.       plain classes resulting from applying traits
09.     Use control-m to show/hide methods
10.     Use control-s to show the resulting state diagram
11.
```

```
*/
trait T1 {
  sm1{
    s0 {e1-> s1;}
    s1 {e0-> s0;}
  }
  sm2{
    s0 {e1-> s1;}
    s1 {e0-> s0;}
  }
}
class C1 {
 isA T1<sm1 as mach1, sm2 as mach2>;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
/*
   Example 2: showing how the operator
   "Changing the name of a state machine" works.
*/
trait T1 {
  sm1{
    s0 {e1-> s1;}
    s1 {e0-> s0;}
  }
  sm2{
    s0 {e1-> s1;}
    s1 {e0-> s0;}
  }
}
class C1 {
 isA T1<+sm1 as mach1>;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
/*
   Example 3: showing how the operator
   "Changing the name of a state" works.
*/
trait T1 {
  sm{
    s0{
      e1-> s1;
      s11{ e12-> s12; }
      s12{ e11-> s11; }
    }
    s1{ e0-> s1;  }
  }
}
class C1 {
  isA T1<sm.s0 as state0, sm.s0.s0.s11 as state11>;
}
// @@@skipcppcompile


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
/*
   Example 4: showing how the operator
   "Change the name of events" works.
*/
trait T1 {
  sm1{
    s0 {e1(Integer index)-> s1;}
  s1 {e0-> s0;}
}
sm2{
```

```
    t0 {e1(Integer index)-> t1;}
  t1 {e0-> t0;}
}
}
class C1 {
  isA T1<sm1.e1(Integer) as event1,
    *.e0() as event0>;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
functionAliasName : [=modifier:+]?
  (
  ( [~smName]
    | [!smPattern:\d+|[**]]
  ) .
  )? [~methodName] [[iEParameterList]] as [[IEVisibilityAlias]]
```

Selecting Operators

As with operators on methods, certain operators can be applied to traits' state machines when they are used in clients. These provide mechanisms to improve flexibility, assign state machines to specific states, and resolve conflicts caused by name collisions. These operators follow the same structure defined for operators on methods.

Removing/keeping a state machine
This operator is used to remove or keep a state machine when a client uses a trait. In the removing mode, specified by the minus symbol '-', the indicated state machine is ignored and is not included in the client. In the keeping mode, specified by '+', only the indicated state machine is kept and the others are ignored. This operator can be used to keep the client free of unneeded detail or conflict. The syntax used for this operator is as follows:

(-|+) stateMachineName

In example 5, trait T1 has three state machines sm1, sm2, and sm3. Classes C1 requires state machine sm2 and sm3 while class C2 requires just state machine sm2. Class C1 achieves this through removing state machine sm1 from trait T1 (line 16). Class C2 obtains its required state machine through keeping just state machine sm2 (line 19). Classes C2 could also achieve the same result through removing sm1 and sm2. Using the keeping operator is more convenient when there are several state machines and modelers need just one them.

Removing/keeping a state
This operator is used to remove or keep a simple or composite state when using a state machine in a trait. The syntax for this operator is as follows:

(-|+) stateMachineName.stateName.....stateName

This works much like removing/keeping a state machine, using a minus sign for removing and a plus for keeping. The symbols are followed by the name of the state. In the removing mode, this operation will delete all incoming and outgoing transitions of the state as well. In the keeping mode, the specified state will be kept and the remaining states will be removed. This also includes removing all transitions from other states to the specified state. This mode can be applied to the initial states, but if it is applied to other states, the initial state will not be removed. The operator is helpful for cases in which base state machines do not need the functionality of that specific state, or have the same state and do not want to merge it with the one coming from the reused trait. Another use is when clients use more than one trait and those traits have common state machines and states. These common states might have different

functionality and clients might want to keep one version.

In example 6, trait T1 has state machine sm1 (line5). Class C1 uses trait T1 and removes state s2 from the state machine sm1 (line 25). As seen, in addition to the state s2, all outgoing transitions (named e2()) from states s0, s1, and s3 have been removed. The operator also removed incoming transition e3() from state s2 to state s3. Class C2 uses trait T1 and requires just state s1 of state machine sm1. As seen, all other states except s0 have been removed. The reason is that state s0 is the initial state of state machine sm1 and removing it results in non-reachable states. The Umple compiler does not allow this situation. Furthermore, all transitions coming from other states to s1 have been removed (in this case, there is none) except transitions coming from the initial state s0. The operator also removes the outgoing transitions e2() and e3() of state s1 to other states s2 and s3 except the ones going to the initial state (in this case, there is none).

Removing/keeping a region
This operator is used to remove or keep a region of a state machine. The syntax used for this operator is exactly like the one defined for removing or keeping a state, except the last name in the dot-separated chain specifies the name of the region. This operator is utilized in cases similar to those explained for removing a state. It can also be used to make a composite state a simple state by reducing the number of regions to zero.

In example 7, two classes C1 and C2 uses trait T1 and manipulate its state machine's regions. State machine sm has a composite state s1 with three regions r1, r2, and r3. Class C1 removes region r1 from the composite state s1 (line 23) while class C2 keeps region r2 (line 26). As seen in both cases, the region name appears after the name of state machine sm and the state s1. We can see since region r1 has been removed in class C1, the incoming transition e2() has also been removed automatically. In class C2, all outgoing transition from region r2 to other regions, including e2() and e4(), have been removed automatically in addition to region r1 and r3.

Removing/keeping a transition
This operator is used to remove or keep a transition from a state machine. The syntax for this operator is as follows:

(-|+) stateMachineName.stateName.....stateName.
((eventName(argumentTypes) ([guard])?) | [guard?])

A minus symbol is followed by a transition that needs to be removed. A plus symbol is used to pick a transition to keep. A transition is defined by specifying the name of the state machine, states (including regions for

nested states), event, and guard. The symbol '?' is not part of the operator and it is there to show which elements are optional. The name of the state machine and states are mandatory. The event name (along with its arguments) depends on the type of transition.

If the transition is 'auto' (immediately taken on entry to the state or upon completion of a do activity) there is no need to specify it, otherwise, it must be specified. If a transition has a guard, it must be specified using the same syntax used when specifying transitions (inside square brackets). However, if a transition is auto and unguarded, it must be defined with an empty guard "[]" and without any event name. Actions and destination states are not part of the definition for this operator because the above definition suffices to uniquely select any transition. The operator is utilized when base state machines do not need a specific transition coming from the used trait. Furthermore, base state machines might want to extend a transition of a state, but it might already have a transition matching the given specification.

In example 8, class C1 uses trait T1 (line 25) and keeps the transition with the event name e2(Integer) from the state machine sm and state s1. Other transitions related to the state s1, which are e3() and e4(), are removed. Since the transition does not have a guard, brackets are not required in the specification of the transition. Class C2 also uses trait T1 (line 28), but it removes the transition with the event name e4() and a guard on the variable cond from the state machine sm and state s1. This case shows how a transition with the event name and guard can be specified. Class C3 uses trait T2, but it removes an auto transition from state s2 with a guard on the variable cond. As seen, no name has been defined in the operator for the event and the guard is just defined inside brackets. Finally, class C4 use the trait T1 (line 34), but it removes an auto transition without a guard. As seen, an empty bracket after the name of the state is used to specify the transition.

Example

view plain

```
01. /*
02.    Example 5: showing how the operator
03.    "Removing/keeping a state machine" works.
04. */
05. trait T1 {
06.   sm1{
07.     s0 { e0-> s0;}
08.   }
```

```
09.  sm2{
10.     t0 { e0-> t0;}
11.   }
12.  sm3{
13.     w0 { e0-> w0;}
14.   }
15.  }
16.  class C1 {
17.   isA T1<-sm1>;
18.  }
19.  class C2 {
20.   isA T1<+sm2>;
21.  }
22.  // @@@skipcppcompile
23.
24.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

```
view plain

01.  /*
02.   Example 6: showing how the operator
03.   "Removing/keeping a state" works.
04.
05.   To see different diagram views in UmpleOnline:
06.     Use control-g for auto-layout as a class diagram
07.     Use control-r to switch between trait view and
08.        plain classes resulting from applying traits
09.     Use control-m to show/hide methods
10.     Use control-s to show the resulting state diagram
11.
12.  */
13.  trait T1 {
14.   sm1{
15.     s0 {
16.       e1-> s1;
17.       e2-> s2;
18.     }
19.     s1 {
20.       e2-> s2;
```

```
    e3-> s3;
   }
   s2 {
    e3-> s3;
    e2-> s2;
   }
   s3 {
    e0-> s0;
    e2-> s2;
   }
  }
}
class C1 {
  isA T1<-sm1.s2>;
}
class C2 {
  isA T1<+sm1.s1>;
}
// @@@skipcppcompile


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
/*
   Example 7: showing how the operator
   "Removing/keeping a region" works.
*/
trait T1{
  sm {
   s1{
    r1{ e1-> r11; }
    r11{}
    ||
    r2{
     e2-> r11;
     e3-> r21;
     e4-> r31;
    }
```

```
    r21{}
    ||
    r3{e5->r31; }
    r31{}
   }
  }
}
class C1{
  isA T1<-sm.s1.r1>;
}
class C2{
  isA T1<+sm.s1.r2>;
}
// @@@skipcompile issue needs raising as Java compile fails

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
/*
   Example 8: showing how the operator
   "Removing/keeping a transition" works.
*/
trait T1{
  internal Boolean cond;
  sm {
    s1{
      e2(Integer i)-> s2;
      e3[!cond]-> s3;
      e4[cond]-> s4;
    }
    s2{
      [cond] -> s3;
      [!cond]-> s4;
    }
    s3{
      -> s1;
    }
    s4{
     -> s1;
```

```
    }
  }
}
class C1{
  isA T1<+sm.s1.e2(Integer)>;
}
class C2{
  isA T1<-sm.s1.e4()[cond]>;
}
class C3{
  isA T1<-sm.s2.[cond]>;
}
class C4{
  isA T1<-sm.s3.[]>;
}
// @@@skipcppcompile


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
//| [[StateMachineTransitionAlias]]
StateMachineAliasName :
  ([=iEStateMachineModifier:+]
  )? [~smName]
  ([[StateNames]]
  )? as [~smDesName]
  ([[DesStateNames]]
  )?

//iEStateMachine : [=iEStateMachineModifier:+|-] [~smName]
[[StateNames]]?
iEStateMachine : [=modifier:+|-] [~smName] ( [[StateNames]] ( (
[[iEParameterList]] [[guardOption]]? )
    | ( . [[guardOption]] ) )? )?
```

Extending a State

This operator is used to assign a state machine to a specific state inside another state machine, hence turning that state into a composite state. The syntax used for this operator is as follows:

srcStateMachineName as desStateMachineName.stateName.....stateName
This operator involves two state machines. The srcStateMachineName is found in the trait, and the desStateMachineName is found in the client. The state in the client can be simle or composite. This operator provides a practical mechanism to incrementally compose a state machine from various parts. For example, simple 'on/off' pairs of states with events to toggle between are fairly common and can be injected easily into destination states using this operator. If a composited state is extended with this operator, it will trigger our composition algorithm which. Furthermore, the operator can be used to bring more than one state machine inside a state.

In example 9, class C1 has state machine sm with two states s1 and s2. Trait T1 has state machine sm1. Class C1 needs to have state machine sm1 activated when it is in state s2. Class C1 achieves this by specifying the source state machine and destination state when it uses trait T1 (line 15).

Example

view plain

```
01. /*
02.   Example 9: showing how the operator
03.   "Extending a state by adding a state machine
04.   to it" works.
05.
06.   To see different diagram views in UmpleOnline:
07.     Use control-g for auto-layout as a class diagram
08.     Use control-r to switch between trait view and
09.        plain classes resulting from applying traits
10.     Use control-m to show/hide methods
11.     Use control-s to show the resulting state diagram
12. */
13. trait T1{
14.   sm1{
15.     m1{
16.       t2-> m2;
17.     }
```

```
    m2{
      t1-> m1;
    }
  }
}
class C1{
  isA T1<sm1 as sm.s2>;
  sm{
    s1{
      e2-> s2;
    }
    s2{
      e1-> s1;
    }
  }
}
// @@@skipcppcompile


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Traits and Umple Mixins

In the same way Umple supports mixins to compose classes, traits can also be composed in this way. This means that a trait can be defined in several places or files and when they are used by clients, all elements defined in those separate places will be applied to clients.

The example 1 depicts two definitions for trait T1 (lines 4 and 8). Class C1 uses trait T1 and implement the required method method1() and also obtains two provided methods method2() and method3().

Example

view plain

```
01. /*
02.   Example 1: showing how traits are combined with
03.   Umple mixins.
04.
05.   To see different diagram views in UmpleOnline:
06.     Use control-q for auto-layout as a class diagram
07.     Use control-r to switch between trait view and
08.       plain classes resulting from applying traits
09.     Use control-m to show/hide methods
10. */
11. trait T1{
12.   void method1();
13.   void method2() Java {/*impl… */ }
14.   void method2() Python {'''impl… ''' }
15. }
16. trait T1{
17.   void method3() Python {'''impl… ''' }
18. }
19. class C1{
20.   isA T1;
21.   void method1() Python {'''impl… ''' }
22. }
23.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Flattening of Traits

A system modeled with traits can be transformed to a compatible model without traits. This process is called flattening. Traits are flattened to clients when they are used by clients, so it would also be beneficial to be able to instantly switch between a view with traits and a flattened view. This could help modelers to understand how traits are represented in any object-oriented target languages, as well as the implications of the traits to the generated system. Furthermore, flattened model is useful when it is required to understand what provided methods are available in each final client.

In order to achieve this in UmpleOnline, first the menu *OPTIONS* must be selected. Then, sub menus *Graphviz Class* and *Traits* must be selected for *DIAGRAM TYPE* and *SHOW VIEW* respectively. In order to switch to the flattened model, modelers simply need to deselect Traits in *SHOW VIEW*. These two views can also be generated through the Umple command line interface and the Umple Eclipse Plugin.

In order to see the flattened model textually, first the menu *TOOLS* must be selected. Then, from the drop down named *GENERATE* select *Internal Umple Representation*. Finally, click on *Generate code* to see the result or to download it.

Before and After Statements

You can request that certain code be run before or after various Umple-defined actions on attributes, associations, and the components of state machines as well as on user-defined methods.

Using 'before' allows you to enforce preconditions, such that the attribute or association will not be set if the precondition is not true. To reject setting a value, add 'return false'.

Code in a setMethod can access the value to be set by prefacing it by 'a' as in the second example or 'an'.

You can add before and after clauses to add code to generated methods of the form getX, setX, addX, removeX, getXs (to get all elements of an association), numberOfXs, indexOfX, where X is the name of the attribute or association. You can also add code before and after the constructor.

See also this example of using aspect-orientation to help interface Umple with existing libraries.

Example with "after" statement

view plain

```
01. class X
02. {
03.   a;
04.   whenWasASet;
05. 
06.   after setA {
07.      setWhenWasASet(getA() + System.currentTimeMillis());
08.   }
09. }
10. // @@@skipcppcompile
11. 
12. 
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example with "before" statement

view plain

```
01. class Person {
02.   name;
03.   before setName {
04.     if (aName !
    = null && aName.length() > 20) { return false; }
05.   }
06. }
07. // @@@skipcppcompile
08.
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example with both "before" and "after" statements

view plain

```
01. class Operation {
02.   const Boolean DEBUG=true;
03.   query;
04.   before constructor {
05.     if (aQuery == null)
06.     {
07.       throw new RuntimeException("Please provide a valid query"
08.     }
09.   }
10.   after constructor {
11.     if (DEBUG) { System.out.println("Created " + query); }
12.   }
13. }
14. // @@@skipcppcompile
15.
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

// Aspect oriented code injection: BeforeandAfterStatements
codeInjection- : [[beforeCode]] | [[afterCode]]

beforeCode : (before | [=around]) [[aspectBody]]

afterCode : after [[aspectBody]]

Code Injection Pattern Matching

When creating before and after statements, you can arrange for the code to be injected into certain methods that match a pattern. You can use * as a wildcard, and ! as an exclusion operator. The following examples will both generate the same code.

Another example of pattern matching for code injection can be found on the page on pooled state machines, where the same code is injected into all the event methods matching a certain pattern

You can specify that code should be injected only into generated methods by specifying the word 'generated' after before or after. Similarly you can specify only custom methods using the keyword 'custom'. The keyword 'all' indicates to inject the code into all matching methods.

Example

view plain

```
01. class Student
02. {
03.   const Boolean DEBUG = true;
04.   firstName;
05.   lastName;
06.   cityOfBirth;
07.   before get*Name {
08.     if ( DEBUG) { System.out.println("accessing the name"); }
09.   }
10. }
11. // @@@skipcppcompile
12.
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. class Student
02. {
03.   const Boolean DEBUG = true;
04.   firstName;
```

```
05.   lastName;
06.   cityOfBirth;
07.   before get*,!getCityOfBirth {
08.     if ( DEBUG) { System.out.println("accessing the name"); }
09.   }
10. }
11.
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Code Injection in Event Methods

State machine event methods are generated automatically from a state machine. Sometimes a programmer wants to execute some code whenever a given event is processed by a state machine, no matter what state the state machine is in.

Code can be injected before or after any event, as shown in the example below. Note that in the case of queued or pooled state machines, the injected code will only be triggered when the event is actually processed by the state machine (i.e. when it is removed from the queue).

Example

view plain

```
01. class Car {
02.   queued radio {
03.     on {
04.       radioToggle -> off;
05.     }
06.     off {
07.       radioToggle -> on;
08.     }
09.   }
10.   after radioToggle {soundBeep();}
11. }
12.
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Code Injection in Custom Methods

Much of the usefulness of code injection is for generated methods, but it is also possible to inject code into custom methods, using the custom keyword.

Example

view plain

```
01. // This illustrates separate code injection
02. // for custom and generated methods
03. // The output will be as follows
04. // Sound of beep
05. // Warning
06. //  ... this is injected after the warning
07. // Sound of beep
08. // Warning
09. //  ... this is injected after the warning
10. // Sound of beep
11. // Warning
12. // ... this is injected after the warning
13. //
14. class Car {
15.   queued radio {
16.     on {
17.       radioToggle -> off;
18.     }
19.     off {
20.       radioToggle -> on;
21.     }
22.   }
23.   after generated radioToggle* {soundBeep();}
24. 
25.   void soundBeep() {
26.     System.out.println("Sound of beep");
27.     radioToggleWarning();
28.   }
29. 
30.   void radioToggleWarning() {
31.     System.out.println("Warning");
32.   }
33. 
34.   after custom radioToggle* {
35.     System.out.println(
```

```
36.       " ... this is injected after the warning");
37.   }
38.
39.   public static void main(String[] args) {
40.     Car x = new Car();
41.     x.radioToggle();
42.     x.radioToggle();
43.     x.radioToggle();
44.   }
45. }
46.
47.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Top-level Aspects

Aspects in Umple can also be specified at top level (i.e. outside any classes). In this case, a set of classes in which to inject the code must be specified, within curly brackets. Glob patterns can be used to match a set of classes, where an asterisk is the wildcard character.

Example with "after" statement that will inject code in all set functions in all classes

view plain

```
01. after {*} set* { doSomething(); }
02.
03. class Student1
04. {
05.   name;
06. }
07.
08. class Student2
09. {
10.   name;
11. }
12.
13. class Employer
14. {
15.   age;
16.   String testFunction() {
17.     return "This is a test function";
18.   }
19. }
20.
21. class Student1 {isA X;}
22. class Student2 {isA X;}
23. class Employer {isA X;}
24. trait X {
25.   doSomething() {
26.     System.out.println("Doing something\n");
27.   }
28. }
29.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example that will inject code after two specific methods in all classes starting with Student

view plain

```
after {Student*} setName,testFunction { doSomething(); }
class Student1
{
   name;
}
class Student2
{
   name;
}
class Employer
{
   age;
   String testFunction() {
      return "This is a test function";
   }
}

class Student1 {isA X;}
class Student2 {isA X;}
trait X {
  doSomething() {
    System.out.println("Doing something\n");
  }
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

toplevelCodeInjection- : [[toplevelBeforeCode]] | [[toplevelAfterCode]]

toplevelBeforeCode : (before | [=around]) {([className])(, [className])*} [[aspectBody]]

toplevelAfterCode : after {

([className]
)
(, [className]
)*} [[aspectBody]]

Around Statement

An around statement can be used used to inject code around a code block. The original code appears in the middle, wherever the special around_proceed keyword is placed. The first example below surrounds the body of an entire method. The second example injects code between two labels

Example with "in class around injection with no labels" statement

view plain

```
01. class AroundClass{
02.   static void doSomeThing()
03.   {
04.     int x; int a;
05.     boolean flag = false;
06.     Label1: x = 0;
07.     Label2: a = 0;
08.     flag = true;
09.   }
10. }
11. 
12. class AroundClass
13. {
14.   around custom doSomeThing()
15.   {
16.     // code before around.
17.     if (true)
18.     {
19.       around_proceed:
20.     }
21.     // code after around.
22.   }
23. }
24. 
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example with "top-level around injection with two labels" statement

view plain

```
01. class AroundClass{
02.   static void doSomeThing()
03.   {
04.     boolean flag = false;
05.     Label1: int x = 0;
06.     Label2:
07.     flag = true;
08.   }
09. }
10. 
11. around {AroundClass} custom Label1-Label2:doSomeThing()
12. {
13.   // code before around.
14.   if (true)
15.   {
16.     around_proceed:
17.   }
18.   // code after around.
19. }
20. 
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

toplevelBeforeCode : (before | [=around]) {([className])(, [className])*} [[aspectBody]]

Basic Mixsets

Mixsets can be used to create different members of the same *product line* from a given Umple model, or to help divide the system into features for *feature-oriented development*. They can also be seen as providing conditional compilation capabilities such as those commonly found in C++ or C.

Mixsets are one of several separation-of-concerns capabilities available in Umple. The others being mixins, traits, aspect orientation and filters.

A mixset is a named set of fragments of Umple code that may or may not be included in the system. A mixset is included using a *use* statement or a command line argument, in the same manner that a .ump file is included. So, taken together the fragments can be seen as a virtual file.

When a use statement (or command line argument) matching the mixset's name is encountered, then the fragments comprising the mixset are mixed in to the Umple code using Umple's mixin mechanism, which allows multiple definitions of an entity such as a class to be combined.

A mixset fragment can be defined in the following ways:

- At the top level of any Umple file, simply by the notation mixset name followed by a block in curly brackets. The block in curly brackets can be any Umple code.
- Within a class, trait or other top-level entity. In this case the contents of the block are included in that entity (if a matching use statement is encountered).

Example

view plain

```
01. // Initially there are two classes with no attributes
02. class X {}
03. class Z {}
04.
05. // In another place, potentially a separate file.
06. // class X is given attribute a
07. // This is a simple mixin
08. class X {
09.  a;
10. }
11.
12. // In a third place we conditionally want to
```

```
// include attribute b, perhaps only in certain
// versions of the software.
mixset specialVersion {
  class X {
    b;
  }
}

// To activate the specialVersion mixset we need
// to encounter the following
use specialVersion;

// We can also have another 'fragment' of
// the specialVersion mixset elsewhere
mixset specialVersion {
  class X {
    c;
  }
}

// The following notations can also be used
class X {
  mixset specialVersion {d;}
}

// Any features of a class can be incorporated
// using a mixset including the following
// Here we introduce a second mixset
mixset specialVersion2 {
 class Z {p;}
}

class X {
  mixset specialVersion2 {
    isA Z;
    0..1 -- * Z;
  }
}

use specialVersion2;

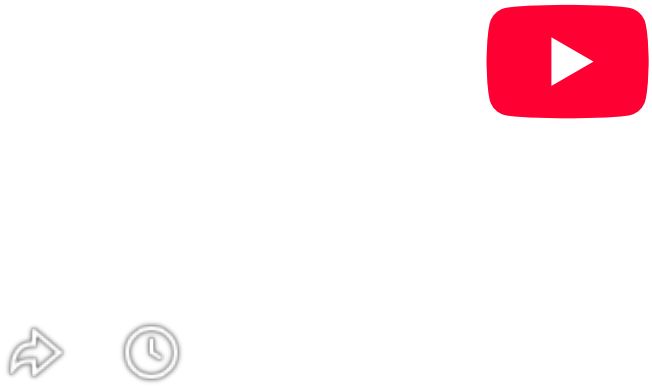
// The following mixset will be ignored since
// there is no use statement for it
mixset specialVersion3 {
 class W {}
```

```
58. }
59.
60.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

YouTube Video with Additional Explanation

Abdualziz AlGablan



Syntax

```
// Anything inside a class that is not parsable by Umple is passed on to the base language
// compiler. To raise warnings when this occurs, use the strictness directive.
extraCode- : [**extraCode]

mixsetDefinition : mixset [mixsetName]
  ( [[mixsetInnerContent]]
    | [[requirement]]
    | [[mixsetInlineDefinition]]
  )
```

mixsetInnerContent- : { [[extraCode]] | [[reqImplementation]] }

mixsetInlineDefinition- :
([entityType] [entityName]
([[mixsetInnerContent]]
| [entityOneElement]
)
| [oneElement]
)

Mixsets in Methods

Umple mixsets can be used to conditionally construct variant versions of methods that enable different members of a product line to behave differently. There are multiple ways this can be done. The simplest way is to add a mixset statement *directly* into a method, as seen on line 11 of the example below. In this example, the code on line 12 will only be part of the method if the World mixset is activated through a use statement or command line argument.

The second way to use mixsets with methods is to create mixsets that contain Umple aspects that inject code in methods. In the example below, the mixsets starting at lines 19, 27 and 35 all do this.

In Umple, code labels (identifiers followed by a colon) can be used to indicate variation points at multiple points among any method's statements. These labels can be utilized by aspects to indicate where to inject code into the methods, and can even enable replacing the code between two labels. The injected code has direct access the local scope (i.e. local variables) at the locations where it is injected. When combined with mixsets, aspect injection at labels therefore provides *fine grained variability* at the statement level. It can be used as an alternative to the approach of using hook methods (empty methods that are called and do nothing by default but can be overridden in subclasses) to provide variability.

The main method in the example below contains three labels: Hello_Label, Beautiful_Label, and Wonderful_Label. They are utilized by the aspects of three mixsets (Hello, Beautiful, and Wonderful) to inject some print statements.

Example

view plain

```
01. class Main
02. {
03.   // The method below has three code labels
04.   // plus an inline mixset (line 11).
05.   // These labels act as points of extensions.
06.
07.   void main(String[] arg){
08.     Hello_Label: ;
09.     Beautiful_Label: ;
10.     Wonderful_Label: ;
11.     mixset World {
```

```
12.       System.out.print(" world !");
13.     }
14.   }
15. }
16.
17. // The following three mixsets inject code
18. // based on the label specified after "before".
19. mixset Hello {
20.   class Main
21.   {
22.     before Hello_Label: main(String) {
23.       System.out.print(" Hello ");
24.     }
25.   }
26. }
27. mixset Beautiful {
28.   class Main
29.   {
30.     before Beautiful_Label: main(String) {
31.       System.out.print(" beautiful ");
32.     }
33.   }
34. }
35. mixset Wonderful {
36.   class Main
37.   {
38.     before Wonderful_Label: main(String) {
39.       System.out.print(" wonderful ");
40.     }
41.   }
42. }
43.
44. // use statements activate mixsets.
45. use Hello;
46. use Beautiful;
47. //use Wonderful;
48. use World;
49.
50.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
aspectBody- :
  ([=operationSource:custom
    |generated
    |all]
  )?
  ([!codeLabel:\S+:]
  )? [[injectionOperation]]
  ( , [[injectionOperation]]
  )*
  ([[codeLang]] [[codeLangs]]
  )? { [**code] }
  ( [[moreCode]]
  )*

mixsetDefinition : mixset [mixsetName]
  ( [[mixsetInnerContent]]
    | [[requirement]]
    | [[mixsetInlineDefinition]]
  )

mixsetInnerContent- : { [[extraCode]] | [[reqImplementation]] }
```

Require Statement

Require statements in Umple are used to specify dependencies and relationships among mixsets.

In the simplest case a require statement simply declares that certain code requires the presence of a certain mixset and that it would be an error if there is no use statement for this mixset. If the require statement is itself in a mixset, then it is stating that mixset requires the presence of one or more others. In the first code example below, the mixset M1 (Line 2) requires mixset M2 (Line 6). Mote that the specification of the required mixset M2 is in square brackets: That is because the argument to the require statement is a Boolean condition involving mixsets with logical connectors

Require statements may include the subfeature keyword to specify mixsets that are features. A require statement containing subfeature forms a parent-child relationship between the source mixset and the target mixset and both will become features.

A require statement at the top level of a file always applies. A require statement inside a mixset applies only when that mixset is used. When the use statements do not satisfy a require statement that applies, the compiler raises warning W1514 and the feature diagram shows the configuration as not valid.

When a mixset acts as an optional feature; the keyword isFeature can be included in the mixset body instead of using the require statement. The keyword isFeature makes a mixset an optional feature to the container mixset (the one which the former is nested in). If the mixset containing isFeature is a top entity element in an Umple file, the mixset becomes an optional feature to the SPL base.

The second code example below contains multiple features that form a feature model. Line 5 indicates that GSM1800 is required but that GSM1900 is not. Line 9 states that one or both of two sound features is required. Line 18 states that up to one of three resolution features is required (i.e. only at most one). Line 21 contains a use statement that produces a software variant with five features.

Example Showing mixset M1 requiring the presence of mixset M2

view plain

```
01. // M1 and M2 are mixsets that are not part of the feature model.
02. mixset M1 {
```

```
 require [M2];
}

mixset M2 { }

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example with various require statements that generates a feature diagram

view plain

```
// These require statements form a feature model.
require subfeature [GSMProtocol opt Mp3Recording and Playback

mixset GSMProtocol {
  require subfeature [GSM1800 opt GSM1900];
}

mixset AudioFormat {
  require subfeature [1..2 of {Mp3,Wav} ];
}
mixset Mp3Recording {
  require [Mp3];
}
mixset Camera {
  require subfeature [Resolution];
}
mixset Resolution{
  require subfeature [0..1 of {Res21MP, Res31MP, Res50MP}];
}

use GSMProtocol, GSM1800,  Playback, AudioFormat, Wav;

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example with isFeature statements that generates optional features

```
view plain

01. // Because Debug contains isFeature, it is an optional feature to t
02.
03. mixset Debug{
04.
05.   isFeature;
06.
07.   // Because ConsolePrint contains isFeature, it is an optional feat
08.   mixset ConsolePrint
09.   {
10.     isFeature;
11.   }
12.
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

requireStatement : require ([=subfeature])? [[requireBody]]

requireBody- : [
  (
  ([[requireLinkingOptNot]]
)?
  ((
)* [[requireTerminal]] [[requireList]]
)]

requireList- :
  ([[requireLinkingOp]]
  ((
)* [[requireTerminal]]
  ()
)*
)*

requireLinkingOp :
  ([[requireLinkingOptNot]]
    | [=and:&|&&|and|,]
    | [!or:

```
  ([|][|]?|or|;
  )]
    | [=xor:xor|XOR]
  )

requireLinkingOptNot- : ([=opt:opt] | [=not:not] )

requireTerminal : [~targetMixsetName] | [[multiplicityTerminal]]

multiplicityTerminal- : [[multiplicity]] of { [~targetMixsetName]
  (, [~targetMixsetName]
  )* }
```

Mixsets with Traits

Traits can include mixsets with their bodies. For example, the code below contains the 'Numerical' mixset, which ensures the implementaion of extra methods (isGreaterThan and isLessThan) when its activited by a use statement.

Example

view plain

```
01. trait TEquality{
02.   Boolean isEqual(Object object);
03.   Boolean isNotEqual(Object object) Java {
04.
05.     return isEqual(object) ? false : true;
06.   }
07.
08.   mixset Numerical {
09.     Boolean isGreaterThan(Object object);
10.     Boolean isLessThan(Object object);
11.   }
12. }
13.
14. class RealNumber
15. {
16.   float value;
17.   isA TEquality;
18.
19.   Boolean isEqual (Object object)Java
20.   {
21.     if(object instanceof RealNumber)
22.     return this.value == ((RealNumber) object).value;
23.     else
24.     return false;
25.   }
26. }
27.
28. /*
29.    The below use statement generates an error in Umple when its
30.    Because the trait TEquality has the Numerical mixset, and there
31.    implementaion for its methods (isGreaterThan and isLessThan).
32.  */
33. //use Numerical;
34.
```

```
35.
36.
37.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

traitDefinition : trait [name] [[traitParameters]]? { [[traitContent]]* }

traitContent- : [[mixsetDefinition]]
| [[testCase]]
| [[requiredModelElements]]
| [[comment]]
| [[reqImplementation]]
| [[traitDefinition]]
| [[trace]]
| [[position]]
| [[displayColor]]
| [[abstract]]
| [[keyDefinition]]
| [[softwarePattern]]
| [[depend]]
| [[symmetricReflexiveAssociation]]
| [[attribute]]
| [[stateMachine]]
| [[inlineAssociation]]
| [[concreteMethodDeclaration]]
| [[abstractMethodDeclaration]]
| [[constantDeclaration]]
| [[invariant]]
| ;
| [[exception]]
| [[extraCode]]

mixsetDefinition : mixset [mixsetName]
([[mixsetInnerContent]]
| [[requirement]]
| [[mixsetInlineDefinition]]
)

Mixsets with State Machine

State machine models can contain variant units by employing mixsets. These can be inline (directly specified within a state macine) or compositional (in a separate file). For example, the mixset "HalfOpenFeature" in the code below captures the state "HalfOpen" and with a transition from "Opening" to the "HalfOpen" state. The second code example shows an equivalent state machine model, but with a compositional mixset.

Example showing conditional transitions specified using a mixset

view plain

```
01. // This example shows how a state machine model can incorporat
02. class GarageDoor
03. {
04.   status {
05.     Open { buttonOrObstacle -> Closing;  }
06. 
07.     Closing {
08.       buttonOrObstacle -> Opening;
09.       reachBottom -> Closed;
10.     }
11. 
12.     Closed { buttonOrObstacle -> Opening; }
13. 
14.     Opening {
15.       // HalfOpenFeature mixset below is an inline mixset.
16.       // It contains an event (a part of the state machine).
17.       mixset HalfOpenFeature {
18.         buttonOrObstacle -> HalfOpen;
19.       }
20.       // end of HalfOpenFeature.
21.       reachTop -> Open;
22.     }
23.     // HalfOpenFeature mixset here contains a state (with its conte
24. 
25.     mixset HalfOpenFeature {
26.       HalfOpen { buttonOrObstacle -> Opening; }
27.     }
28.   }
29. }
30. 
31. // A use statement to activate HalfOpenFeature mixset.
```

```
32.  use HalfOpenFeature;
33.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example of a mixset incorporated compositionally to modify a state machine

view plain

```
01.  class GarageDoor
02.  {
03.    status {
04.      Open { buttonOrObstacle -> Closing;  }
05.      Closing {
06.        buttonOrObstacle -> Opening;
07.        reachBottom -> Closed;
08.      }
09.      Closed { buttonOrObstacle -> Opening; }
10.      Opening {
11.        reachTop -> Open;
12.      }
13.    }
14.  }
15.  // The mixset below contains a compostional mixsets.
16.  mixset HalfOpenFeature {
17.    class GarageDoor
18.    {
19.      status {
20.        Opening {
21.          buttonOrObstacle -> HalfOpen;
22.        }
23.        HalfOpen { buttonOrObstacle -> Opening; }
24.      }
25.    }
26.  }
27.  // A use statement to activate HalfOpenFeature mixset.
28.  use HalfOpenFeature;
29.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
//Issue 148
inlineStateMachine : [=queued]? [=pooled]? [~name] {
  ( [[comment]]
    | [[reqImplementation]]
    | [[state]]
    | [[trace]]
    | [[mixsetDefinition]]
    | [=||]
    | [[standAloneTransition]]
  )* }

stateEntity- : [=||]
    | [[mixsetDefinition]]
    | [[entryOrExitAction]]
    | [[autoTransition]]
    | [[transition]]
    | [[activity]]
    | [[state]]
    | [[displayColor]]
    | [[trace]]
    | ;

mixsetDefinition : mixset [mixsetName]
  ( [[mixsetInnerContent]]
    | [[requirement]]
    | [[mixsetInlineDefinition]]
  )
```

Mixset Feature Tree Selector

UmpleOnline includes a feature-tree panel (opened from the *Show and Hide* toolbar) that renders every mixset in the current model as a node in a hierarchical tree. The tree also visualises feature-model constraints (and / `or` / `xor` / multiplicity) defined by `isFeature` and `require` statements, so parent-child relationships and mutually-exclusive alternatives are shown graphically.

Clicking a node toggles the corresponding `use` statement and instantly re-generates the diagram for the resulting configuration, giving a single-click way to switch between alternative designs. Nodes whose activation would violate a feature-model constraint are highlighted.

The panel is backed by the FeatureModelJson generator, which exposes the model's feature structure — including a top-level `requirements` array and per-mixset `reqImplementations` — as JSON consumed by the browser.

Combined with quality requirements and the quality comparison table, the selector supports design-decision workflows in which several mixsets represent alternative designs and can be switched, compared, and benchmarked without leaving UmpleOnline.

Filters

A filter directive in Umple can be used to select only a certain part of a model (i.e. a set of classes including their associations) and ignore the rest. It is particularly designed to allow creation of different diagrams from the same model. However it could also be as a way of building a particular version of the system that only has certain features. It is one of Umple's separation of concerns mechanisms.

There are two types of filter statements, *named* and *unnamed*. An unnamed filter statement is active by default, and tells the system to include only certain indicated classes (multiple such filters are shown in the example below). A named filter statement is ignored until activated with an includeFilter statement (the example below shows two named filters appearing after the Filter 7 comment).

Filters can be used to describe a particular diagram (or system version). There can be any number of named filters, but only one unnamed filter should be active at one time.

Within a filter directive, one can specify

- ***include*** statements. These list the main classes to be included. The superclasses of these classes are included too. Classes are specified either by their **full name** or using a **pattern** where an asterisk matches zero or more characters and a question mark matches any single character. A pattern can also start with a tilde ~, meaning to *exclude* the class or classes, rather than include them. Multiple class names or patterns are separated by commas. Several include statements can be seen in the example below. Any associations between included classes are shown (see Filter 8)

- Optionally ***includeFilter*** statements. These include named filters. To activate a named filter, it is necessary to use an includeFilter statement in an unnamed filter. See Filter 7 in the example.

- Optionally ***namespace*** statements. These include all classes in the given namespace.

- Optionally ***hops*** statements. These indicate that additional classes should be included that are connected to included classes by associations or generalizations. See examples 2 and 6 for association hops, and example 4 for a subclass hop. As mentioned above, including any class will by default include its superclasses, so the complete picture of what is inherited is presented; to avoid

including superclasses, specify **hops {super 0;}**, but generated code will be very unlikely to work in such a case.

Consider wrapping filter statements within mixsets. This will allow you to turn on and off various filters using command line arguments.

Load the example below into UmpleOnline and uncomment any of the filter statements, one at at a time, to see the effect of filtering. The comment just before each filter statement describes the effect. Note that the named filters described as Filter 7 have been left uncommented as they are ignored until the un-named filter following them is activated.

Example

view plain

```
01. // Example to demonstrate filters. Uncomment one of
02. // the filters shown below to display a different
03. // diagram, or generate a different subset of the
04. // code
05.
06. // This model describes abstract art consisting of
07. // various shapes connected in 3-D space, some
08. // of which can contain others.
09.
10. // Filter 1. Show only Shape3D
11. // filter {include Shape3D;}
12.
13. // Filter 2: Show classes connected to Model3D with
14. // one hop
15. //  filter {include Model3D; hops {association 1;}}
16.
17. // Filter 3: Show classes related to qualities
18. // filter {namespace qualities ;}
19.
20. // Filter 4: Show Shape3D, its subclasses and all
21. // their connections
22. // filter {include Shape3D; hops {sub 1;}}
23.
24. // Filter 5: Including also includes parents
25. // filter { include Cube;}
26.
27. // Filter 6: Show two classes and their neighbours
28. //  filter {include Surface, Model3D; hops {association 1;}}
29.
30. // Filters 7: Named filters and use of the named filter
```

```
31.   filter 7 {include Connector; hops {association 1;}}
32.   filter 7a {include Cube;} // includes its parents
33.  //  filter {includeFilter 7,7a;}
34.
35.  // Filters 8: Including classes that are associated
36.  // will always show the associations
37.  //  filter { include Colour, SurfaceQuality;}
38.
39.  // Filter 9: Use of patterns
40.  //filter { include M????3D,*Polyhedron;}
41.
42.  // Filter 10: Excluding those that start with P (except superclasses
43.  // filter { include ~P*;}
44.
45.  class Point3D {
46.    Double x;
47.    Double y;
48.    Double z;
49.  }
50.
51.  class Model3D {
52.    depend Qualities.*;
53.    depend Shapes.*;
54.    name;
55.    1 -- * Shape3D;
56.    1 -- * Connector;
57.  }
58.
59.  namespace Qualities;
60.
61.  class Colour {
62.    Integer red;
63.    Integer green;
64.    Integer blue;
65.  }
66.
67.  class SurfaceQuality {
68.    * -> 1 Colour;
69.    Float reflectance;
70.    Float transparency;
71.  }
72.
73.  namespace Shapes;
74.
75.  // Shape3Ds are nodes, they can also contain other
```

```
// Models inside themselves.
class Shape3D {
  // All surfaces, points, connections
  // are relative to the centre which acts
  // as the Shapes origin.
  // The centre also positions the shape in 3-space.
  * -> 1 Point3D centre;

  // An Internal 3D model appears inside a Shape3D
  // All points are relative to the shape's centre
  0..1 containingShape -<@> 0..1 Model3D internalModel;
}

// A surface is used to describe the shape
// of one of the sides of a polyhedron
// points are relative to the centre of the
// polyhedron. The points must fall on some
// 3D plane.
class Surface {
  * -> * Point3D;
  * -> 1 SurfaceQuality;
}

// Connectors are sticks connecting 3D Shapes
class Connector {
  * -> 2 Point3D ends;
  * -- 2 Shape3D connectedShapes;
  Double radius;
  * -- 1 SurfaceQuality;
}

class Sphere {
  isA Shape3D;
  Double radius;
  * -> 1 SurfaceQuality;
}

class Polyhedron {
  depend Qualities.*;
  isA Shape3D;
  1 -- * Surface;
}

class RegularPolyhedron {
  isA Polyhedron;
```

```
121. }
122.
123. class Cube {
124.   isA RegularPolyhedron;
125. }
126.
127. strictness ignore 42; // hide namespace warnings
128.
129.
130.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

association : [=modifier:immutable]? [[associationEnd]] [=arrow:--
|->
|<-
|><
|<@>-
|-<@>] [[associationEnd]] ;

filter : filter
([filterName]
)? {
([[filterStatement]]
)* }

filterStatement- : [[filterCombinedValue]] | [[filterNamespace]] |
[[filterValue]] | [[hops]]

hops- : hops {([[super]] | [[sub]] | [[association]])*}

filterValue : include ([classname]) (, [classname])*;

super : super [superNum];

sub : sub [subNum];

association : association [associationNum];

filterNamespace : namespace [Namespace] (, [Namespace])* ;

```
filterCombinedValue : includeFilter ([filterName]) ( , [filterName] )*;
```

Basic Templates

Generating string output is a very common task for programs. Common types of output include html, xml and executable code. Umple has a special capability for this, as do many other technogies (it is central to PhP, for example). The advantage of Umple's approach is that it adds generation templates in a uniform manner to C++, Java, PhP and other languages that Umple supports. The same templates can be reused.

The Umple generation templates are essentially a readable way to generate special methods that emit Strings (and also in Java's case StringBuilders).

Two basic elements are needed to use generation templates:

Templates The first essential element is the templates themselves. These are shown as a name followed by arbitrary text in <<! !>> brackets. The text in the brackets can contain anything you want. See the examples below to understand how these are used.

```
<<!output this!>>
```

will build the string 'output this' when specified in an emit method (below).

Emit method specifications: The second essential element is one or more 'emit' statements. These specify the methods to be generated. As with any method there can be a set of arbitrary parameters. Following this, in a second set of parameters is comma-separated list of templates to emit. For example, the following says to generate a method with signature String printRow(intTimes, intCount); that will emit a string containing the contents of the row and cr templates:

```
emit printRow(int times, int count)(row, cr);
```

Optional elements in templates are:

Expression blocks: Inside the template, programmers can specify arbitrary expressions in <<= >> brackets. These can refer to attributes of the class, parameters to the emit method, states and so on. The result of

the expression will be substituted every time the template method is called. This appears in all examples below.

Code blocks: Also inside the template program logic can be embedded in <<# #>> brackets. This enables conditional emission of parts of a template, or looping within the template. This appears in the second example below.

Comment blocks: Comments in templates can be shown within <</* */ >>

The first two examples below show how simple templates can be used to output strings, in this case a one-column multiplication table. The first example uses two template methods, the second being called in a loop. The second example generates the same output, but all the looping logic is enclosed in the template itself. The third example shows a more substantial template with a lot of 'boilerplate' text that is easy to read and edit in the Umple source. Manually writing the generated would be substantially more awkward.

Umple's mixin capability allows templates to be kept in separate files. This can faciliatate reuse.

Example 1

view plain

```
01. // Simple example to demonstrate umple's template
02. // base string generation mechanism.
03. // In this approach there is an iterative call
04. // to the emit function
05.
06. class MathExample {
07.
08.   // A simple template to inject a platform
09.   // specific newline
10.   cr <<!
11. !>>
12.
13.   // Two templates for different lines of output
14.   header <<!Print out a <<=times>> times table!>>
15.   row <<!  <<=times>> times <<=count>> is <<=times*count
      >>
16.
```

```
17.   // Specification for two methods to be generated
18.   // to output parts of the text
19.   // Method arguments are in the first parentheses
20.   // Templates to output are in the second set
21.   emit printHeader(int times)(header, cr);
22.   emit printRow(int times, int count)(row, cr);
23.
24.   // Main program to run the above and generate
25.   // the output
26.   public static void main(String[] argv) {
27.     int times = 10; // default
28.     MathExample m = new MathExample();
29.
30.     if(argv.length > 0) {
31.       times=Integer.parseInt(argv[0]);
32.     }
33.
34.     // Print the header
35.     System.out.print(m.printHeader(times));
36.     // Print one row for each element
37.     for (int i=0; i <= times; i++) {
38.       System.out.print(m.printRow(times,i));
39.     }
40.   }
41. }
42.
43.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 2

view plain

```
01. // Simple example to demonstrate umple's template
02. // base string generation mechanism.
03. // In this approach iteration is embedded in the
04. // rows template and there is a single emitter
05. // function generated called result
06.
07. class MathExample {
08.
09.   // A simple template to inject a platform
```

```
  // specific newline
  cr <<!
!>>

  // A template for the header lines
  header <<!Print out a <<=times>> times table!>>

  // A template that generates all the rows by
  // iterating
  rows <<!<<# for  (int i=0; i <= times; i++) {#>>

<<=times>> times <<=i>> is <<=times*i>><<#}#>>!>>

  // Specification of a single method to emit
  // the result
  emit result(int times)(header, rows, cr);

  public static void main(String[] argv) {
    int times = 10; // default
    if(argv.length > 0) {
      times=Integer.parseInt(argv[0]);
    }
    // Output the entire result
    System.out.print(
      new MathExample().result(times));
  }
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example 3

view plain

```
// Example of creating a lengthy output in Umple
// from a template. Note this is plain text.
// See the next page for html generation.
// The company is, of course, fictitious.
class RefLetterRequest {
  // Attributes used to construct the instance
  String fileno; String recipient; String applicant;
```

```
08.   String sender; String senderTitle;
09.
10.   // Letter template
11.   letterTemplate <<!
12. Subject: Reference request for <<=applicant>>, File #<<=filenc
13.
14. Dear <<=recipient>>,
15. Our company, Umple Enterprises, is hiring talented software
16. engineers.
17.
18. We have received an application from <<=applicant>> who name
19. as an individual who could provide a letter of reference. Would yo
20. please reply to this letter, answering the following questions:
21.   * In what capacity do you know <<=applicant>>
22.   * For how long have you known <<=applicant>>
23.   * Describe the abilities of <<=applicant>> in software developr
24.   * What his or her strengths and weaknesses?
25.   * Please provide your phone number and suitable times to call i
26.     case we need to follow up
27.
28. Yours sincerely,
29. <<=sender>>
30. <<=senderTitle>>
31. !>>
32.
33.   // Specification of the method to generate
34.   emit letterTemplate()(letterTemplate);
35.
36.   // Main program to generate the letter
37.   public static void main(String[] argv) {
38.     if(argv.length < 5) {
39.       System.err.println("You must specify arguments for fileno, rec
40.     }
41.     // Output the entire result
42.     else System.out.print(new RefLetterRequest(
43.       argv[0], argv[1],argv[2], argv[3], argv[4]
44.     ).letterTemplate());
45.   }
46. }
47.
48.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
templateAttributeDefinition : [[templateName]] [[templateAttribute]]

templateName :
  ( [~classname] .
  )? [name] [[templateParameters]]?

emitMethod : [=modifier:public
    |protected
    |private]? [=static]? [type]? [=emit] [[methodDeclarator]]
[[templateList]]?;

templateList- : (
  ( [[templateName]]
  ( , [[templateName]]
  )*
  )? )

templateAttribute# : <<! [[templateAttributeContent]]* !>>

templateAttributeContent- : [[templateExpression]]
    | [[templateComment]]
    | [[templateCodeBlock]]
    | [[templateExactSpaces]]
    | [[templateInclude]]
    | [[templateText]]

templateText# : [**templateTextContent:<<(=|#|/[*]|$|@)]

templateComment# : <</* [**templateCommentContent] */>>

templateExpression# : <<=
  ( [[templateExpression]]
    | [**templateExpressionContent:<<
  (=|#|/[*]|$|@
  )]
  )+ >>

templateCodeBlock# : <<#
  ( [[templateExpression]] | [**templateLanguageCode:<<
  (=|#|/[*]|$|@
  )]
```

```
)* #>>
```

Html Generation

The first example below illustrates the generation template capability of Umple. It is an html generation library. The second example is a program that uses the first example.

Example

view plain

```
01. // Umple library for generating html
02. // Currently it supports h1, h2 ... as well as p, and table tags.
03. // This will be extended to support more of html later
04. // It serves as an illustration of Umple's generation templates
05.
06. strictness allow 75; // We are going to allow custom constructor
07.
08. // Class representing either a top level html node or a collection o
09. class HtmlNode {
10.   String getContent() {return "";};  // should be abstract
11. }
12.
13. // Simple text to be output between tags - these are leaves
14. class HtmlTextNode {
15.   isA HtmlNode;
16.   String content;
17. }
18.
19. // Non-leaf nodes in the html tree
20. class HtmlRegularNode {
21.   isA HtmlNode;
22.   const String Xmltagstart = "<";
23.   const String Xmltagend = ">";
24.
25.   // Arguments for the constructor
26.   String tag; // e.g. p, h1, a
27.   String arguments; // e.g. href
28.
29.   0..1 -
> * HtmlNode subnodes; // whatever to emit between tags
30.
31.   // The following template will render any html node
32.   rendering <<!
<<=Xmltagstart>><<=getTag()>> <<=getArguments()>><<=
33.   <<#
```

```
34.   for(HtmlNode n : getSubnodes()) {#>><<=n.getContent()>>
35.   <<#}#>><<=Xmltagstart>>/
      <<=getTag()>><<=Xmltagend>>!>>
36.
37.   emit getContent()(rendering);
38.
39.   HtmlTable table() {
40.     HtmlTable t = new HtmlTable();
41.     addSubnode(t);
42.     return t;
43.   }
44.
45.   HtmlTextNode text(String s) {
46.     HtmlTextNode t = new HtmlTextNode(s);
47.     addSubnode(t);
48.     return(t);
49.   }
50.
51.   HtmlRegularNode taggedText(String tag, String arguments, Strir
52.     HtmlRegularNode r = new HtmlRegularNode(tag, arguments);
53.     HtmlTextNode t = new HtmlTextNode(s);
54.     r.addSubnode(t);
55.     addSubnode(r);
56.     return(r);
57.   }
58. }
59.
60. class HtmlTable {
61.   isA HtmlRegularNode;
62.
63.   HtmlTable() {
64.     super("table","");
65.   }
66.
67.   HtmlRow tr() {
68.     HtmlRow r = new HtmlRow();
69.     addSubnode(r);
70.     return r;
71.   }
72. }
73.
74. class HtmlRow {
75.   isA HtmlRegularNode;
76.
77.   HtmlRow() {
```

```
78.     super("tr","");
79.   }
80.
81.   // Add a cell that contains text
82.   HtmlCell td(String s) {
83.     HtmlCell c = td();
84.     c.text(s);
85.     return(c);
86.   }
87.
88.   // Add a cell that can contain anything
89.   HtmlCell td() {
90.     HtmlCell c = new HtmlCell();
91.     addSubnode(c);
92.     return c;
93.   }
94. }
95.
96. class HtmlCell {
97.   isA HtmlRegularNode;
98.
99.   HtmlCell() {
100.    super("td","");
101.  }
102. }
103.
104. class HtmlGen {
105.
106.  lazy HtmlNode firstNode;
107.
108.  // Subtrees for the header and body
109.  0..1 -> * HtmlNode headerNodes;
110.  0..1 -> * HtmlNode bodyNodes;
111.
112.  after constructor {
113.    firstNode = new HtmlTextNode(filehead());
114.  }
115.
116.  filehead <<!<!DOCTYPE html PUBLIC "-//W3C//
DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">!>>
117.  emit filehead()(filehead);
118.
119.  xmlns <<!xmlns="http://www.w3.org/1999/
xhtml" xml:lang="en"!>>
```

```
  emit xmlns()(xmlns);

  wholefile <<!<<=getFirstNode().getContent()>>
<html <<=xmlns()>> >
  <head>
  <<#for(HtmlNode h: getHeaderNodes()) {#>>
    <<=h.getContent()>>
  <<#}#>>
  </head>
  <body>
  <<#for(HtmlNode h: getBodyNodes()) {#>>
    <<=h.getContent()>>
  <<#}#>>
  </body>
</html>
!>>

  emit wholefile()(wholefile);

  // Creates a new simple node; but does not add it anywhere
  HtmlRegularNode simpleNode(String tag, String content) {
    HtmlTextNode t = new HtmlTextNode(content);
    HtmlRegularNode r = new HtmlRegularNode(tag,"");
    r.addSubnode(t);
    return r;
  }

  // Add a top level node of any type that contains a string
  HtmlNode addNode(HtmlNode n) {
    addBodyNode(n);
    return n;
  }

  // Add a node of any type that contains a string
  HtmlRegularNode addSimpleNode(String tag, String text) {
    HtmlRegularNode t = simpleNode(tag,text);
    addBodyNode(t);
    return t;
  }

  // Adds a top level header at a certain level with plain text
  // Subsequent calls can be made to add additional elements
  // to the result
  HtmlRegularNode h(int level, String text) {
    return addSimpleNode("h"+level,text);
```

```
165.   }
166.
167.   // Adds a top level p with plain text
168.   // Subsequent calls can add additional elements to the p
169.   // beyond just the text
170.   HtmlRegularNode p(String text) {
171.     return addSimpleNode("p",text);
172.   }
173.
174.
175.   // Adds a top level table
176.   HtmlTable table() {
177.     HtmlTable t = new HtmlTable();
178.     addBodyNode(t);
179.     return t;
180.   }
181. }
182.
183.
184.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // Umple program demonstrating generation
02. // of the html generation library for Umple
03.
04. use ../HtmlGeneration.ump;
05.
06. class TimesTable {
07.   public static void main(String[] argv) {
08.     int times = 10; // default
09.     HtmlGen h = new HtmlGen();
10.     HtmlRegularNode p;
11.
12.     if(argv.length > 0) {
13.       times=Integer.parseInt(argv[0]);
14.     }
15.
16.     // Specify the table main header
```

```
17.     h.h(1, "Multiplication Table: "+times);
18.
19.     // Generate an explanatory paragraph
20.     p = h.p("The following is a ");
21.     p.taggedText("i","","simple");
22.     p.text(" multiplication table");
23.
24.     // Generate another paragraph with some subnodes
25.     // showing a slightly different use of the API
26.     p = new HtmlRegularNode("p","");
27.     h.addNode(p);
28.     p.text("Nodes in ");
29.     p.taggedText("font", "color=\"red\"", "red");
30.     p.text(" are powers of 3");
31.
32.     h.h(2, "The table");
33.
34.     // Create a table at the top level
35.     HtmlTable t=h.table();
36.
37.     // Add the top row of the table
38.     HtmlRow r;
39.     r=t.tr();
40.     r.td("*");
41.     for(int i=1; i<= times; i++) {
42.       r.td().taggedText("b","",""+i);;
43.     }
44.
45.     // Add the interior rows of the table
46.     for(int i=1; i<= times; i++) {
47.       r=t.tr();
48.       r.td().taggedText("b","",""+i);
49.       for(int j=1; j<= times; j++) {
50.         String output=""+i*j;
51.         if((i*j)%3==0) {
52.           HtmlCell c = r.td();
53.           c.taggedText("font", "color=\"red\"",output);
54.         }
55.         else {
56.           r.td(output);
57.         }
58.       }
59.     }
60.
61.     System.out.println(h.wholefile());
```

```
  }
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Active Objects

An active object is an object that, when constructed, runs its own thread. This is indicated in Umple by the 'active' keyword followed by a block of code. A collection of active objects, working together, constitute a concurrent system.

Additional Umple features are under development to allow for sending signals to and from concurrent objects.

Example

view plain

```
01. class A {
02.   name;
03.
04.   active {
05.     System.out.println("Object "+getName()+"is now active");
06.     for (int i = 1; i < 10; i++) {
07.       System.out.println(" "+name+" was here");
08.       Thread.sleep(1000);
09.     }
10.   }
11.
12.   public static void main(String[] argv) {
13.     new A("1");
14.     new A("2");
15.   }
16. }
17.
18.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

activeDefinition : [=active] [[codeLangs]] [name]? [[moreCode]]+

Concurrency Using Do Activities

Multiple concurrent blocks that become active when in a particular state can be accomplished in Umple using a state machine with two substates separated by the || symbol, each with a do activity. Both do activities will run concurrently.

Example

view plain

```
01. class X {
02.   activesm {
03.     topstate {
04.       thread1 {
05.         do {
06.           System.out.println(
07.             "Doing the activity - Normally "+
08.             "this would last a long time");
09.           Thread.sleep(1000);
10.           System.out.println("Done thread 1");
11.         }
12.       }
13.       ||
14.       thread2 {
15.         do {
16.           System.out.println(
17.             "Doing the other activity - Again "+
18.             "this would last a long time");
19.           Thread.sleep(3000);
20.           System.out.println("Done thread 2");
21.         }
22.       }
23.     }
24.   }
25.   public static void main(String[] argv) {
26.     new X();
27.   }
28. }
29.
30.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // This is a more sophisticated example
02. class X {
03.  sm {
04.    s1 {
05.      do {
06.         System.out.println("In S1");
07.         Thread.sleep(1000);
08.         System.out.println(
09.           "Still in S1 after sleep");
10.         Thread.sleep(2000);
11.         System.out.println(
12.           "About to leave S1");
13.       }
14.       -> s2;
15.     }
16.     s2 {
17.      do {
18.         System.out.println("In S2");
19.         Thread.sleep(1000);
20.         System.out.println(
21.           "Still in S2 after sleep");
22.         Thread.sleep(2000);
23.         System.out.println(
24.           "About to leave S2");
25.       }
26.       nesteds2a {
27.         do {
28.           System.out.println("In S2a");
29.           Thread.sleep(1000);
30.           System.out.println(
31.             "Still in S2a after sleep");
32.           Thread.sleep(2000);
33.           System.out.println(
34.             "About to leave S2a");
35.         }
36.       }
37.       ||
38.       nesteds2b {
39.        do {
40.           System.out.println("In S2b");
```

```
41.           Thread.sleep(1000);
42.           System.out.println(
43.             "Still in S2b after sleep");
44.           Thread.sleep(2000);
45.           System.out.println(
46.             "About to leave S2b");
47.         }
48.       }
49.     }
50.   }
51.   public static void main(String[] argv) {
52.     System.out.println("In Main");
53.     X theX = new X();
54.     System.out.println("End of Main");
55.   }
56. }
57.
58.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Basic Distributed Systems

By default, compiling an Umple system will result in set of classes that execute as a single application operating on objects that exist in a single Java virtual machine (VM) on a single computer (node). In this default mode of operation all method calls are made locally to methods in the same running VM.

However, by using the 'distributable' keyword, modelers can instruct the Umple compiler to allow objects to be distributed to multiple running VMs most likely running on different computers (although they can be on the same computer if desired). In this mode, some method calls result in remote invocation of methods in a different VM using either Remote Method Invocation, or Web Services.

To make an Umple model into a distributable one that runs on multiple VMs, the developer needs to perform the following. Each of these steps is described in more detail later.

1. Identify the distributable classes by adding *distributable;* to such classes and set a namespace.
2. Decide on groups of objects that are to exist together on a given VM by grouping them into different runtime components
3. Identifying the VMs and the groups of objects that must run on each VM using a configuration file.
4. Run each VM.

Distributable classes

The developer must first identify which classes are allowed to have objects on multiple machines using the 'distributable' keyword. They must also set a namespace. For example:

```
namespace example;
class Customer{

   distributable;

}
```

The above need not require modifying the original code as it could be introduced using a mixin. Indeed it is possible in Umple to turn a non-distributed system into a distributed system with a minimum of code change.

The distributable feature is inherited by subclasses. One can also put distributable in an interface (It makes the classes that use this interface

distributable). Human and Customer are both distributable in the example below.

```
class Human{

  distributable;

}

class Customer{

   isA Human;

}
```

Runtime components

Objects can be grouped to run on the same machine. These groups are called runtime components. By default, all instances of the same class are grouped together in a runtime component named after the class name. For example all of the objects of the Customer class above would be in the Customer runtime component.

The runtime components are managed by a special class that Umple creates called UmpleRuntime. An instance of UmpleRuntime will exist on each running VM. It is mostly invisible to the program; it is only called by the program in a few special circumstances.

To put a specific instance of a class in a different group, one can name the runtime component of the object as the last parameter of the constructor when creating the instance of the distributable class.

In the example below, "customer1" goes into the runtime component "Customer" by default.

```
Customer customer1= new Customer ("customer 1");
```

In the example below, we put "v" in the runtime component "Customer".

```
Vendor v= new Vendor ("Vendor 1", UmpleRuntime.getComponent("Cus
```

The names of the runtime components can be any alphanumeric word and there is no need to declare them at compile time or in the configuration file. In the example below, we put "v" in the runtime component "Component1" which is not named after any class.

```
Vendor v= new Vendor ("Vendor 1", UmpleRuntime.getComponent("Com
```

An Umple program using distributed objects can call methods on an object that happens to exist in another VM. Arguments can also be passed containing objects that are on another VM. Behind the scenes proxy objects are used to make this happen, and method calls use Remote Method Invocation (RMI) by default, or else web services if specified as discussed below.

Configuration file

As mentioned above, the UmpleRuntime class is generated for every distributed system. The instance of UmpleRuntime on each VM is responsible for distributing the objects. It reads a file (by default configuration.txt) to know which VM is where, and which VM and runs which runtime components.

In the configuration file, one declares a machine using curly brackets ({}) and can define specifications of the machine in the brackets (id, ip, url,..) as shown in the example below. One also names the runtime components that the machine runs.

The first example shows a configuration file with four nodes and five runtime components.

```
{id=0 ip=192.168.2.1 port=10541 {rc1}}

{id=1 ip=192.168.2.2 {rc2, rc3}}

{{Vendor} ip=192.168.2.3 id=2 }

{id=3 ip=192.168.2.4 {Warehouse}}
```

The next example is a configuration file with two nodes on the same physical machine with different ports to distinguish them.

```
{id=0 ip=192.168.2.1 port=10541 {rc1}}

{id=1 ip=192.168.2.1 port=10540 {rc2, rc3}}
```

Finally, the following shows configuration file with all runtime components on the same node

```
{id=0 url=http://localhost  port=10541 {rc1, Vendor, rc2, rc3}}
```

Building the system

Once you have generated the code in umpleonline you can download the zip file.

Once you have unzipped the file navigate into the Java directory and create your configuration.txt file.

To compile the code run the following command in the Java directory (in this example the inner folder is named ecommerce):

```
javac ecommerce/*.java
```

Running the system

To run the system, the operator starts Java VMs for as many copies of the system as he or she desired by invoking a main class (a class with a main method). At least one of the started machines must be started using such a main class; other machines can be started by running UmpleRuntime on those machines.

The id of the machine must be communicated to the UmpleRuntime as an argument. The example below shows two different machines being started where the Java code generated by umple is in the ecommerce directory. One VM is started by running a class called Main; this starts the whole system. The other one runs Umpleruntime on machine 1.

On machine 0:

```
java ecommerce.Main
```

On machine 1:

```
java ecommerce.UmpleRuntime 1
```

Note that the command will display connection refused errors until you have started every node.

It is also possible to change the configuration file's default address:

```
java ecommerce.UmpleRuntime 1 c:\configFile.txt
```

The system will not do anything until all of the VMs in the system have started and connected.

The user can change the defaults by calling certain UmpleRuntime static methods from the main method.

```
UmpleRuntime.setThisNodeId(1);

UmpleRuntime.setFileAddress("otherconfigFile.txt");

UmpleRuntime.getInstance();
```

Syntax

```
distribute : distributable [=distributeTech:RMI
   |WS]? [=distributePattern:0
   |1
   |2
   |3]? [=distributeVal:on
   |off
   |forced] ;

distributable- : [=distributable:distributable] [=distributeTech:RMI|WS]?
;
```

Inheriting Distributable from a class

If a class is declared distributable then all children classes of that parent class are also distributable.

Example

view plain

```
01. /*
02. In this example all 3 classes are distributable because Supplier ar
03. */
04.
05. namespace example;
06.
07. class Person {
08.     distributable;
09. }
10.
11. class Supplier {
12.     String supplierID;
13.     isA Person;
14. }
15.
16. class Vendor {
17.     String vendorId;
18.     isA Person;
19. }
20.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Distributable Singleton

In a distributed system, a singleton class could mean two different semantics. One is that the class has only one instance on each node. The other semantic is to assume that there is only one instance of the object in the whole system. The developer can choose which semantic to be used for the singleton class. If a singleton class is also distributable, it means there should be only one instance of the class in the whole system and remotely accessible by other nodes. Otherwise, there can be one instance on every node.

The target runtime component of the singleton object is the name of the class by default. All of the getInstance() calls in the entire system will be redirected to a single object.

```
SingletonClass.getInstance().getName();
```

Alternatively, a specific runtime component can be given as a parameter of the getInstance() method. In this case, there will be a single object on each node.

```
SingletonClass.getInstance(UmpleRuntime.getComponent(“component
```

More Options

All classes can be set to be distributable by using the following directive:

```
distributable forced;
```

Alternatively, all classes can be forbidden to be distributable by using the following directive:

```
distributable off;
```

One can use Web services instead of RMI by adding ws after distributable keyword in the class:

```
distributable WS;
```

Note: Web services is not currently working due to an issue with outdated dependencies.

Example

view plain

```
01. distributable forced;
02. namespace example;
03.
04. class Person {
05.     String name;
06. }
07.
08. class School {
09.     String schoolName;
10. }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Extended Distributed Example

Below is a more in depth model using the distributed feature that you can load into umple online to visualize and better understand distributed systems.

The following will be put in the configuration.txt file to run the components on 3 separate nodes:

```
{id=0; port=8081; http://localhost; {component1,Manager}}

{id=1; port=8082; http://localhost; {component2,component4}}

{id=2; port=8083; http://localhost; {component3,SystemTracer}}
```

Example

view plain

```
01. /*
02. The following is an example of how distributed can be used to run
03.
04. Example Taken from
05. https://ruor.uottawa.ca/bitstream/10393/37143/5/
    Zakariapour_Amid_2018_thesis.pdf
06. */
07.
08. namespace ecommerceRMI1;
09. distributable 1 on;
10. class Warehouse
11. {
12.    distributable;
13. }
14. class Agent
15. {
16.    distributable;
17.    * -- * Warehouse;
18. }
19. class Customer
20. {
21.    distributable;
22.    public Order orderProduct(String productType,String vendorNa
23.    {
24.       for(Vendor v: getVendors())
```

```
25.         {
26.            if(v.getName().equals(vendorName))
27.               return v.makeOrder(self,v.findProduct(new ProductTy
28.         }
29.         return null;
30.      }
31.   }
32.
33.   class Vendor
34.   {
35.      public Order makeOrder(Customer aCustomer, Product aProdu
36.      {   if(aProduct==null)
37.            return null;
38.         Order aOrder= new Order(aProduct);
39.         aOrder.setCustomer(aCustomer);
40.         aOrder.setVendor(self);
41.         return aOrder;
42.      }
43.   }
44.   class Main
45.   {
46.     public static void main (String[] args){
47.      Supplier supplier1= new Supplier("supplier1",UmpleRuntime.g
48.      Supplier supplier2= new Supplier("supplier2",UmpleRuntime.g
49.      Warehouse warehouse1=new Warehouse("warehouse1",Umple
50.      Warehouse warehouse2=new Warehouse("warehouse2",Umple
51.      Vendor vendor1= new Vendor("vendor1",UmpleRuntime.getCo
52.      Vendor vendor2= new Vendor("vendor2",UmpleRuntime.getCo
53.      Customer customer1= new Customer("Customer",UmpleRunti
54.      Customer customer2= new Customer("Customer",UmpleRunti
55.      customer1.addVendor(vendor1);
56.      customer1.addVendor(vendor2);
57.      customer2.addVendor(vendor2);
58.      vendor1.addWarehous(warehouse1);
59.      vendor2.addWarehous(warehouse1);
60.      vendor2.addWarehous(warehouse2);
61.
62.      supplier1.addWarehous(warehouse1);
63.      supplier2.addWarehous(warehouse2);
64.      supplier1.createBulk("freezer",1000,"0013M2X");
65.      supplier2.createBulk("freezer",1000,"0023M2X");
66.      supplier1.createBulk("microwave",1000,"23M2X");
67.      supplier2.createBulk("microwave",1000,"13M2X");
68.      supplier1.createBulk("stove",1000,"M2X");
69.      supplier2.createBulk("stove",1000,"M3X");
```

```
70.     supplier1.createBulk("laptop",1000,"LL3L");
71.     supplier2.createBulk("laptop",1000,"LL4L");
72.
73.
74.     customer1.orderProduct("freezer","vendor1");
75.     customer2.orderProduct("freezer","vendor2");
76.     customer1.orderProduct("stove","vendor1");
77.     customer2.orderProduct("laptop","vendor2");
78.     customer2.orderProduct("laptop","vendor2");
79.     customer1.orderProduct("laptop","vendor2");
80.
81.
82.     UmpleRuntime.stopAll();
83.
84.   }
85. }
86. class Warehouse
87. {
88.     name;
89.     public Product findProduct(ProductType productType)
90.     {
91.        for(ProductTypeInWarehouse ptiw:getProductTypeInWareho
92.           {
93.              if(ptiw.getProductType().equals(productType))
94.              {
95.                 for(Product p : ptiw.getProducts())
96.                    return p;
97.              }
98.           }
99.        return null;
100.    }
101.    public void addProduct(Product p)
102.    {
103.       for(ProductTypeInWarehouse ptiw : getProductTypeInWareh
104.       {
105.          if(ptiw.getProductType().equals(p.getProductType()))
106.          {
107.             ptiw.addProduct(p);
108.             return;
109.          }
110.       }
111.       ProductTypeInWarehouse ptiw = new ProductTypeInWareho
112.       ptiw.addProduct(p);
113.       addProductTypeInWarehous(ptiw);
114.    }
```

```
}
class Agent
{
   name;
}
class Order
{
   * -- 0..1 Customer;
   * -- 0..1 Vendor;
   * -> 1 Product;
}
class Supplier
{
   isA Agent;
   0..1 -> * Product;
   public void createBulk(productType,Integer number,serialNum
   {
      for(int i=0; i<number; i+=1)
      {
         String postfix=String.valueOf(i);
         putInWarehouse(createProduct(serialNumber+postfix,pro
      }
   }
   public Warehouse createWarehouse(name)
   {
      Warehouse warehouse=new Warehouse(name);
      addWarehous(warehouse);
      return warehouse;
   }
   public Product createProduct(serialNumber, String productTyp
   {
      Product aProduct=new Product(serialNumber,new ProductT
      addProduct(aProduct);
      return aProduct;
   }
   public void putInWarehouse(Product aProduct,Warehouse war
   {
      warehouse.addProduct(aProduct);
      removeProduct(aProduct);
   }
}
class Vendor
{   isA Agent;
   public Product findProduct(ProductType productType)
   {
```

```
160.        for(Warehouse w:getWarehouses())
161.        {
162.           Product p= w.findProduct(productType);
163.           if(p!=null)
164.              return p;
165.        }
166.        return null;
167.     }
168.  }
169.  class Product
170.  {
171.     immutable;
172.     serialNumber;
173.     * -> 1 ProductType;
174.  }
175.  class ProductType
176.  {
177.     immutable;
178.     name;
179.     key {name};
180.  }
181.  class ProductTypeInWarehouse
182.  {
183.     * -> 1 ProductType;
184.     * <- 0..1 Warehouse;
185.     0..1 -> * Product;
186.
187.  }
188.  class Customer
189.  {
190.     name;
191.     * -- * Vendor;
192.
193.  }
194.
195.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Basic Composite Structure Diagrams

A composite structure diagram is used to show how various instances of classes, called *parts*, are connected at run time in order to communicate with each other.

To specify composite structure, the following are the elements that need to be present in an Umple system:

- **Classes**: Ordinary Umple classes, enhanced to have the items below.
- **Ports**: (required for a composite structure diagram to be generated) These are special attributes used as the origin and/or destination of data to be transferred among the parts.
- **Port bindings (Connectors)**: These specify how data is to be transferred among the various parts of the system. They connect ports using the -> syntax. Output of one port is transferred to the input of another port.
- **Active method declarations**: These run in their own thread when constructed and are needed to ensure that data is sent and received on the ports without delay.

The example below first defines two classes with ports; a third class contains instances of the first two.

Under development: Composite structure diagrams, ports and active methods are currently under development. Code generation only works in C++

See also the Wikipedia page on Composite Structure Diagrams.

Example

view plain

```
01. // Example system representing constraint-based ping-
    pong example
02. // Demonstrates the composite Structure Diagram in Umple
03.
04. class Component1 {
05.   // Relay-Port
06.   public in Integer pIn1;
07.   public out Integer pOut1;
08.   pIn1 -> pOut1;
09.
10.
```

```
  [pIn1]
  active increment {
    pOut1(pIn1 + 1);
  }

  [pOut1]
  active logOutPort1 {
    cout << "CMP 1 : Out data = " << pOut1 << endl;
  }

}

class Component2 {
  public in Integer pIn2;
  public out Integer pOut2;
  pIn2 -> pOut2;

  [pIn2, pIn2 < 10]
  active stop {
      pOut2(pIn2 + 1);
  }

  [pOut2]
  active logOutPort2 {
    cout << "CMP 2 : Out data = " << pOut2 << endl;
  }
}


class Atomic {
Component1 cmp1;
Component2 cmp2;
Integer startValue;

  after constructor {
    cmp1->pIn1(startValue);

  }


  cmp1.pOut1 -> cmp2.pIn2;
  cmp2.pOut2 -> cmp1.pIn1;
}

```

56.

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

```
primitiveDefinition : primitive [name] { [[primitiveBody]]* }

portMultiplicity : [!portMultiplicity:\[[0-9]+\]]

typedPortName : [~type]? [~portName] [[portMultiplicity]]?

portDefinition : [[portClass]] | [[portDeclaration]]

// Port Connectors
portBindingDefinition :
  ( [~fromClassname] .
  )? [fromPort] ->
  ( [~toClassname] .
  )? [toPort] [[bindinHandler]]

bindinHandler : { [**code] } | ;

// Port Watchlist Definition
portWatch- :
  ( [[constraintList]] | [[comment]] | [[reqImplementation]]
  )*

// Active Method Definition
activeMethodDefinition : [[portWatch]]? [=modifier:public
    |protected
    |private]? [type]? [=activeType:atomic
    |synchronous
    |intercept]? [=active] [[activeMethodDeclarator]]
[[activeMethodBody]]+

activeMethodDeclarator : [~methodName] [[parameterList]]?

//activeMethodBody : [[activeDirectionHandler]] { ( [[activeTrigger]] )*
[**code] }
activeMethodBody : [[activeMethodBodyContent]]
[[inverseDirectionHandler]]?
```

```
activeTrigger : [[hitchConstraint]]? [[constraintList]]? [=trigger:/]
[[activeTriggerBody]] [[thenDefinition]]? [[resolveDefinition]]?

activeTriggerBody- :
  ( [[deferredList]] | [[activeTriggerDefinition]]
  )

deferredList : [ [[activeTriggerDefinition]]
  ( , [[activeTriggerDefinition]]
  )* ]

activeTriggerDefinition- : [[anonymousTriggerBody]] | [[invoke]]

thenDefinition : .then ( [[anonymousTriggerBody]]? )

resolveDefinition : .resolve ( [[anonymousTriggerBody]]? )

hitchConstraint : [=clause:after|poll] ( [timer] )

constraintList : [ [[basicConstraint]]
  ( , [[basicConstraint]]
  )* ]

basicConstraint- : [[timeConstraint]] | [[messageConstraint]] |
[[constraint]]

timeConstraint : [=clause:latency|period|timeout] ( [timer] )

messageConstraint : [=clause:priority] ( [priorityValue] )

invoke :
  ( [~classname] .
  )? [name] (
  ( [parameter]
  ( , [parameter]
  )*
  )? ) ;

anonymousTriggerBody : { [**code] }
```

Model Oriented Tracing Language (MOTL)

Model Oriented Tracing Language (MOTL) allows developers to specify tracing at the model level.

Historically, developers have traced software using various techniques:

- Adding print statements or some other special debug statement to code, often with conditional compilation so it can be switched off when the final system is compiled.
- Using tracepoints or breakpoints in a debugger.
- Using a tool Like DTrace that specifies a trace script to dynamically patch live running code.
- Using a tool like LTTNG to insert tracepoints.

However, in software generated from UML, these techniques can only be used effectively on *generated code*. They require understanding the structure of that generated code; furthermore trace directives need replacing when the code is replaced. Tracing thus occurs at a level of abstraction below the level at which the system is implemented.

MOTL is designed to allow tracing with direct reference to UML model elements (state machines and their states and events, associations and attributes). MOTL specifies how to inject trace code into generated code for UML constructs.

MOTL syntax is designed to be be usable independently of Umple. For example, trace statements could be used to inject tracing into code generated from a UML model specified in Papyrus. The current implementation of MOTL focuses on tracing Umple-generated code. In order to trace a UML model created by an arbitrary UML tool, it is currently necessary to have the model from that tool converted to Umple first.

MOTL can be used to generate tracing directives for different types of tracers. At its simplest level, if the phrase 'tracer Console;' is used (or no tracer statement is included), then trace output is directed to *standard error*. Generation of tracing for DTrace and LTTNG is under development.

MOTL can be used in two modes:

- As scripts that are independent of the model. These appear like DTrace scripts. When used with Umple, they are merged into an

Umple model using Umple's mixin capability.
- By directly annotating model elements.

The following pages describe how to use MOTL. Many aspects of MOTL are under development. Some future pages for tracing features under development cam be found in Umple wiki pages.

The MOTL project has been sponsored by NSERC, Ericsson and Defence Research and Development Canada, in the Distributed Multi-Core Tracing Project managed by Ecole Polytechnique de Montreal

Tracing Basics

Simple tracing is specified using trace directives, which all start with the word 'trace'. This is normally followed by the UML entity to trace (attribute, association, state, etc.).

Beyond, this, other clauses can be added to limit tracing to certain conditions, to switch on or off tracing in certain situations, and to specify data that will be output.

Each 'trace' statement emits exactly one trace record when something occurs to a matching UML entity; by default it is only *changes* in value that trigger emission of a trace record, but later pages describe how accesses to the value can also be traced. The format of the trace record will depend on the tracer being used but by default will contain the name of the entity and the new value of that entity.

The following are extremely simple example of tracing using MOTL on a UML Integer attribute. The default console tracer is used, so output will be sent to standard error.

Example

view plain

```
01. @SuppressWarnings("deprecation")
02. class Student
03. {
04.   Integer id;
05. 
06.   // Whenever the id attribute is changed,
07.   // emit a trace record
08.   // The record will have the format id=value
09.   trace id;
10. 
11.   // The following Java main program can be used
12.   // to demonstrate this in operation
13.   // This will output a record when id is set to 9.
14.   public static void main(String [ ] args) {
15.     Student s = new Student(6);
16.     s.setId(9);
17.   }
18. }
19. 
```

```
20.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. @SuppressWarnings("deprecation")
02. class Student
03. {
04.   lazy Integer id;
05.   lazy name;
06.
07.   // It is possible to specify multiple
08.   // UML entities to trace as follows
09.   // It is also possible to indicate interesting
10.   // information to record in addition to the
11.   // value of the element being traced
12.   trace id, name record "interesting behavior";
13.   trace id record "Even more interesting";
14.
15.   // Traces will be triggered for each call to a 'set' method
16.   // and twice for id, since there are two trace statements for id
17.   public static void main(String [ ] args) {
18.     Student s = new Student();
19.     s.setId(9);
20.     s.setName("Tim");
21.   }
22. }
23.
24.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Tracing Attributes

MOTL allows the tracing of attributes at the model level. Attribute tracing can occur whenever an attribute value is changed (i.e. setter is called) or/ and when the value is accessed (i.e. the getter method is called). Thus, attribute tracing can be made to occur any of the following modes:

- **set**: When the "set" keyword is specified before the traced attribute, tracing occurs when the attribute setter method is executed. This is also the default when no keyword is specified before the attribute
- **get**: When the "get" keyword is specified before the traced attribute, tracing occurs when the attribute setter method is executed
- **set,get**: When "set,get" is specified tracing occurs when both the getter and setter method are executed.

Example

view plain

```
01. // This traces setId() and setName()
02. @SuppressWarnings("deprecation")
03. class Student
04. {
05.   Integer id;
06.   String name;
07.   trace id;
08.   trace set name;
09. }
10.
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // This traces  getId() method
02. @SuppressWarnings("deprecation")
03. class Student
04. {
05.   Integer id;
06.   trace get id;
```

```
}
// @@@skipphpcompile See issue 596 (PHP tracing causes issues


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
// This example traces getId() and setId()
@SuppressWarnings("deprecation")
class Student
{
  Integer id;
  trace set,get id;
}


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

traceDirective : trace [[Prefix]]? [[traceEntity]] [[Postfix]] ;

traceEntity- : [traceEntity]
((
)?
()
)?
(, [traceEntity]
((
)?
()
)?
)*

Tracing State Machines

State machines are representations of system behaviour. States can have entry or exit actions and do activities; they can also be composite (i.e. nested) or concurrent. MOTL provides modellers with the capability of specifying tracing for all these modelling elements.

- **State machine**: State Machines can be traced as a whole, considering all states, nested states, events, etc.
- **State**: Tracing a state involves the tracing of its entry/exit actions, incoming/outgoing events, and do-activities. In case of composite states, all nested states will be traced accordingly.
- **Event**: A targeted event can be traced specifically. Tracing an event records previous state before the triggering of the traced event with the resulting new state after the transition.

Example

view plain

```
01. @SuppressWarnings("deprecation")
02. class LightBulb
03. {
04.   Integer v = 0;
05.   status
06.   {
07.     On {
08.       entry / { setV(1); }
09.       flip -> Off;
10.     }
11.     Off {
12.       entry / { setV(2); }
13.       flip -> On;
14.     }
15.   }
16.   // trace whenever we enter/exit state On
17.   trace On;
18.   // trace whenever we exit state Off and
19.   // report value of attribute v
20.   trace exit Off record v;
21. }
22. // @@@skipphpcompile See issue 596 (PHP tracing causes issues
23.
```

Load the above code into classic UmpleOnline

Load the above code into UmpleOnline V2

Another Example

view plain

```
01. @SuppressWarnings("deprecation")
02. class LightBulb
03. {
04.   status
05.   {
06.     On {
07.       flip -> Off;
08.     }
09.     Off {
10.       flip -> On;
11.     }
12.   }
13.   // trace any triggering of event flip
14.   trace flip;
15. }
16. // @@@skipphpcompile See issue 596 (PHP tracing causes issues
17.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // this example will trace all states of
02. // State machine GarageDoor
03.
04. @SuppressWarnings("deprecation")
05. class GarageDoor
06. {
07.    // UML state machine digram for a Garage door,
08.    // written in Umple
09.    status {
10.       Open { buttonOrObstacle -> Closing;  }
11.       Closing {
12.         buttonOrObstacle -> Opening;
```

```
13.         reachBottom -> Closed;
14.       }
15.       Closed { buttonOrObstacle -> Opening; }
16.       Opening {
17.         buttonOrObstacle -> HalfOpen;
18.         reachTop -> Open;
19.       }
20.       HalfOpen { buttonOrObstacle -> Opening; }
21.     }
22.     // trace whole state machine
23.     trace status;
24.   }
25.   // @@@skipphpcompile See issue 596 (PHP tracing causes issues
26.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

traceDirective : trace [[Prefix]]? [[traceEntity]] [[Postfix]] ;

traceEntity- : [traceEntity]
(
(
)?
()
)?
(, [traceEntity]
(
(
)?
()
)?
)*

Tracing Associations

Associations describe the relationships between class instances; role names are used to clarify the relation at both ends of an association. MOTL allows the tracing of associations by referring to the role names. Tracing can occur whenever an association link is added or deleted at run time:

- **add**: When the "add" keyword is specified before the traced assoc role name, tracing occurs when an association link is being added.
- **remove**: When the "remove" keyword is specified before the traced assoc role name, tracing occurs when an association link is being deleted.

Example

view plain

```
01. // This example will record any addition/removal
02. // to supervisor assoc link
03.
04. class Student {
05.   Integer id;
06. }
07.
08. @SuppressWarnings("deprecation")
09. class Professor {
10.   1 -- * Student supervisor;
11.
12.   trace supervisor;
13. }
14. // @@@skipphpcompile See issue 596 (PHP tracing causes issues
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // This example will record any addition to
02. // assoc link 'supervisor'
03.
04. class Student {
```

```
  Integer id;
}

@SuppressWarnings("deprecation")
class Professor {
  1 -- * Student supervisor;

  trace add supervisor;
}
// @@@skipphpcompile See issue 596 (PHP tracing causes issues

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
// This example will record any removal of any
// supervisor assoc link

class Student {
  Integer id;
}

@SuppressWarnings("deprecation")
class Professor {
  1 -- * Student supervisor;

  trace remove supervisor;
}
// @@@skipphpcompile See issue 596 (PHP tracing causes issues

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

traceDirective : trace [[Prefix]]? [[traceEntity]] [[Postfix]] ;

```
traceEntity- : [traceEntity]
 ((
 )?
 ()
 )?
 ( , [traceEntity]
 ((
 )?
 ()
 )?
 )*
```

Tracing Constraints

Constraints can be imposed upon the traced UML entity to limit the scope of tracing to when a certain condition is true. MOTL provides a set of constraints that can be classified into a prefix and postfix constraints.

- **Condition**
  Conditions consist of a left hand side, Boolean operator and a right hand side. Either side can refer to a UML attribute, role name, state, state value or constant. PreConditions are evaluated prior to the operation being traced, while PostConditions are the opposite.
- **Number of Occurrences**
  Trace will be recorded for a specific number of appearances, then tracing stops.
- **Record clause**
  Record clause postfix gives the flexibility to record another model element or a specific string.
- **Life Timeline**
  There are two cases. (1) trace the UML construct from its creation and stop when the trace condition is satisfied. (2) Tracing will begin only after the trace condition is satisfied and will then continue nonstop.

Conditions and Number of Occurences

view plain

```
01. @SuppressWarnings("deprecation")
02. class Student
03. {
04.   name;
05.   Integer id;
06. 
07.   // trace code is injected in attribute setter
08.   // with specified precondition
09.   trace name where [name == "john"];
10. 
11.   // trace code is injected in attribute setter
12.   // with specified postcondition
13.   trace id giving [id > 9000];
14. 
15.   // this example traces attribute name whenever
16.   // its value is changed for 5 times
17.   trace name for 5;
18. }
```

```
19.
20.  @SuppressWarnings("deprecation")
21.  class Professor {
22.    name;
23.    1 -- * Student supervisor;
24.
25.    // trace code is injected in attribute setter
26.    // with role name cardinality condition
27.    trace name where [ supervisor cardinality > 10];
28.  }
29.
30.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Record Clause

```
view plain

01.  @SuppressWarnings("deprecation")
02.  class Student
03.  {
04.    name;
05.    Integer id;
06.    // trace attribute id and record provided string
07.    // message as part of  the trace output
08.    trace id record "i am tracing attribute id";
09.
10.    status {
11.      s1 { e1 -> s2; }
12.      s2 {}
13.    }
14.    // trace whenever we enter/exit state "s1" and
15.    // record attribute name
16.    trace s1 record name;
17.  }
18.
19.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Life Timeline

view plain

```
@SuppressWarnings("deprecation")
class Student
{
  name;
  Integer id;

  // tracing of attribute name starts and doesn't
  // stop until condition becomes true
  trace name until [name == "john"];

  // tracing of attribute id until condition
  // becomes true and then always continue tracing
  trace id after [id == 234];
}


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

// A constraint is an expression optionally be surrounded in round brackets or negated
constraint- : [[constraintBody]] [[linkingOp]]? | [[constraintExpr]] [[linkingOp]]?

traceCondition : [=conditionType:where|until|after|giving]? [[[constraintToken]]]

traceFor- : for [traceFor]

Tracing Methods

Tracing non api methods is possible using MOTL. Entry and/or exit of methods can be traced.

- **Method Entry**: trace code is injected at the method entry.
- **Method Exit**: trace code is injected at method exit before return statements if exits.

Method Entry

view plain

```
01. // this example shows how to trace
02. // non-generated method entry
03.
04. @SuppressWarnings("deprecation")
05. class JavaMethod
06. {
07.   trace method();
08.
09.   int method( int x ) {
10.     x += 5;
11.     return x;
12.   }
13. }
14.
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Method Exit

view plain

```
01. // this example shows how to trace generated
02. // method exit. In case of methods with a return
03. // statement, trace code is injected before
04. // the return
05.
06. @SuppressWarnings("deprecation")
07. class JavaMethod
08. {
09.   trace exit method();
```

```
10.
11.   int method( int x ) {
12.     x += 5;
13.     return x;
14.   }
15. }
16.
17.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

traceDirective : trace [[Prefix]]? [[traceEntity]] [[Postfix]] ;

Tracers

MOTL sets the Console to be its default tracer. However, it provides a set of potential tracers that will have an impact on how tracing is injected and how its collected. Modellers can control the model tracer using a tracer directive.

MOTL tracers can be classified into two main categories: Built in tracers and third party tracers. The main difference between these two categories is that the first category of tracers doesn't require any additional jars imported into your model generated code, while the later requires jars specific to each tracer.

1. (Built in tracers)
    - **Console**: trace output will be directed to system error.
    - **File**: trace output will be directed to trace log file.
    - **Java Native Logging**: Java provides a logging API implemented in its platform. MOTL provides the capability of injecting trace code using Java native logging API.

2. (Third party tracers)
    - **log4j**: a well known java tracer. In MOTL, if log4j is selected as a tracer, log4j trace points will be in injected. In addition, an xml file (log4j2.xml) configuration file will be generated.
    - **Linux Trace Toolkit Next Generation (LTTNG)**: Lttng is a well known tracer that allows kernel and userspace tracing. It targets C/C++ programs, however, tracing of Java programs using Lttng is possible through Java Native Interface (JNI). To execute Java programs instrumented with lttng trace points, (liblttng-ust-java.jar) is required.

File tracer

view plain

```
01. // this example generates traces using the
02. // File tracer
03.
04. tracer File;
05.
06. @SuppressWarnings("deprecation")
07. class Student
08. {
09.   Integer id;
10.   trace id;
```

```
11. }
12.
13. // The suppressWarnings annotation allows this example
14. // to keep working as of Java 19 as per Umple issue £2018
15.
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Java native logging API

view plain

```
01. // this example generates traces using the Java
02. // native logging API
03. tracer JavaAPI;
04.
05. @SuppressWarnings("deprecation")
06. class Student
07. {
08.   Integer id;
09.   name;
10.   // trace id with default logging level info
11.   trace id;
12.   // trace name to logging level severe
13.   trace name logLevel severe;
14. }
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Lttng tracer

view plain

```
01. // this example generates tracespoints using
02. // the Lttng tracer
03. tracer Lttng;
04.
05. class Student
06. {
```

```
  Integer id;
  trace id;
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

log4j tracer with default log4j xml configuration file

view plain

```
// this example generates traces using the
// log4j tracer
// Default log4j xml configuration file is
// generated (log4j2.xml)
tracer log4j;

class Student
{
  Integer id;
  name;

  // trace attribute id with log4j level set to (info)
  trace id logLevel info;

  // trace attribute id with log4j level set to (error)
  trace name logLevel error;
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

log4j tracer with customized log4j xml configuration file

view plain

```
// this example generates traces using the
// log4j tracer
//
```

```
04. // Customized log4j xml configuration file
05. // is generated (log4j2.xml)
06. //
07. // Root logger level will be debug
08. //
09. // Info logging level will be written into
10. // console output
11. //
12. // Error logging level will be written into file
13. tracer log4j root=debug info = console error = file;
14.
15. class Student
16. {
17.   Integer id;
18.   name;
19.
20.   // trace attribute id with log4j level set to (info)
21.   trace id logLevel info;
22.
23.   // trace attribute id with log4j level set to (error)
24.   trace name logLevel error;
25. }
26.
27.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Syntax

Trace Output

Specifying trace directives at the model level will inject trace points in source code generated from the model. MOTL was designed to collect a wide range of information when tracing is triggered at the execution time.

Collected information includes current system time, thread ID, Object hash code, etc. MOTL tracers will output an initial header to indicate each traced field name.

Using UmpleOnline

The following describes how to use the UmpleOnline tool accessible online at try.umple.org

For information about writing Umple models or code in UmpleOnline, look at the Getting Started section of this user manual, or any other topic in the left column of this page, including information about defining classes, attributes, associations and state machines.

Basic use of UmpleOnline

- **To explore examples of Umple**, In the centre pane, select a pre-prepared example from the 'TOOLS' menu. You can also directly access these examples using a URL whose last component is the name of the example. For example click here to jump to the 2DShapes example: https://try.umple.org/?example=2DShapes

- **To restart UmpleOnline** (you will lose any work you have not saved), click 'Start Over' under the 'SAVE & LOAD' menu.

- New in 2026:
  - We have transitioned to automatic layout of diagrams. You can still manually lay out diagrams by clicking the 'E' at the top left.
  - You easily select a subset of classes by specifying a pattern in the Show & Hide field in the Tools menu.
  - We have a beta AI feature that allows you to generate models from requirements.

Loading and saving

- **To save your work**, there are several approaches:
  1. **Create a bookmarkable URL**. To do this, click 'Create Bookmarkable URL' at the top near the centre 'Save as URLg' in the centre pane under 'SAVE LOAD'. Then copy or bookmark the URL that appears in your browser's URL bar.

     Note that from this point on you can keep editing the Umple code or diagram, and your work will continue to be saved at this URL. You are thus saving your work in the 'cloud'. You can embed the URL in other pages to document your product. Note that at the current time, anyone who knows your URL can also edit your work (which enables collaboration). If

someone messes up the saved contents at that URL, you can get back an earlier version using 'Restore Earlier Version' (see below), but if you forget your URL there is no way to retrieve it. Saved URLs are also only saved for one year since their last edit. Therefore it is also suggested that if this is a concern, you use saving mehods 2 or 3.

2. **Create a URL that embeds the entire diagram**. This is only guaranteed to work for small diagrams, since some browsers such as older versions of Internet Explorer limit the number of characters in a URL.

   To do this, click on "Encoded URL" in the 'SAVE' menu, and copy the URL that appears in a separate window. You might want to use a URL shortening service such as bit.ly to shorten the URL before posting it or embedding it in another document. You also need to be aware that you are limited by certain browsers and URL shortening services to files with about 2000 characters or less, so test your URL once you have created it.

3. **Temporarily stash your current work in your browser.** You can click 'Store in Browser' in the 'SAVE' menu at any time. Later on, in the same browser, you can retrieve this model using 'Load from Browser' (and if you load by mistake, you can click undo from the Tools menu). This method is not for permanent saving, but can be useful for temporary backups to protect against power failure, browser crashes and so on.
4. **Download Files.** This is probably the safest approach. It will generate a zip file with the code.
5. **Copy and paste the text of important models somewhere else**. One thing to note is that if you just copy the visible text in the left hand pane, you will not be saving the *layout of the diagram*. If you want to save the layout too, you have to do one of the following:
   - Click on 'Popup Current' in the 'SAVE' menu. This will generate a popup that contains Umple code with the layout information. Copy and paste the contents of this window, and then close the window.
   - Check the box 'Layout Editor' in the 'OPTIONS' menu to view a separate window containing the layout, and then save the automatically-generated layout information in a file. You can actually edit this layout textually, but we

suggest you only do that if you are an expert since it us much easier to edit the layout from the right-hand graphical pane, as described later.

- **To go back to an earlier version of your model**, click 'Restore Earlier Version' in the 'SAVE & LOAD' menu. While you edit, UmpleOnline keeps earlier versions of your model on the server: a version is kept once you have made at least five changes and at least two minutes have passed since the previous one, and also just before you restore a version. Older versions are gradually thinned out: all versions from the last half hour are kept, then one every 5 minutes for versions up to 2 hours old, one every 10 minutes up to 12 hours, one every half hour up to 2 days, one a day up to 30 days, and one a month after that. Earlier versions are only kept for as long as the model itself.

  Select a version in the list to see what it contained, then click 'Restore this version'. Restoring can itself be undone, using the 'Undo restore' link that then appears at the top of the page, or, for a model with a single tab, using 'Undo' in the 'TOOLS' menu.

  If you have the same model open in more than one browser tab or window and change it in one of them, the others notice this when you return to them, and ask whether you want to load the latest version or keep the one they show. If you keep it, the newer version is saved first, so you can still restore it. Similarly, if a tab saves over changes made in another tab, it tells you, and lets you switch to the other tab's version.

- **To load Umple code into UmpleOnline** do one of the following:
  1. **Load a URL you saved using one of the saving methods 1 and 2 above.**
  2. **Put a .ump file on the web (saved from Eclipse or using saving method 3 above), and load it using the filename argument to Umpleonline.** The filename argument's syntax is ?filename=X, where X is the URL of the file containing the Umple file, *minus the leading 'https://'*. So, for example there is a .ump file in the repository at the following URL: https://raw.githubusercontent.com/umple/umple/master/cruise.umple/src/Umple.ump . This can be loaded into Umple using the following URL: https://try.umple.org/?filename=raw.githubusercontent.com/umple/umple/master/cruise.umple/src/Umple.ump
  3. **Paste Umple code into UmpleOnline.** Note however, that

you have to make sure the pasted file contains the layout information, as described under the saving methods above. it is suggested to drag and drop the file into the center pane to ensure the layout information is properly handled.

Note that in most of these cases, the user can edit the Umple, without affecting the original content. In other words, a copy is saved first. The only exception is that when you edit Umple saved using method 1, any changes are saved permanently.

Left Pane: Umple textual code

- Type any Umple code into this pane. Three seconds after you stop typing, your program will be interpreted by Umple, and Umple will attempt to draw a corresponding class diagram in the right-hand side. Just about the simplest Umple program you could type here would be class X {}
- **If you make a mistake**, you can click 'Undo' in the 'TOOLS' menu, repeatedly to 'back out' each change you made (or use your browser's undo keystroke). You can also redo changes in a similar manner.
- **You can hide this textual Umple pane entirely** by unchecking 'Text Editor' in the 'OPTIONS' menu.
- **If you are doing a lot of text editing**, and don't want to have the diagram generated all the time, you can check 'Manual Sync' in the 'OPTIONS' menu or uncheck 'Canvas' to hide the diagram entirely.
- **Syntax assistance:** As you type Umple text, the text editor will attempt to help you keep the correct indentation of your code. It will also highlight matching parentheses, and umple keywords. We intend to improve syntax assistance.
- **Jumping to a line in the text editor:** You can type a line number in the box above the textual pane and hit 'tab' to jump to that line. In most browsers you can also hit 'return'. If an error message is displayed below the browser, the line number causing the error will also appear. You can click on this line number to jump to the line in the text.
- **Collapsing blocks of text** You can click on the line number of any line that ends in ' {' to *collapse* that block of code. This can be used to temporarily hide the bodies of classes or methods that you are not interested in viewing. Note that currently there must be a space before the '{' for this to work.

Right Pane: UML diagram

This can be an editable class diagram, and The 'E' button at the top-left returns the display to this state as will ctrl-E. The instructions below refer to this format. This can also be a class diagram laid out using Graphviz (G button or ctrl-G), which is the default for many examples, or a state diagram (S button or ctrl-S). To change the Graphviz diagrams, you have to edit Umple textually, or use the gv_manual option in the show and hide menu to drag the diagram to your liking.

- **To add a** class, click on 'Class' in the 'TOOLS' menu, then position the cursor over the canvas. To add a whole series of classes, double-click on 'Class' then click repeatedly on the canvas. Press escape if you change your mind and you don't want to add a class.

- You can also use the text (left) pane to write Umple code that will appear on the canvas. If you do this, you will have to then lay out the diagram, since the layout algorithm is rather simple.

- **To rename a class**, click on the class name and type over it, then hit enter or click somewhere else. Alternatively edit it on the text (left) pane.

- **To reposition classes,** you can drag classes around to align them the way you like. You can also select a class and use the arrow keys to move it by a small amount in any direction. This is nice to carefully arrange classes. Note that the positions of your classes are saved in a hidden text pane on the left hand side. See the discussion above regarding showing this pane and saving the layout information

- **To add an** attribute, click in the 'Add More' box and type the attribute name. To give an attribute a type, follow the name by a colon, and then one of the Umple types (Integer, String, Date, Time, Boolean or the name of an Umple class). Note that this is UML diagram syntax for typed attributes which is not the same as the Umple textual syntax. You can click on an attribute at any time to change its name or type, or you can edit the attribute in the textual (left) pane. You can add a whole series of attributes by just hitting return after each one and typing the next one.

- **To delete an attribute**, click on the little red X next to it.

- **To add a** generalization (an inheritance relationship between classes), click on 'Generalization' in 'TOOLS' menu., and then click first on the subclass, then the superclass. If you double-click on Tools/Generalization you can then hook up a subclass to its parent,

and then the parent to a higher-level parent, etc. Press escape to cancel the process of adding generalizations..

- **To add an** association (a relationship that describes how instances of one class are to be connected to instances of another at run time), click on 'Association' in 'TOOLS' menu. Then click on one class to be connected, and then the second class. By default, the multiplicity of a new association is many-to-many (* -- *). To change this, or to add a name to the assocation, you currently have to edit the Umple code in the left (textual) pane.

- **To adjust the location of an association**, in other words how it is connected to each class, hover over the association until you see little squares at its ends, then click. The squares should change to a darker colour. Select a little square and move it to a new location on the class. If an association crosses a class's border, just move the class a little. Note that the location of the association end is saved with the layout information of the diagram.

- **To delete a class, generalization or association** click on 'Delete' in the 'TOOLS' menu and then select the item to delete.

- **To hide the 'add more' line in each class**, select 'Photo Ready' from the 'OPTIONS' menu. You can also click on any class to adjust its size a little. UmpleOnline is an excellent tool for preparing nice-looking diagrams ready for publication. Note that this works best in Firefox at the current time. You can also turn on pinch to zoom in the "show and hide" menu in tools, to more easily change your view of the diagram and to make the diagram more legible.

Adjusting the panes

- **To hide (or re-show) the graphical canvas**, click the 'D' button or use ctrl-D.

- **To hide (or re-show) the text editor**, click the 'T' button or use ctrl-T.

- **To adjust the size of the canvas** hover the cursor over either edge of the menu; an arrow will appear, allowing you to drag left or right to change the portion of the screen allocated to text or graphics .

Bottom Pane: Errors, Warnings and Generated Results

- **Messages:** As you edit your text or diagram, warnings or errors

may appear at the bottom of the page. Each warning or error message has a line number pointing you to the line in the code where the error originated. Each message also has a link to the user manual page describing the problem and how to fix it.

- **Generated code:** You can generate many types of output from your Umple model/code. Pick an output format from the pulldown menu under 'GENERATE' in the 'SAVE & LOAD' menu, then click 'Generate Code'.
  Multiple files of generated code are concatenated when they appear on the web page. A link is also provided to download a zip file with the code so you can easily compile and execute it.

Shortcuts

- **Common shortcuts for all computers:**
  - Control + E: Show the Editable class diagram (on some browsers, jumps to search bar instead)
  - Control + G: Show GraphViz class diagram (on some browsers, opens find and replace menu instead)
  - Control + S: Show GraphViz state diagram (on some browsers, tries to download the webpage)
  - Control + L: Show Composite Structure diagram (on some browsers, jumps to search bar)
  - Control + D: Show/Hide the diagram (on some browsers, favorites the webpage)
  - Control + A: Toggle attributes (on some browsers, selects all text)
  - Control + M: Toggle methods
  - Control + R: Toggle traits
- **Mac Specific Shortcuts:**
  - Command + Z: Undo text changes
  - Command + shift + Z: Redo text changes
  - Control + T: Toggle the text area
  - Control + N: Toggle the menu
- **Windows/Linux Specific Shortcuts:**
  - Control + Z: Undo text changes
  - Control + Y: Redo text changes
  - Control + Alt + Shift + T: Toggle the text area
  - Control + Alt + N: Toggle the menu

See here for the statement regarding **privacy and other risks when using UmpleOnline**.

UmpleOnline URL options

UmpleOnline allows you to append various options to its URLi, enabling startup in a certain state. These are preceded by an ampersand, except the first in a list which would be preceded by a question mark.

- ?nochrome: Hide the header of the UmpleOnline page (sometimes called the chrome) so only the model is visible.
- ?nodiagram: Don't show any diagram, in order to show the text only.
- ?diagramtype=GvClass: By default show graphviz class diagrams instead of editable diagrams.
- ?diagramtype=state: By default show graphviz state diagrams instead of editable class diagrams.
- ?notext: Don't show Umple text (doesn't have any effect if the diagram is also hidden).
- ?nomenu: Hide the middle menu. Either ?notext or ?nodiagram must also be present.
- ?readOnly: Turn off editing capability - use for display only.
- ?filename=xxxxx: Load the model xxxxx. xxxxx is a URL with the leading https:// left off.
- ?showlayout: Show a pane with information about where each class should appear in a class diagram. This is normally hidden (it cam be manually displayed from the Other Options menu). This can be used to finely tune the layout; after adjusting the layout the user should then make a small change to the main text (e.g. just add and delete a space) to force the diagram to update).
- ?displayoptions=xxxxx: Pre-filter the diagram or request a specific layout algorithm, a specific filters or a mixset activation. The options specified in the xxxxx here are the same that can be specified in the "Show and Hide" text field. Some options only work with specific diagram types such as gvClass. Examples include gvmanual to enable manual layout, or a class name to filter on this class. You can separate multiple items with spaces, but the code %20 would be safer in the URL format than using a space. If you are including an integer as part of the displayoptions, to include connected classes in a filter, then place this first.

- ?generateDefault=xxx: Set the generate menu to by default show a particular choice. The default is Java. Other options in include python, cpp, ruby, php, alloy

Umple in Presentations and Documents

Embedding a pure diagram in a web page

Using **&nochrome&notext&nomenu&readOnly** allows you to embed a pure diagram in a web page. Using **&nochrome&nodiagram&nomenu&readOnly** allows textual listings in any other web page. Using **&nochrome&notext** allows a UML diagram editor to be embedded.

For example, Click here to see a document with an embedded iFrame containing an editor opening on the RoutesAndLocations example. The chrome and text is hidden.

For a read-only version, with ability to generate code, use https://try.umple.org/?example=RoutesAndLocations&notext&nochrome&readOnly

Coloring model elements

Umple has a special directive that allows you to highlight certain elements (currently only classes) in UmpleOnline and GraphViz output. Simply write 'displayColor xxx;' in a class where xxx is any html colour such as "red" or "#BBCCFF". UK spelling displayColour also works.

The valid set of colors can be found online at the W3C official site.

Setting up Tasks

The following describes how to set up ***tasks*** in UmpleOnline. A task can be an assignment in a course that some students are asked to submit, or an experiment in modeling conducted by a researcher.

This is currently a beta feature. Please raise issues to help us improve it. Some aspects may not work perfectly yet, although it works and can be used in courses. We expect to take this out of beta in late September 2020.

People involved: We refer to the person *creating* a task (professor, teacher, researcher) as the **requestor**. We refer to the people doing the tasks (students, research subjects) as **responders**

Task names: Each task in UmpleOnline has a unique name that can contain alphanumeric characters, underscores and dots. Names are case insensitive and cannot contain spaces or special characters. Requestors should select a meaningful name. For example an assignment in a course at the University of Ottawa could have task name *UOtt_2020_SEG2105F.A1* Responders would use this name to find the task or else they can be given a URL to be able to respond to the task (described below).

What information does the requestor provide to set up the task?

When setting up a task, the requestor provides the following information:

- **The requestor's name.** This might be their personal name, the name of the course, the name of the institution, or a combination. Responders will see this at the top of their task instructions.
- **Instructions.** These are written in Markdown format. So for example, to make a heading appear in the instructions, start a line with =, or more equals signs for lower-level headings. To make bullet lists start the line with *. To emphasize text in a line use * before and after. Use ``` before and after blocks of code. URLs will generate links to open a page in another browser tab.

  It is best to keep instructions relatively short as they take up space

on the screen that will be unavailable for modeling, unless the responder asks for them to be hidden. For lengthy instructions, we suggest providing a link to another page. Instructions should include, if appropriate, a request for the responder to include their name and/or ID in the model. UmpleOnline is (at the current time) a login-free system, so responses will be anonymous by default.

- **An Umple model** (optional). The requestor can populate Umpleonline in the normal way (i.e. text and diagram) with a model of any level of sophistication, including one with many tabs (Umple files). Initiating the request with a pre-prepared model can be useful when the instructions are to make improvements, or fix bugs, or add new features. Both requestors and responders have the power to use all the features of UmpleOnline when creating or responding to a request.

  If no Umple model is entered, then responders will have to start modeling from scratch, and will be presented with an empty text pane and diagram.

- **A completion URL** (optional) After the responder completes and submits the task, they will be taken to this URL, and the taskname and the URL of the reponse will also be passed to the URL as additional arguments. This could be a URL in a custom tool, or in a survey tool like SurveyMonkey. More detailed explanation for how to do this is in a separate manual page. The completion URL can be used to ask questions in an experiment after the task is done, simply to record the response in a database, or to ask test questions of students.

  If no completion URL is given, the responder is returned to their model after submission, but it becomes read-only.

- **Is the task is an experiment requiring detailed logging?** (this feature is still currently under development) The requestor can select a checkbox requesting that detailed logs of every stage of the modeling process be recorded, as performed by each responder. This is most useful for those conducting experiments. Researchers ought to have received ethics approval to run experiments that track data at this level of detail.

What is the process of setting up a task?

1. Anyone can set up a task. Select the **SAVE & LOAD** menu in

UmpleOnline From there click on **Create a Task** which appears below the TASK heading.

2. The task creation pane will appear, allowing the requestor to enter the information described above (requestor name, instructions, model etc.). It is very important at this point to *bookmark* the URL that appears, or to copy and paste it somewhere safe. For security reasons it will not be possible for the requestor to edit their task or retrieve responses unless they know the randomly-generated URL.

3. When task preparation is complete, or if the requestor wants to save their work, they select **Submit Task**

4. The requestor can continue to edit the task, selecting **Save Changes to this Task** at any point. Any information can be edited except the task name.

5. To test the task and see how it will appear to responders, the requestor can select **Launch Participant URL in a new tab**. Any work done will be recorded as an actual response, unless deleted. Another alternative for testing is to click **Copy Participant URL** and to paste this into the URL bar of the browser.

How do requestors ask responders to do a task?

There are two ways:

- Give the responders the task name and ask them to go to the **SAVE & LOAD** menu, to select **Load a Task** and enter the task name.

- Give them the URL obtained from selecting **Copy Participant URL**; clicking on this URL will create a new individual response (a new secret URL) for the person clicking. Note you can't give them the URL that appears from selecting Launch Participant URL in a new tab because that is an individual response from *you* and you would be giving all participants access to the same response!

Note: If the requestor edits the model of a task after a responder creates a response, the responder will not have access to the revised model. However, responders will have access to revised instructions.

Requestors can distribute the task name or the participant URL via email, a learning management system, or any other means.

How do responders respond to a task?

This information is in a separate manual page intended for responders. The tool is designed, however, such that responders should not need to access any help. They just read the instructions provided by the requestor, do the task, and submit.

There are a few points responders should be aware of in certain circumstances.

1. Responders would benefit from bookmarking the URL of their response if they want to be able to continue their work in several sessions before submitting (if they lose their URL they will not be able to recover their work easily, although see below for alternatives).
2. Responders ought to provide their name or other ID in their response as a comment, if the requestor wants this. So tell them to do so in your instructions.
3. Responders need to choose **Submit task** when they are done.
4. Responders cannot go back to edit their response after submitting (they can look at it, but it will be read only).
5. A responder can delete their work by clicking on **Cancel this task response** before submission (but not afterwards).

How can the requestor obtain submissions?

Submissions can be obtained in the following ways. The first two of these will work even before submission.

- Responders can be asked to simply send the URL of their submission to the requestor (e.g. in an email, or in some other tool).
- The responder can click on the link **Request all Directories as a zip under this task** from the task editing pane. This will work even if a responder has forgotten to submit, so could be useful in an emergency where a a responder forgets their response URL or closes their browser.
- If the requestor has set up a **Completion URL** the response URL will be passed to this. The requestor can use their own software, or a tool like SurveyMonkey to gather the URLs along with answers to other questions.

Additionally, a responder could in some cases want to copy and paste their Umple model (text or diagram) into a separate file, and requestors might ask them to do this in the instructions

If a responder loses the URL of their submission, they might be able to obtain their model by clicking on the 'Restore Saved State' link that appears for a short period when UmpleOnline is relaunched. This does not, however, restore their submission: they would have to create a new submission and copy and paste the retrieved model into it.

How long do submissions persist?

It is intended that submissions will persist for a year after they are last edited, but it is recommended that requestors and responders not rely on this, and back up their Umple code somewhere else.

See here for the statement regarding **privacy and other risks when using UmpleOnline**.

Using Completion URLs

A completion URL is one of the data items that an task requestor can provide when setting up a task in UmpleOnline.

So for example, if the completion URL is https://www.surveymonkey.ca/r/UmpleOnlineDefaultTaskTest, the task name is abc and the responder's submission URL is https://cruise.eecs.uottawa.ca/umpleonline/umple.php?model=task-abc-2008262stdov293 then upon completion the URL https://www.surveymonkey.ca/r/UmpleOnlineDefaultTaskTest?task=abc&url=https://cruise.eecs.uottawa.ca/umpleonline/umple.php?model=task-abc-2008262stdov293 would be submitted.

The above example uses SurveyMonkey as a tool to gather data following submission. Within SurveyMonkey you can create a survey and give it 'custom variables' called 'task' and 'url'. When you save your survey data as individual responses in an Excel file, the task and URL information will appear as columns in the Excel file.

SurveyMonkey is not the only tool that can be used for this purpose. Any tool that can be set up to accept task and url arguments would work. Even a tool that just accepted a URL argument would work.

Responding to Tasks

If you are asked to complete a *task* in Umple as part of an experiment or a course assignment, the following are some tips you might want to consider.

- If you are given a URL to load a task, then just click on it (or copy and paste it). Make sure the URL start with https://cruise.eecs.uottawa.ca/umpleonline, or else is a URL in your own university or company that you trust.
- If you are instead given a taskname, you can load the task as follows: Go to UmpleOnline; open up the menu in the center that says **SAVE & LOAD**; and click on **Load a Task**. This will display a field where you can type the task name, and then hit **Load Task**.
- After you have loaded the task you will see a set of instructions at the top. These have been set by a task requestor; simply do what the instructions say. Normally this involves writing Umple models. See the rest of this user manual for information about how to do that, including the page about UmpleOnline itself.
- If the instructions take up too much space use the relevant buttons to open them in a new window, and hide them from the main UmpleOnline window.
- Responses to a task are **anonymous unless you explicitly put your name and/or ID in your Umple model**. Add your name as a comment in the model if this is a course assignment or if the instructions tell you to.
- If this is a class assignment or other work that you might need to do in several steps over several hours, you should **save the URL of your individual response. Either bookmark it using your browser, or copy and paste it into another tool like a text editor**. This protects against accidentally closing your browser, or your computer crashing. Each person's response to a task has a separate secret URL, and if you lose track of it, it will not be easy to recover it (the requestor might be able to get your work back by looking at all the responses, but it would be a multi-step process for them and they would have to search for a submission that has your name in it)
- To submit your work, make sure you are completely finished, and then select **Submit Your Work**. At this point your work will become read only (you will not be able to make more changes) so don't

submit too soon. An alternative way to submit would be to send the URL of your individual response to the requestor; do this if they ask. Note that the requestor is able to see your response even before you submit or if you forget to do so.

- If you want to throw away your work before submitting, select **Cancel this task response**.

VSCode

Umple for VS Code

A VS Code extension for the Umple modeling language. Provides IDE features for `.ump` files.

Umple enables textual description of data models (class diagrams) and state models (finite state machines) with embedded code in Java, Python, C++ and other languages, along with generation of high quality code in those languages. Umple enables extensive analysis of models, as well as generation of diagrams, example data and many other outputs. Special features of Umple include specification of product lines (using mixsets), code injection (aspects), code snippet management (traits) and so on. Umple can scale to systems of any size, but its codebase will always be much smaller than software written directly in target languages due to Umple eliminating much boilerplate code. Umple can complement LLM-generation of code: You can arrange for your AI to generate Umple.

This plugin is listed in the VSCode Marketplace

Features of the Umple Plugin

Language Intelligence (via LSP)

- **Diagnostics** — Real-time error and warning detection via the Umple compiler
- **Go-to-definition** — Jump to classes, interfaces, traits, enums, attributes, methods, state machines, states, associations, mixsets, and requirements
- **Find references** — Semantic reference search across all reachable files
- **Rename** — Safe rename of symbols across all references
- **Hover** — Contextual information for symbols
- **Inlay hints** — Editor-only inferred type annotations for untyped attributes when VS Code inlay hints are enabled
- **Document symbols** — Hierarchical outline of classes, state machines, states, attributes, methods
- **Code completion** — Context-aware keyword and symbol suggestions
- **Formatting** — AST-driven indent correction, arrow spacing, declaration assignment spacing, structural comma spacing, blank-line normalization
- **Syntax highlighting** — TextMate grammar baseline plus LSP semantic tokens generated by `umple-lsp`
- **Cross-file support** — Transitive `use` statement resolution and cross-file diagnostics

- **Import error reporting** — Errors in imported files shown on the `use` statement line
- **Runtime bootstrap** — Automatically downloads `umplesync.jar` on activation and offers a one-click recovery download if it is missing later

Compile

Generate target language code from Umple source.

- **Command:** `Umple: Compile` (also accessible from the Umple menu button)
- **Target languages:** Java, PHP, Ruby, Python, C++, RT-C++, SQL
- **Status bar:** Shows the current target language (e.g., `Umple: Java`) — click to switch. Only visible when editing `.ump` files.
- **Output:** Results appear in the "Umple" output channel with parsed errors and warnings

Diagrams

View diagrams (UML and others) generated from your Umple model.

- **Command:** `Umple: Show Diagram` or `Cmd+Shift+U` (`Ctrl+Shift+U` on Windows/Linux)
- **10 diagram / view types** in a dropdown selector:
  - UML Class Diagram
  - Class + Trait Diagram
  - Entity-Relationship Diagram (equivalent to the class diagram)
  - UML State Machine Diagram (if any state machines are present)
  - Feature Diagram (if any mixsets are specified and activated with a require statement)
  - UML Instance Diagram (random; each time it is generated, different instances appear)
  - State Tables (equivalent content to the state machine diagram, but different format)
  - Event Sequence (random; each time it is generated a new sequence is created)
  - Metrics
  - Plain Requirements Doc (HTML view — structured rendering of `req` declarations and `useCase` blocks; degenerate placeholder for class-only models)
- **Click-to-select:** Click classes, traits, interfaces, states, or transitions in the diagram to jump to their source location
- **Layout engine:** Choose between dot, circo, neato, fdp, and osage (configurable via `umple.diagramLayout` setting or the in-panel

dropdown)
- **Diagram filter:** Filter diagram content by class/trait name, matching UmpleOnline filter syntax
- **Zoom controls:** +/−/Fit/1:1 buttons and Ctrl/Cmd+scroll wheel (20%–500%)
- **Export:** Save as SVG or PNG
- **Live auto-refresh:** Diagrams update automatically as you edit (1.5s debounce)
- **Theme-aware:** Diagram colors adapt to your VS Code light/dark theme
- **Cross-file support:** Diagrams resolve `use` imports to include referenced files

Snippets

Code snippets for common Umple patterns. Type the prefix and press Tab:

Prefix	Description
`class`	Class with attributes
`sm`	State machine skeleton
`assoc`	Inline association
`isa`	Inheritance (`isA`)
`interface`	Interface block
`trait`	Trait block
`enum`	Enum attribute
`use`	Use (import) statement

Umple Menu Button

An Umple logo button appears in the editor title bar when editing `.ump` files. Click it to access:

- Compile (with current target language)
- Show Diagram
- Change Target Language

Requirements

- **Node.js 20+** (tested on 20 and 23)
- **Java 11+** (optional — only needed for diagnostics, compilation, and diagrams)

Installation

Install from the VS Code Marketplace, or build from source:

```
view plain
01. # Side-by-side checkout: this repo currently pins umple-lsp-server to a local
02. # tarball produced by `npm pack` in the umple-lsp monorepo. So you also need
03. # umple-lsp cloned as a sibling.
04. git clone https://github.com/umple/umple-lsp.git
05. git clone https://github.com/umple/umple.vscode.git
06.
07. # Build the server first and pack it
08. cd umple-lsp
09. npm install
10. npm run compile
11. cd packages/server
12. npm pack          # produces umple-lsp-server-X.Y.Z.tgz
13.
14. # Then build the extension. package.json's `umple-lsp-server` dep points at
15. # the .tgz path; if you packed a new version, edit package.json to
16. cd ../../../umple.vscode
17. npm install
18. npm run compile
```

To package as `.vsix`:

```
view plain
01. npx @vscode/vsce package
```

Alternative: change the `umple-lsp-server` dep in `package.json` from the `file:` tarball path to a registry version like `^0.4.3`, then `npm install` pulls from npm. Cleaner long-term; requires the matching server version already published.

Configuration

Setting	Type	Default	Description
`umple.autoUpdate`	boolean	`true`	Automatically update umpl
`umple.generateLanguage`	string	`"Java"`	Target language for code ge
`umple.diagramLayout`	string	`"dot"`	Layout engine for diagram

Keyboard Shortcuts

Shortcut	Command
`Cmd+Shift+U` / `Ctrl+Shift+U`	Show Diagram

Development

To test local changes to the LSP server:

1. Clone both repos side by side:

```
workspace/
├── umple-lsp/       # LSP server monorepo
└── umple.vscode/    # This extension
```

2. Build the server:

```
cd umple-lsp
npm install
npm run compile
```

3. Link the local server into the extension:

```
cd umple.vscode
npm install
npm link ../umple-lsp/packages/server
npm run compile
```

4. Press **F5** in VS Code to launch the Extension Development Host.

5. After making changes to the server, recompile and reload:

```
cd umple-lsp
npm run compile
```

Then in the dev host: `Cmd+Shift+P` (or `Ctrl+Shift+P`) → **Developer: Reload Window**

Where to make language-feature changes

This extension is mostly VS Code glue. Features such as diagnostics, completion, hover, inlay hints, go-to-definition, references, rename, formatting, workspace symbols, code actions, and LSP semantic tokens live in `umple-lsp/packages/server`.

Inlay hints are handled by VS Code's standard LSP client. No extension code is needed when the server advertises `textDocument/inlayHint`; users only need VS Code's `editor.inlayHints.enabled` setting to be on.

Formatting is provided by the LSP server. It is conservative: it formats parse-clean Umple structure, parser-visible structural commas, and already split multi-line list indentation. It does not format embedded target-language method or action bodies. Formatter behavior should be changed in `umple-lsp/packages/server`, not in this VS Code wrapper.

For highlighting:

- VS Code uses this repo's TextMate grammar as the baseline coloring path.
- The LSP server also advertises semantic tokens. Those are derived from `umple-lsp/packages/tree-sitter-umple/queries/highlights.scm` and mapped in `umple-lsp/packages/server/src/semanticTokens.ts`.
- If a token is parsed correctly but colored incorrectly across editors, update `highlights.scm` and/or `semanticTokens.ts` in `umple-lsp`, then rebuild the server and reload the Extension Development Host.
- Change this repo only when the VS Code wrapper needs different activation, commands, snippets, TextMate fallback scopes, settings, packaging, or server-launch wiring. Current TextMate fallback scopes intentionally cover common LSP semantic-token gaps such as tracer directives, port declarations/connectors, test sequences, and UmpleOnline layout directives.

Useful checks:

view plain

```
cd umple-lsp
npm run compile
npm test -w packages/server

cd ../umple.vscode
npm run compile
```

In VS Code, run **Developer: Inspect Editor Tokens and Scopes** to

distinguish TextMate scopes from LSP semantic token classifications.

Architecture

This extension is a thin client that launches the Umple LSP server. The server handles all language intelligence (diagnostics, completion, go-to-definition, references, rename, hover, inlay hints, formatting, workspace symbols, code actions, and semantic tokens). The extension adds VS Code-specific features: compile command, diagram panel, snippets, and UI chrome.

```
view plain

VS Code Extension (this repo)
  |
  +-- (stdio) --> umple-lsp-server --> umplesync.jar (diagnostics)
  |               |
  |               +-- tree-sitter (go-to-def, completion, references,
  |                                rename, hover, formatting, symbols,
  |                                semantic tokens, inlay hints)
  |
  +-- Compile command --> umplesync.jar -generate <lang>
  |
  +-- Diagram panel --> umplesync.jar -generate GvClassDiagram/...
  |                     |
  |                     +-- @viz-js/viz (DOT → SVG rendering)
  |
  +-- Click-to-select --> umple/resolveStateLocation (custom LSP request)
                          umple/resolveTransitionLocation
```

Original markdown source for this page: https://github.com/umple/umple.vscode/blob/master/README.md

Zed

Umple for Zed

Umple language support for the Zed editor, providing syntax highlighting, diagnostics, code completion, inlay hints, and go-to-definition for `.ump` files.

Installation

The extension is available on the Zed marketplace. Install it from Zed:

1. Open the command palette (`Cmd+Shift+P`) and run **zed: extensions**
2. Search for **Umple** and click Install

The extension automatically downloads the LSP server and Umple compiler — no manual setup required.

Dev install (from source)

To work on the extension itself, install it as a dev extension:

1. Clone this repo:

```
view plain
01. git clone https://github.com/umple/umple.zed.git
```

2. In Zed, open the command palette (`Cmd+Shift+P`) and run **zed: install dev extension**
3. Select the `umple.zed` directory

Prerequisites

- **Rust** (via rustup, not Homebrew rust — needed to compile the dev extension)
- **Node.js** 20+ (for running the LSP server; tested on 20 and 23)
- **Java** 11+ (optional — needed for diagnostics from the Umple compiler)

Features

- **Syntax highlighting** via tree-sitter grammar and highlight queries
- **Diagnostics** from UmpleSync compiler
- **Code completion** with context-aware keyword suggestions
- **Go-to-definition** for classes, interfaces, traits, enums, attributes, methods, state machines, states, associations, mixsets,

requirements, and `use` statements
- **Find references** across all reachable files with state-path disambiguation
- **Rename** safe symbol rename across all references
- **Hover** contextual information for symbols
- **Inlay hints** editor-only inferred type annotations for untyped attributes when Zed renders LSP inlay hints
- **Formatting** AST-driven indent correction, arrow spacing, declaration assignment spacing, structural comma spacing, blank-line normalization
- **Outline view** showing classes, methods, state machines, and more
- **Cross-file support** transitive `use` statement resolution and cross-file diagnostics
- **Auto-indentation** for blocks

Formatting is provided by the LSP server. It formats parse-clean Umple structure, parser-visible structural commas, and already split multi-line list indentation, but intentionally does not format embedded target-language method or action bodies. Formatter behavior belongs in `umple-lsp/packages/server`; this extension should only handle Zed launch/packaging behavior.

How It Works

The extension automatically installs `umple-lsp-server` from npm and downloads `umplesync.jar` for compiler diagnostics. The server is launched via Node.js in `--stdio` mode.

Configuration (optional)

You can adjust certain settings using Settings … / Open Settings (cmd ,) or by editing the `settings.json` file.

Configuration settings for users

By default lines of Umple code that have errors or warnings are underlined; you can see the error or warning at the bottom of the screen if you click on the underlined text.

However, if you would like such messages from the Umple compiler to appear inline (on the line where each problem occurs), then you can change the Languages & Tools / Diagnostics / Enabled setting to be true. You can also do this by adding the following to the settings.json file.

```
view plain

{
  "diagnostics": {
    "inline": {
      "enabled": true
    }
  }
}
```

Configuration settings for developers working on the Umple LSP server

For development, you can override the auto-downloaded server with a local build. Add to your Zed `settings.json` (`Cmd+,`):

```
view plain

{
  "lsp": {
    "umple-lsp": {
      "settings": {
        "serverPath": "/path/to/umple-lsp"
      }
    }
  }
}
```

This points to a locally cloned and built umple-lsp repository.

Updating

Marketplace install: Zed updates installed extensions automatically. New versions reach you via the Zed Extensions marketplace.

Dev install (from source): pull the latest changes and Zed will pick them up:

```
view plain

cd umple.zed
git pull
```

Then restart Zed or reload the extension.

The LSP server itself is downloaded from npm at every extension load (via

`npm_package_latest_version("umple-lsp-server")`), so server-only changes reach you on the next Zed startup without any extension update needed.

Grammar Sync

The tree-sitter grammar and query files are derived from umple-lsp. The following files are auto-synced and should not be edited manually:

- `extension.toml [grammars.umple].rev` — pinned commit for `parser.c`
- `languages/umple/highlights.scm` — copied from `umple-lsp/packages/tree-sitter-umple/queries/`

How sync usually happens

A GitHub Action in `umple/umple-lsp` auto-opens (or updates) a PR on this repo whenever grammar or query files change upstream. The PR force-pushes onto a stable branch `sync/umple-lsp-master`, bumps `extension.toml` patch version, and includes a body that classifies the change as `grammar | highlights | both` plus a "safe to merge?" hint. You just merge it.

Manually syncing (rarely needed)

If the auto-PR workflow is broken or you want to sync ahead of an upstream push:

```
view plain
01. ./scripts/sync-grammar.sh --source /path/to/umple-lsp
```

Checking for drift (CI or local)

```
view plain
01. ./scripts/sync-grammar.sh --source /path/to/umple-lsp --check
```

This exits with code 1 if any synced file is out of date. The `.github/workflows/check-sync.yml` workflow in this repo runs this on every push/PR to master to catch drift.

Where to Make Language-Feature Changes

This extension is mainly the Zed wrapper. Core language behavior belongs in `umple-lsp`:

- Diagnostics, completion, hover, inlay hints, go-to-definition, references, rename, formatting, workspace symbols, code actions, and LSP semantic tokens: `umple-lsp/packages/server`
- Parser and tree-sitter highlighting: `umple-lsp/packages/tree-sitter-umple`
- Zed extension loading, npm server download, language registration, and grammar pinning: this repo

Zed's visible syntax highlighting comes from the synced tree-sitter highlight query at `languages/umple/highlights.scm`. The LSP server also advertises semantic tokens for clients that use them; those are generated from the upstream `highlights.scm` and mapped in `umple-lsp/packages/server/src/semanticTokens.ts`.

When highlighting changes upstream, prefer the sync workflow or run:

```
view plain

./scripts/sync-grammar.sh --source /path/to/umple-lsp --check
./scripts/sync-grammar.sh --source /path/to/umple-lsp
```

Change this repo directly only when the Zed wrapper behavior changes or when accepting the generated grammar/highlight sync.

Troubleshooting

Extension fails to compile ("failed to compile Rust extension")

Zed compiles extensions to WebAssembly (`wasm32-wasip2`), which requires the Rust toolchain from rustup. Homebrew's `rust` package only includes the native target and can't cross-compile to WASM.

```
view plain

# Remove Homebrew rust if installed
brew uninstall rust

brew install rustup
# Or install rustup directly via [rustup](https://rustup.rs/)
```

LSP server not starting

Check **View > Toggle Language Server Logs** in Zed for errors. Common issues:

- **Node.js not found:** Install Node.js 18+ and ensure it's on your PATH
- **npm install failed:** Check internet connection, restart Zed to retry

No diagnostics

Diagnostics require Java 11+. Check the LSP logs (**View > Toggle Language Server Logs**) for errors related to `umplesync`.

License

MIT

Original markdown source for this page: https://github.com/umple/umple.zed/blob/master/README.md

BBEdit

Umple for BBEdit

Umple language support for BBEdit 14+, providing syntax highlighting, diagnostics, code completion, go-to-definition, find references, rename, formatting, and document symbols for `.ump` files.

BBEdit has built-in Language Server Protocol support — no plugins required.

How It Works

Two systems work together:

Responsibility	Pı
Syntax coloring (keywords, comments, strings)	Cc
Comment/uncomment commands	Cc
Diagnostics, completion, go-to-def, references, rename, formatting, symbols	LS

BBEdit explicitly does NOT use LSP for syntax highlighting — you need both the plist and the server.

Prerequisites

- **BBEdit 14+** (built-in LSP support)
- **Node.js 18+** (for running the LSP server)
- **Java 11+** (optional — only needed for diagnostics from the Umple compiler)

Installation

1. Install the LSP server

view plain

```
01. npm install -g umple-lsp-server
```

Download `umplesync.jar` (needed for diagnostics):

view plain

```
01. curl -fSL -o "$(npm root -g)/umple-lsp-server/umplesync.jar" \
02.   https://try.umple.org/scripts/umplesync.jar
```

2. Install the Codeless Language Module

The `Umple.plist` file in this directory is a complete Codeless Language Module that provides both syntax coloring and LSP server auto-discovery. Copy it to BBEdit's Language Modules folder:

```
view plain

mkdir -p ~/Library/Application\ Support/BBEdit/Language\ Modules
cp Umple.plist ~/Library/Application\ Support/BBEdit/Language\ Modules/
```

Or if you cloned this repo:

```
view plain

mkdir -p ~/Library/Application\ Support/BBEdit/Language\ Modules
cp editors/bbedit/Umple.plist ~/Library/Application\ Support/BBEdit/Language\ Modules/
```

You may need to set the initialization options as specified in Section 5.

3. Restart BBEdit

Quit BBEdit completely and relaunch. Language modules are only loaded at startup.

4. Verify

1. Open any `.ump` file
2. Check the language selector at the bottom of the editor — it should show **Umple**
3. Check **BBEdit > Settings > Languages > Umple > Server** — the status dot should be **green** (server found)
4. If the dot is red, see Troubleshooting below
5. Create an Umple file (.ump suffix) and type `statemachine sm {}` and save the file. Verify that the keyword statemachine is coloured. If not see Troubleshooting
6. In the same Umple file, type an error (such as a single line with junk) and save the file. Verify that a pink colour appears on the line number and also in a little triangle at the top; click either of these to see the diagnostic information; then click Show All to see the diagnostics panel. If this does not work follow the directions in the next section to set initialization options.

5. Diagnostics setup

The server automatically finds `umplesync.jar` next to itself (since v0.2.6). As long as you downloaded the jar into the npm package directory in step 1, diagnostics work with no extra configuration.

If diagnostics still don't work, you can explicitly point BBEdit to the jar. Find the jar path:

```
view plain

01. echo "$(npm root -g)/umple-lsp-server/umplesync.jar"
```

Create a configuration file:

```
view plain

01. mkdir -p ~/Library/Application\ Support/BBEdit/
    Language\ Servers/Configuration
```

Save the following as `~/Library/Application Support/BBEdit/Language Servers/Configuration/Umple.json` (adjust the path if your jar is in a different location):

```
view plain

01. {
02.   "initializationOptions": {
03.     "umpleSyncJarPath": "/usr/local/lib/node_modules/umple-lsp-
    server/umplesync.jar"
04.   }
05. }
```

Then tell BBEdit to use it: **Settings > Languages > Custom Settings** > double-click **Umple** > **Server** tab > set **Configuration** to **Umple**.

BBEdit supports C-style comments (`//`, `/* */`) in these JSON config files.

Features

From the Codeless Language Module (plist)

- **Syntax highlighting** — Keywords, comments (`//`, `/* */`), and strings (`"..."`) are colored

- **Comment/uncomment** — `Cmd+/` toggles line comments; `Cmd+Shift+/` toggles block comments
- **Block folding** — `{ / }` blocks can be folded

From the LSP Server

Feature	How to Use
Diagnostics	Errors/warnings appear as colored dots in the line numb
Code completion	Type and BBEdit shows completion suggestions from th
Go to Definition	`Cmd`-double-click a symbol, or right-click > **Go to Definit**
Go to Declaration	Right-click > **Go to Declaration**
Find References	**Search > Find References to Selected Symbol**
Rename	Right-click > **Rename Selected Symbol**, or **Edit > Renam**
Document Formatting	**Edit > Reformat Document** (or **Reformat Selection** for r
Document Symbols	**Go > Go to Named Symbol** for quick navigation
Workspace Symbols	**Go > Find Symbol in Workspace** for project-wide search
Code Actions	Right-click for context-menu code actions
Signature Help	**Edit > Show Parameter Help**

Not supported by BBEdit

- **Hover** — BBEdit does not support `textDocument/hover`
- **Semantic tokens** — Syntax coloring comes from the plist, not the LSP

How the Plist Works

The `Umple.plist` file does two things:

1. Language definition — Maps `.ump` files to the "Umple" language, defines comment/string delimiters, and provides a keyword list for syntax coloring.

2. LSP auto-discovery — The `BBLMLanguageServerInfo` dictionary tells BBEdit to automatically launch `umple-lsp-server --stdio` when `.ump` files are opened:

```
view plain

<key>BBLMLanguageServerInfo</key>
<dict>
    <key>ServerCommand</key>
    <string>umple-lsp-server</string>
    <key>ServerArguments</key>
```

```
    <array>
       <string>--stdio</string>
    </array>
    <key>ServerLanguageID</key>
    <string>umple</string>
</dict>
```

This means no manual configuration in Settings > Languages > Server is needed — it's all automatic from the plist.

Troubleshooting

Server status dot is red

The red dot in **Settings > Languages > Umple > Server** means BBEdit found the configuration but can't locate the executable.

Common cause: BBEdit reads $PATH from ~/.zshenv, NOT ~/.zshrc. If you installed Node.js via nvm, fnm, or Homebrew and only added it to PATH in .zshrc, BBEdit won't find it.

Fix options:

1. Add the PATH export to ~/.zshenv:

```
view plain

# ~/.zshenv
export PATH="/opt/homebrew/bin:$PATH"
# or for nvm:
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh"
```

2. Or use the full absolute path: open **Settings > Languages > Umple > Server** and set **Command** to the full path:

```
view plain

which umple-lsp-server
# e.g., /opt/homebrew/bin/umple-lsp-server
```

3. Or symlink into BBEdit's Language Servers directory:

```
view plain
```

```
mkdir -p ~/Library/Application\ Support/BBEdit/Language\ Servers
ln -s "$(which umple-lsp-server)" ~/Library/Application\ Support/BBEdit/Language\ Servers/
```

Server status dot is gray

Gray means the server is not configured. Make sure `Umple.plist` is in `~/Library/Application Support/BBEdit/Language Modules/` and you've restarted BBEdit.

No diagnostics

1. Ensure `umplesync.jar` exists:

```
view plain
ls "$(npm root -g)/umple-lsp-server/umplesync.jar"
```

2. Verify Java is installed: `java –version`
3. If the jar is in a non-standard location, create `Umple.json` (see step 5 above)

LSP features not working

Check the debug log:

```
view plain
~/Library/Logs/BBEdit/LanguageServerProtocol-Umple.txt
```

This log shows all JSON-RPC messages between BBEdit and the server.

Server hangs on quit

If BBEdit hangs when closing, the server may not be responding to the `exit` notification. Fix:

```
view plain
defaults write com.barebones.bbedit ForceQuitLSPServerAfterExit_lsp-server -int 2
```

This tells BBEdit to force-kill the server after 2 seconds if it doesn't exit cleanly.

Per-Project Configuration

For project-specific settings, place a `.BBEditLSPWorkspaceConfig.json` file in the project root:

```
view plain

{
  // Project-specific LSP settings
}
```

BBEdit passes this to the server as workspace configuration.

Updating

```
view plain

npm update -g umple-lsp-server
```

Then restart BBEdit. The plist does not need updating unless new keywords are added to the Umple language.

Customizing Keywords

The keyword list in `Umple.plist` controls which identifiers get syntax coloring. To add or remove keywords, edit the `BBLMKeywordList` array in the plist and restart BBEdit. The full list of keywords is organized by category:

- **Entity declarations:** `class`, `interface`, `trait`, `enum`, `association`, `associationClass`, `namespace`, `statemachine`
- **Class body:** `isA`, `singleton`, `immutable`, `abstract`, `lazy`, `const`, `autounique`, `defaulted`, `depend`, `use`, `key`
- **Visibility:** `public`, `private`, `protected`, `internal`
- **Types:** `Integer`, `String`, `Boolean`, `Double`, `Float`, `Date`, `Time`, `void`
- **State machine:** `queued`, `pooled`, `entry`, `exit`, `do`, `final`, `unspecified`, `as`, `afterEvery`
- **Code injection:** `before`, `after`, `around`
- **Directives:** `generate`, `mixset`, `require`, `trace`, `tracecase`

Original markdown source for this page: https://github.com/umple/umple-lsp/blob/master/editors/bbedit/README.md

Neovim

umple.nvim

Neovim plugin for the Umple modeling language. Provides diagnostics, code completion, go-to-definition, find references, rename, hover, formatting, and syntax highlighting for `.ump` files.

Requirements

- Neovim 0.9+ (0.10+ recommended)
- Node.js 20+ (for running the LSP server; tested on 20 and 23)
- A C compiler (`cc` or `gcc`) for compiling the tree-sitter parser
- Java 11+ (optional — only needed for diagnostics)
- A plugin manager — this guide uses lazy.nvim

Installing prerequisites

macOS (via Homebrew):

```
view plain

brew install neovim node
brew install openjdk  # optional, for diagnostics
```

Ubuntu/Debian:

```
view plain

sudo apt install neovim nodejs npm build-essential
sudo apt install default-jdk  # optional, for diagnostics
```

Setting up lazy.nvim

If you don't have a plugin manager yet, follow the lazy.nvim installation guide.

Installation

Add this to your lazy.nvim plugin list:

```
view plain

{
  'umple/umple.nvim',
  build = './scripts/build.sh',
  dependencies = {
    'neovim/nvim-lspconfig',
```

```
    'nvim-treesitter/nvim-treesitter',
  },
  config = function()
    require("umple-lsp").setup()
  end,
}
```

The build script downloads the LSP server from npm and fetches the tree-sitter grammar. The tree-sitter parser is compiled automatically on first load — no manual steps needed.

Auto-completion (recommended)

For auto-popup completion, install a completion plugin such as nvim-cmp with cmp-nvim-lsp. Without one, the LSP server still provides completions but you won't see them automatically.

Configuration

view plain

```
require("umple-lsp").setup({
  on_attach = function(client, bufnr)
    -- your custom keybindings
  end,
})
```

> Earlier versions of this plugin documented a `port = 5556` option (passed to the server as `umpleSyncPort`). The current LSP server doesn't read it — diagnostics now spawn `umplesync.jar` as a subprocess per request, no socket. The option is harmlessly ignored if you set it.

Keybindings

No keybindings are set by default — use `on_attach` to configure your own. Example:

view plain

```
require("umple-lsp").setup({
  on_attach = function(client, bufnr)
    local opts = { buffer = bufnr }
```

```
    vim.keymap.set("n", "gd", vim.lsp.buf.definition, opts)
    vim.keymap.set("n", "gr", vim.lsp.buf.references, opts)
    vim.keymap.set("n", "K", vim.lsp.buf.hover, opts)
    vim.keymap.set("n", "<leader>rn", vim.lsp.buf.rename, opts)
    vim.keymap.set("n", "<leader>f", vim.lsp.buf.format, opts)
    vim.keymap.set("n", "<leader>e", vim.diagnostic.open_float, 
    vim.keymap.set("n", "[d", vim.diagnostic.goto_prev, opts)
    vim.keymap.set("n", "]d", vim.diagnostic.goto_next, opts)
  end,
})
```

Features

- **Diagnostics**: Real-time error and warning detection (requires Java)
- **Go-to-definition**: Jump to classes, interfaces, traits, enums, attributes, methods, state machines, states, associations, mixsets, requirements, and use statements
- **Find references**: Semantic reference search across all reachable files
- **Rename**: Safe symbol rename across all references
- **Hover**: Contextual information for symbols
- **Code completion**: Context-aware keyword and symbol completion
- **Document symbols**: Hierarchical outline (accessible via `:Telescope lsp_document_symbols` or similar)
- **Formatting**: AST-driven indent correction, arrow spacing, blank-line normalization
- **Syntax highlighting**: Via tree-sitter grammar (compiled automatically)
- **Cross-file support**: Transitive use statement resolution and cross-file diagnostics

Updating

```
view plain
:Lazy update umple.nvim
```

This pulls the latest plugin code and re-runs the build script (which updates the LSP server and tree-sitter grammar).

Troubleshooting

Parser, highlighting, and diagnostics

Neovim uses two separate Umple checks:

- Tree-sitter parser + queries: syntax highlighting, `:InspectTree`, and local parse shape. The cached native parser is `~/.local/share/nvim/site/parser/umple.so`; query files are linked at `~/.local/share/nvim/site/queries/umple`.
- LSP diagnostics: compiler errors and warnings from `umplesync.jar`. These are the pink/red diagnostics such as E1502.

A file can highlight and parse correctly in tree-sitter while still showing an `umplesync.jar` compiler diagnostic. In that case the LSP is working; the compiler is rejecting the Umple program.

No syntax highlighting

The plugin compiles the tree-sitter parser on first load. If it fails:

1. Make sure you have a C compiler: `cc --version` or `gcc --version`
2. Check that the query symlink points to this plugin:

```
view plain
01. readlink ~/.local/share/nvim/site/queries/umple
```

3. Delete the cached parser and restart Neovim to recompile:

```
view plain
01. rm -f ~/.local/share/nvim/site/parser/umple.so
02. rm -rf ~/.local/share/nvim/site/queries/umple
```

No diagnostics

Diagnostics require Java 11+. Verify: `java -version`

LSP not starting

1. Check that Node.js is installed: `node --version`
2. Run `:LspLog` in Neovim to see server errors
3. Rebuild the plugin: `:Lazy build umple.nvim`

Development

To test local changes to the LSP server:

1. Clone both repos side by side:

```
view plain

workspace/
├── umple-lsp/       # LSP server monorepo
└── umple.nvim/      # This plugin
```

2. Build the server locally:

```
view plain

cd umple-lsp
npm install
npm run compile
npm run download-jar
```

3. Override the plugin's paths to use your local build:

```
view plain

{
  dir = '/path/to/umple.nvim',
  dependencies = {
    'neovim/nvim-lspconfig',
    'nvim-treesitter/nvim-treesitter',
  },
  config = function()
    require("umple-lsp").setup({
      plugin_dir = '/path/to/umple.nvim',
    })
  end,
}
```

Then symlink the local server into the plugin's expected locations:

```
view plain

cd umple.nvim
# Symlink tree-sitter grammar
ln -sf ../umple-lsp/packages/tree-sitter-umple tree-sitter-
umple
# Symlink local server into node_modules (jar is already inside pa
server/)
mkdir -p node_modules
```

```
ln -sf ../../umple-lsp/packages/server node_modules/umple-lsp-server
```

After making changes to the server, recompile and restart:

```
view plain

cd umple-lsp
npm run compile       # Server-only changes
npm run build-grammar # After grammar.js changes (regenerates parser + WAS
```

Then in Neovim: `:LspRestart`

After grammar changes, also delete the cached native parser so Neovim recompiles it. The plugin repairs stale query symlinks on startup, but deleting the query path is still a useful reset when debugging local setup:

```
view plain

rm -f ~/.local/share/nvim/site/parser/umple.so
rm -rf ~/.local/share/nvim/site/queries/umple
```

How it works

The build script installs the pre-compiled umple-lsp-server from npm, downloads `umplesync.jar` for diagnostics, and extracts the tree-sitter grammar (`src/parser.c` + `queries/`) from the umple-lsp repo. The plugin compiles the tree-sitter parser automatically on first load (into `~/.local/share/nvim/site/parser/umple.so`), links query files into `~/.local/share/nvim/site/queries/umple`, and points Neovim's LSP client at the npm-installed server.

Original markdown source for this page: https://github.com/umple/umple.nvim/blob/master/README.md

Sublime

Umple LSP for Sublime Text

This guide explains how to set up the Umple Language Server with Sublime Text.

Prerequisites

- Sublime Text 4 (or 3 with LSP package)
- LSP package installed
- Node.js 18+
- Java 11+ (for umplesync.jar)

Installation

1. Build the LSP server

From the umple-lsp directory:

```
view plain

npm install
npm run compile
npm run download-jar
```

2. Install LSP package

Open Sublime Text and install the LSP package via Package Control: 1. `Cmd+Shift+P` (or `Ctrl+Shift+P` on Linux/Windows) 2. Type "Package Control: Install Package" 3. Search for "LSP" and install it

3. Add Umple syntax definition

Copy the syntax file to your Sublime `Packages/User` directory:

```
view plain

cp editors/sublime/Umple.sublime-syntax \
  ~/Library/Application\ Support/Sublime\ Text/Packages/User/
```

Or create `Packages/User/Umple.sublime-syntax` manually with:

```
view plain

%YAML 1.2
---
```

```
name: Umple
file_extensions: [ump]
scope: source.umple

contexts:
  main:
    - match: '\b(class|interface|trait|enum|association|namespace|use)\b'
      scope: keyword.control.umple
    - match: '\b(isA|singleton|immutable|abstract|lazy|const)\b'
      scope: keyword.other.umple
    - match: '\b(Integer|String|Boolean|Double|Float|Date|Time)\b'
      scope: storage.type.umple
    - match: '//.*$'
      scope: comment.line.umple
    - match: '/\*'
      push: block_comment
    - match: '"[^"]*"'
      scope: string.quoted.double.umple

  block_comment:
    - meta_scope: comment.block.umple
    - match: '\*/'
      pop: true
```

4. Configure LSP

Open LSP settings: `Preferences > Package Settings > LSP > Settings`

Add the Umple client configuration:

view plain

```
{
  "clients": {
    "umple": {
      "enabled": true,
      "command": ["node", "/path/to/umple-lsp/packages/server/out/server.js", "--stdio"],
      "selector": "source.umple",
      "initializationOptions": {
        "umpleSyncJarPath": "/path/to/umple-lsp/packages/server/umplesync.jar",
```

```
      "umpleSyncPort": 5558
    }
   }
  }
}
```

Important: Update `/path/to/umple-lsp` to your actual installation path.

5. Restart Sublime Text

Close and reopen Sublime Text for all changes to take effect.

Vim keybindings (Vintage mode)

If you use Vintage mode, add this to your keybindings (`Preferences > Key Bindings`) for `gd` go-to-definition:

view plain

```
[
  {
    "keys": ["g", "d"],
    "command": "lsp_symbol_definition",
    "context": [
      { "key": "setting.command_mode", "operand": true }
    ]
  }
]
```

Features

- **Syntax highlighting**: Basic regex-based highlighting for keywords, types, comments, and strings
- **Diagnostics**: Real-time error and warning detection
- **Go-to-definition**: Jump to class, attribute, state definitions
- **Code completion**: Context-aware keyword and symbol completion

Troubleshooting

LSP not starting

1. Verify Java is installed: `java -version`
2. Test the server manually:

```
view plain

01. node "/path/to/umple-lsp/packages/server/out/server.js" --stdio
```

3. Open the Sublime console (`View > Show Console`) for errors
4. Check the LSP log panel: `Cmd+Shift+P` > "LSP: Toggle Log Panel"

No syntax highlighting

1. Ensure the `.sublime-syntax` file is in the `Packages/User` directory
2. Check the status bar shows "Umple" when a `.ump` file is open
3. Restart Sublime Text

Go-to-definition not working

1. Check if the LSP is running (status bar should show "umple" as a running server)
2. Make sure the file is recognized as Umple (check status bar)
3. Open the LSP log panel to see if the server is responding

Updating

After pulling updates:

```
view plain

01. cd /path/to/umple-lsp
02. npm run compile
```

Then restart Sublime Text.

Original markdown source for this page: https://github.com/umple/umple-lsp/blob/master/editors/sublime/README.md

UmpleOnline AI Capabilities

This page describes the AI capabilities built in to UmpleOnline. This works at a basic level and is subject to extensive improvement in late 2026 and 2027. The capabilities described below focus on classic UmpleOnline, but the same capability is present in UmpleOnline V2.

The AI capability can use various large language models (LLMs) to generate class diagrams and state machines from requirements specified in Umple. Before using this feature you will need to create a set of requirements, or load an example set of requirements, such as one of the examples in this manual.

- **AI setup.** UmpleOnline uses a **BYOK (bring-your-own-key)** approach for AI features. Your API key is stored locally in your browser's local storage and is never sent to our servers. All AI requests are made directly from your browser to the selected provider.
  - **AI settings dialog.** In UmpleOnline, open the AI settings from the AI menu which appears at the bottom of the central menu. When you open this for the first time, click where it says 'Not Configured'
  - **Select a provider.** Currently, the supported providers are OpenAI, Google Gemini, and OpenRouter.
  - **Enter API key.**
    - If you already have an API key, enter it in the field.
    - If you don't have one, click the "Get API Key" link next to the provider selection to obtain a key from your provider.
  - **Verify API key.** Click the Verify button. The system checks the key's validity with the provider. If verification succeeds, a list of available models is displayed.
  - **Select a model.** Choose a model from the dropdown list. Free models are listed at the top, followed by paid models grouped by provider. The cost per million tokens (input and output) is displayed below the selection dropdown.
- **Status indicator.** The AI status indicator in the sidebar shows the current configuration status. Once a model is selected, the indicator displays the model name and provider name. AI preferences

(provider, model, verification, usage) are stored in your browser's local storage.

- **Prompt caching and token usage.** Prompt caching reuses previously computed prompt prefixes. Cache hits require exact prefix matches, so stable/repeated content should be placed first and request-specific content last. UmpleOnline sends AI requests as chat messages and appends new turns, which helps preserve a reusable prefix across related requests. When cache hits occur, billed input tokens and latency can drop, but cache behavior is provider/model dependent and not guaranteed on every request. For details and examples (including OpenAI's automatic caching behavior, token thresholds, and the `cached_tokens` usage metric), see OpenAI Prompt Caching.

- **Explain code.** Open the AI Explain tool. The dialog analyzes the entire editor content and streams the explanation. You can use the follow-up input to ask questions about the same model, as the dialog maintains a short conversation history for context. If the editor is empty, an alert dialog will prompt you to enter Umple code first.

- **Fix guidance.** After compiling your model, when errors appear, each issue is shown in the bottom pane. Click the AI fix icon beside an issue to generate guidance. The tool highlights the relevant line in the editor and returns a short fix suggestion. Note that this feature provides guidance only and does not auto-edit your code.

- **Generate from requirements.** Add requirement blocks to your model with the syntax `req ID { ... }`. Learn more about requirements. Open the AI Requirements dialog. If multiple requirements exist, choose to use all or select specific ones. Select a generation type (state machine or class diagram) and click Generate. The result is streamed into the dialog, where you can stop generation, review the output, and insert it into the editor. When enabled, the system runs a compiler-based self-correction loop to repair issues in the generated block before insertion.

- **Troubleshooting.**
  - If AI actions are disabled, ensure you have selected a provider, entered an API key, verified it successfully, and selected a model.
  - If you see messages such as *Rate limited* or *Provider error*, or

if you are unsure about keys and OpenRouter integrations, see Help with AI keys.

- If no requirements are found, confirm your model contains `req ... { ... }` blocks.
- If a generated block fails to compile, review the error list and re-run generation with fewer or clearer requirements.

Umple AI Skills

This page describes AI agent skills for working with Umple and the Umple CLI (Command line interface). This is in beta and will be enhanced.

A 'skill' is a technology released by Vercel that is gaining acceptance in 2026. The npm skills package is here.

Repository overview

The umple-skills repository (https://github.com/umple/umple-skills or initially here) provides reusable AI agent skills, scripts, and guidance for working with Umple models and diagrams. This page gives a high-level overview and links to separate sections for each skill.

- **Install the skills pack.** Use `npx skills add umple/umple-skills` to add the skills to your agent environment.
- **Skill definition (SKILL.md).**
  - **name** declares the skill identifier used for routing and invocation.
  - **description** summarizes when the skill should be selected.
  - **allowed-tools** lists the executable commands the agent may run.
  - **Prompt body** (the markdown content) gives step-by-step guidance and output expectations.
- **Scripts and references.**
  - **scripts/** holds runnable helpers (for example, `scripts/main.ts`) invoked by the skill.
  - **references/** contains focused guidance (for example, diagram rules) that the prompt cites.

Skills follow a lazy-loading strategy: only the selected skill folder is loaded at runtime. Semantic routing uses the `name` and `description` fields to pick the best skill for a user request.

umple-diagram-generator

- Generates class diagrams and state machines from natural-language requirements by producing Umple models and rendering them to SVG. It uses guidance from the `references/` folder and relies on the Umple CLI, Graphviz, and Bun.
- **Output location.** The skill invokes the Umple CLI through `scripts/main.ts`.
  - **Scenario A: output path not specified.** The script creates a folder named after the diagram plus a timestamp, and stores

all generated files inside it.

```
<output-dir>/
└── <diagram-name>_<timestamp>/
    ├── model.ump
    ├── model.gv
    └── model.svg
```

- **Scenario B: output path specified.** If you provide a full `.svg` path, the script writes only the SVG file to that exact location.

```
<output-dir>/
└── <diagram>.svg
```

Help with AI keys

This page helps you understand how AI keys work in UmpleOnline, and what to do when you see errors such as *Provider error* or *Rate limited*.

Important: UmpleOnline uses a **BYOK (bring-your-own-key)** approach. Your AI key is stored in your browser's local storage and is never sent to Umple servers. Your browser sends requests directly to the AI provider you selected.

1. What does "Verify key" do?

- **Verify key** checks that the key you entered is accepted by the selected provider.
- If verification succeeds, UmpleOnline enables the model list and lets you choose a model.
- Verification status is stored in your browser's **local storage**. In normal browsing mode it should persist across page reloads and future sessions on the same browser/computer.

2. Common errors and what they mean

- **Authentication failed. Please check your API key.**

  The provider rejected your key. Common causes:

  - The key was pasted incorrectly (extra spaces, missing characters).
  - The key belongs to a different provider than the one selected.
  - The key was revoked or expired in the provider dashboard.
- **Rate limited (...). Please retry shortly.**

  This means the provider (or an upstream provider behind OpenRouter) temporarily rejected requests because too many requests are happening in a short time. What to do:

  - Wait and retry.
  - Switch to a different model (often the fastest fix).
  - OpenRouter may prompt you to set up your own **integration key**. This means the upstream provider behind OpenRouter is rate-limiting requests. Users are being asked for a **BYOK** setup because it is assumed that no providers are currently available for the selected model. Consequently, users must upload an API key from their own chosen provider to continue using it.

3. Tips for reliable use

- **Providers prefer paid users:** Free tier users might reach their threshold easily compared to paid users.
- **Switch models when blocked:** With OpenRouter, multiple upstream providers may exist behind a model, and availability can change minute-to-minute.
- **If verification seems to "reset":** check whether your browser is clearing local storage (private mode, strict tracking protection, storage cleaners, etc.).

Java

The Java generator is the most full-featured of all the Umple code generators.

Java code can be embedded in Umple, and Umple can generate Java from all its modeling features.

Java code can be executed in UmpleOnline if one or more of the classes has a main method. UmpleOnline guards against compile-time and run-time Java errors when executing code. It provides feedback about the relevant line of code where any error occurs.

The Umple compiler itself is written in Umple with embedded Java, and the Java generator is then used to generate the compiler. All Umple features are supported by this generator

PHP

The PHP generator supports all class diagramming features of Umple, and generates code that is fully tested.

PHP code can be embedded in Umple, and Umple can generate PHP features such as classes, attributes and associations. This can save a lot of boilerplate programming for systems with complex class structures. The generated PHP code can then be combined with other PHP code written separately.

State machines are not yet fully supported in PHP

Python

The Python generator in Umple generates Python 3 classes from Umple models, and lets you embed Python methods in Umple. It is chosen with `generate Python;` in a model, `-g Python` on the command line or Python Code in UmpleOnline. The generated classes follow the public API of the Java generator's classes, with the Python behaviour described below.

Changed in Umple 1.38: As of Umple 1.38, the Python generator described here replaces the earlier one, which translated the generated Java code into Python. It generates Python from the model directly, so the generated code differs from what earlier versions of Umple produced. If you relied on the earlier Python generator, earlier versions of Umple are available from the Umple releases on GitHub.

Embedding Python methods: To insert a Python method in Umple, you specify the method signature followed by the keyword Python, an open curly bracket, any number of lines of Python code, and a closing curly bracket. The code is used as written, indented to fit the generated method. A method can have a Java body and a Python body side by side; each generator uses its own. A body without a language tag is taken to be Python. A method that has bodies only for other languages is left out of the Python with warning W9212, and so are actions, activities and code injections written only for other languages. A derived attribute written only for other languages would have no value, so it is reported with error E9211.

Umple finds the closing curly bracket with Java's comment rules. A curly bracket in a Python # comment therefore counts, so keep curly brackets in comments balanced. The rest of a line after Python's // operator is skipped as a comment, so put the closing curly bracket on a line of its own after such a line.

To create a **main Python function**, use the same signature as for Java with the code tagged as Python: `public static void main(String [ ] args) Python {`. Running the generated module calls main with the full `sys.argv`. From the folder Umple generated into, run `python3 -m shop.Main` for a class Main in namespace shop, or `python3 Main.py` for a class without a namespace. Other modules that import the main class get the running one.

Python code can use the classes of its own namespace, and those listed with depend (`depend shop.Item;` or `depend shop.*;`), without importing them: Umple imports such a class at the start of each method that names

it, as Java sees these classes without an import, unless the method binds that name itself (as a parameter, a local variable or its own import). Other classes are imported as in any Python code, for example `from shop.Item import Item`, where `shop` is the class's namespace.

Untagged code in actions, injections and defaults: Actions of state machines, entry, exit and do activities, code injections, derived attributes and initial values written without a language tag are Java. For Python, Umple translates such code when it keeps to a small subset of Java: literals, local variables, assignments, conditions and loops, calls to the model's own methods, `new` for model classes, the String methods `equals`, `contains`, `startsWith`, `endsWith`, `length` and `isEmpty`, `Thread.sleep` (which in a do activity also ends the activity once it is cancelled, as Java's interrupt does), `System.currentTimeMillis`, `Math`'s `abs`, `max`, `min`, `sqrt`, `pow`, `floor`, `ceil`, `round` and `random`, `System.out.println` and `throw new RuntimeException`. Other code is reported with error E9211, which shows how that kind of code is written in Python. Actions, activities, injections and derived attributes can have a Python version instead, which is used as written; an initial value can call a method written in Python. In text, booleans, the `null` literal and a missing model object read `true`, `false` and `null`, as in Java.

The generated Python:

- Each class is a module `ClassName.py` in the package given by its namespace. Files are written in UTF-8.
- Attribute setters and the methods that set, add or remove links return `True`, or `False` when the change is refused (for example by a constraint, a multiplicity bound or a key that can no longer change). An add method that creates the linked object from constructor arguments returns it, or `None` when the association is full.
- Getters of association ends with many objects return tuples; getters of list attributes return copies.
- An `autounique` attribute, like a `unique` one, gets the class methods `getWith` and `hasWith` followed by its name, which find an object by its value.
- Links between objects are compared by identity. Classes with a key also get __eq__ and __hash__ based on the key.
- `str()` of an object has the layout of Java's toString, with Python's spelling of values (`True`, `None`) except that a missing date or linked object prints `null` as in Java. A toString method written in the model replaces it.

- Comments on classes, attributes, associations and methods, and the requirements they implement, become docstrings.
- Parameters named like Python keywords are renamed: `in` becomes `input`, and other keywords get a trailing underscore.
- Overloaded methods become one method that calls the most specific overload matching the arguments, judged by their types when the program runs (Java judges by the declared types of the expressions passed). `None` matches any reference type, boxed numbers such as `Integer` included; when several overloads take it, the most specific is called, then the first declared, where Java may report the call as ambiguous.
- A method written in the model with the same parameter types as a method Python generates (for example a Python `delete()`) replaces that method; other overloads of the name stay. `int` and `Integer` count as one type.
- A subclass's methods compete with the inherited methods of their name that they do not override: the most specific one is called, wherever it is declared. A subclass method reaches the parent's methods through `super()` (calling `self` would call its own method again). A static method a subclass declares hides the inherited methods of its name.
- Guards, preconditions and invariants can use Java's `equals`, `contains` and a String's `length()`, and the constants of other classes (`Limit.MAX`, `shop.Limit.MAX`, `Outer.Inner.MAX`, also through a subclass of `Outer`), which the method imports. A String's length counts UTF-16 code units, as in Java.
- A class constant whose value is made of literals and operators alone, such as `60 * 24`, is set in the class body. Any other is computed once, when it is first read, and then kept, so it can use any class of any module, whichever is loaded first; a branch of a conditional that is not taken is never run. A value whose computation needs the value itself raises `RuntimeError` naming it, where Java would read a default such as 0.
- A synchronized method holds a reentrant lock of its object while it runs, contracts and injections included, as Java's holds the object's monitor; the object's class and its subclasses share that lock. A static synchronized method holds its class's lock.
- Translated code computes with Python's numbers: integers do not overflow at 32 or 64 bits, `Float` has double precision, and a floating division by zero raises `ZeroDivisionError` where Java gives NaN or infinity. `Math`'s functions give Java's results, halves, NaN, infinities and the sign of zero included; as `Float` is a double, `Math.round` of a Float gives a long, of an integer an int.
- Queued and pooled state machines and queued methods take

events or calls on worker threads, as in Java: each class of an object has one for its events and one for its queued methods. A worker runs its calls one at a time in the order they were made, and the event's or method's own call returns at once. A pooled machine first runs an event its current state can take, leaving the others waiting. Automatic transitions run at once. Deleting the object drops what is still waiting. Unlike Java's, a worker with nothing to run ends once the main thread has finished and every other thread left is an idle worker, so a program ends without deleting its objects; a worker starts again if work arrives.

- Timers and do activities of state machines use Python threads. Leaving a state or deleting the object cancels its timers and asks its do activity to stop; an activity that runs for long checks `thread.cancelled.is_set()`, where `thread` is the worker it runs in, and returns when it is set, or pauses with `thread.sleep(milliseconds)`, which returns `True` as soon as the activity is cancelled. An event on a deleted object does nothing and returns `False`; a queued event returns at once, as always.

Python in UmpleOnline: Select Python Code from the Generate menu, then click Execute It. The model needs a main method written in Python. An error in Python code you wrote names its line in your model, except in a method body a trait composes with `superCall`; an error there, or in code Umple translated or generated, names the line of the generated Python.

Command line use: Generating Python needs only Umple (`java -jar umple.jar -g Python model.ump`). Checking the generated Python for syntax errors (the `-c` option) needs Python 3: Umple uses the interpreter named by the environment variable `UMPLE_PYTHON`, or else `python3`, or else `python`. The check never runs the generated code.

Currently unsupported features, reported with error E9210 (the class concerned is then not generated):

- Tracing.
- Distributed classes.
- Static queued methods.
- Custom constructors and `emit` template methods.
- Around injections and injections at code labels.
- Association specialization.

- An inner class whose superclass is an inner class declared elsewhere in the same outer class, or is that outer class (a superclass declared beside it works).
- JSON generation (the genJson suboption).
- A guard testing a state with `is in state`.
- A guard, precondition or invariant that calls a Java library method other than `equals`, `contains` and a String's `length()`, or that names something the model does not define or a member of another class other than a constant.
- A method named like another that is static where the other is not, including a method Python generates for the class.

If any of the features that are supported appear to function incorrectly, please submit a new issue here.

CPlusPlus

The C++ generator is in beta because it has a few bugs in certain aspects. When generating C++ on the command line use the Cpp as the generator name.

C++ code can be embedded in Umple, and Umple is designed to generate C++ from its modeling features.

Setting Up Your Environment for C++ on Linux, Windows, and MacOS

General Information

It is essential to ensure that your development tools are up-to-date and compatible with the latest standards. For CMake the recommended version is 3.12.xx or later. Please consult the CMake Release Notes for the most current version.

Step 1: Install Prerequisites

All Platforms

- Update Package Information: Ensure your system's package list is up-to-date by using the appropriate command for your operating system. This prepares your system for new software installations.
- Install CMake:
  - Linux: Run `sudo apt install cmake -y` in your terminal.
  - Windows: Download it from the CMake official site. For command line installation refer to the same steps as in Linux. Or Install CMake through Visual Studio other IDEs like IntelliJ can also work.
  - MacOS: Use Homebrew to install CMake by running `brew install cmake`.
- Verify Installation: After installation verify the CMake version by typing `cmake --version` in your terminal for Windows Or `cmake -version` for Linux.

Step 2: Obtain and Prepare C++ Code

General Setup

- Obtaining C++ Code: Depending on your development setup you might generate C++ code using different methods:
  - UmpleOnline: Generate C++ code via the web interface and download it as a ZIP file.
  - Command Line: Use Umple with the `-g RTCpp X.ump` option to generate C++ code. The file X.ump refer to your umple file
  - VS Umple Plugin(or other IDE): Generate C++ code directly within the VS environment.
- Preparing the Project: Extract or Access C++ Files: If you have a ZIP file (e.g. from UmpleOnline) extract it to a suitable directory. If you generated files directly on your system or through an IDE navigate to the directory containing your C++ files.
- Set Up Build Directory: Navigate to your project's directory. Create and move into a new directory for the build process by executing:
  - `mkdir build`
  - `cd build`

Step 3: Configure and Build Your Project

All Platforms

- Configure the Project: From the build directory configure your project by running `cmake ..`
- Build the Project: Compile the project by executing `make` on Linux and MacOS or `cmake --build .` on Windows. The dot '.' refers to the current directory which is the build directory.

Step 4: Run Your Executable

- Execute the Program: If the build is successful run the generated executable. Replace *executable* with the actual name of your program:
  - Linux and MacOS: `./executable`
  - Windows: `.\executable.exe`

  where *executable* is the name of the program you compiled.

Ruby

The Ruby generator supports all class diagramming features of Umple, and generates code that is fully tested.

Ruby code can be embedded in Umple, and Umple can generate Ruby features such as classes, attributes and associations. This can save a lot of boilerplate programming for systems with complex class structures. The generated Ruby code can then be combined with other Ruby code written separately.

The Umple developers do not intended to extend the Ruby generator to support state machines, since there are already Gems in Ruby that provide this capability.

Alloy

Alloy is a formal method notation that can be used check properties of models using logic.

The Umple Alloy generator generates statements that can be used to analyse properties of class models involving associations. Both classes (with their attributes) and associations (with their multiplicity) are turned into sig statements in Alloy.

More information about Alloy can be found at the Alloy project home page, or on the Wikipedia Alloy page.

This generator is discussed in detail in Opeyemi Adesina's PhD thesis: Integrating Formal Methods with Model-Driven Engineering.

NuSMV

NuSMV is a formal method notation that can be used check properties of behaviour models using logic.

The Umple NuSMV generator generates statements that can be used to analyse properties of state machine models.

More information about NuSMV can be found at the Alloy project home page, or on the Wikipedia NuSMV page.

This generator is discussed in detail in Opeyemi Adesina's PhD thesis: Integrating Formal Methods with Model-Driven Engineering.

Composite State Table

A composite state table is a state table which, instead of displaying single states and their transitions, displays a combination of multiple concurrent states. They are generated as part of the state table generation.

They can be generated in UmpleOnline through "TOOLS > GENERATE > State Tables" and clicking "Generate It".

They can also be generated through the command-line with "java -jar umple.jar -g StateTables *.ump".

State Machine With Concurrent States

view plain

```
01. class Student {
02.  status {
03.    ParentState1{
04.      changeParent -> ParentState2;
05.      NonConcurrentSubState1 {
06.        changeNonCon -> NonConcurrentSubState2;
07.      }
08.      NonConcurrentSubState2 {
09.        changeNonCon -> NonConcurrentSubState1;
10.      }
11.    }
12.
13.    ParentState2{
14.      changeParent -> ParentState1;
15.      ConcurrentSubState1 {
16.        change1 -> ConcurrentSubState2;
17.      }
18.      ConcurrentSubState2 {
19.        change1 -> ConcurrentSubState1;
20.      }
21.      ||
22.      Css1 {
23.        change2 -> Css2;
24.      }
25.      Css2 {
26.        change2 -> Css1;
27.      }
28.    }
29.   }
```

```
}
// @@@skipcppcompile


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Instance Diagram

Given a class diagram described in umple the instance diagram generator will create a valid random instance diagram from the class diagram.

Instance names and ids

If the first attribute of a class is a String name then the name of the instance will match the randomly generated attribute value.

If the first attribute of a class is id or ends in Id (case sensitive such as productId and not cupid) the instance name will be className + id.

Suboptions

- Instance Links: Determines the amount of links between created instances for any associations with variable multiplicities.
  - randomInstLinks (default): Creates a random valid number of links between generated instances.
  - minInstLinks: Creates the minimum required number of links between generated instances based on their associations.
  - maxInstLinks: Creates the maximum valid number of links between generated instances.
- Initial Instance Count: Determines the number of instances to create for each concrete class.
  - randomInitialInstCount (default): Creates a random number of initial instances for each concrete class. Instances required by associations will still be created.
  - initialInstCount (integer): Creates a minimum number of initial instances for each concrete class based on the passed value.
- maxInstIterations (integer, default: 100): Acts as a timeout for creating many instances dependent on one another. If the diagram is very large with long chains of associations, increase this suboption accordingly.
- maxInstClassCount (integer, default: 200): Determines the maximum number of instances in the generated diagram.

Possible Errors

1. Reached maximum iterations. Ensure that the system is instantiable and non-infinite
   - If this error occurs it is likely that the given system is too large for the given iterations. Increase the maxInstIterations if

a large diagram is needed.

2. Reached maximum class count. Ensure that the system is instantiable and non-infinite
    - If this error occurs the generator attempted to generate more instances than the maxInstClassCount suboption. Either decrease the size of the multiplicities or increase the suboption as needed.
3. Inheritance unresolvable. Ensure that all abstract/interface classes have concrete implementations that can be instantiated
    - The given system has invalid inheritance from associations to abstract classes or interfaces which can't be instantiated.
4. Infinite loop detected. Instantiating the following classes will lead to an infinite number of instances: (list of classes)
    - The given system has multiplicities which lead to an infinite number of class instances. Check the associations between the listed classes.
5. Generated InstanceDiagram is empty. For random diagrams, consider trying again or setting the initial class counts to a constant
    - This occurs when randomInitialInstCount generates no instances at all, and occurs more often with small systems. Set an initialInstCount suboption or try again.

FeatureModelJson

Generates a JSON representation of the feature model defined by require statements and mixset features in an Umple file.

Can be invoked using the command line option `-g FeatureModelJson` or the directive `generate FeatureModelJson;` in an Umple file.

If the Umple file does not define any feature model, the output will contain `{"featureModel":null}`.

When the model has an attached `UmpleModel` with requirements, the output also carries a top-level `requirements[]` array (sibling of `nodes[]` and `links[]`) listing every `Requirement`. Each entry carries the user-story fields (`identifier`, `language`, `statement`, `who`, `when`, `what`, `why`) plus nested `useCaseSteps[]` (with `id`, `stepType`, and `content`) and `qualityClasses[]` (with `name` and `content`). The array is sorted deterministically by `identifier` so `toJSON` / `fromJSON` round-trips remain byte-identical. The `requirements` key is omitted entirely when no `UmpleModel` is attached.

Each `FeatureLeaf` whose `mixsetOrFileNode` is a `Mixset` also carries a `reqImplementations[]` sub-array describing every `implementsReq` binding owned by that mixset. Each entry records `requirementIdentifier`, `qualityClassName` (nullable), `implementingFeature` (nullable), plus two recomputed-on-serialization booleans `requirementResolved` and `qualityClassResolved` that surface whether the parser post-pass successfully bound the entry to its `Requirement` and `QualityClass`. The key is emitted only on Mixset leaves — never on `UmpleFile` leaves nor on `FragmentFeatureLeaf` nodes.

JSON

JSON as a data-interchange format widely used to save and load data in many contexts and languages. It can be used in transmitting over networks. and saving the state of the system persistently.

Umple provides a capability to generate a Json representation of instances of any Umple class, including the complete network of all associated (linked) objects, as well as to reload objects that have been saved. This currently only works with the Java generator for Umple.

To generate functions for serializing and deserializing Json in Umple Java code, specify the suboption "genJson" using the command line or UmpleOnline. A suitable command line would be: *umple [FILENAME].ump -g Java -s genJson -c-* ... this will generated the needed functions and also compile the code.

When specifying the suboption genJson, all generated Java classes will have the following functions as part of their API:

- **String toJson()**. This method generates (serializes) a Json representation of a given object, and all objects connected to it by associations.
- **Object fromJson(String)** This is a static method that instantiates (deserialize) all the objects that had been saved using toJson. Data stored that represents attributes, associations and the current states of state machines is restored.

For each state machine, the Json holds the current state under the name of the state machine, and each nested region is stored the same way under its own name, for example *"status" : "Paid"* and *"statusPaid" : "ReadyToShip"*. When an object is reloaded, its states are set directly: no entry actions or do activities run, because the object is being restored rather than changing state. Timed transitions of the restored states start again with their full delay. Queued and pooled state machines take events as before, but events still waiting in their queue when the Json was generated are not saved, so they start with an empty queue. A state name that the state machine does not have makes fromJson throw an IllegalArgumentException, for the reloaded object and for the objects associated with it. A state missing from the Json (for example in Json saved before state machines were stored) gets the value it has in a new object, without taking automatic transitions, so a nested region is Null unless its enclosing states are start states: *"status" : "Paid"* without *"statusPaid"* leaves statusPaid Null. Keep every state field when editing saved Json. History states are not saved, so after reloading they start as

they do in a new object.

External tools such as GSON can also be used to serialize and deserialize object data, but these tools do not support circular references. An attempt to save data with circular references in GSON will result in the code crashing with an infinite recursion error. A circular reference occurs when one object points to a second object, and the second object points back (possibly indirectly) to the first. Circular references are inherent in the code generated by Umple bidirectional associations and allow very useful and efficient APIs to be present in generated Umple code. Umple's builtin Json generation handles these circular references in persisted Json data by attaching an umpleObjectID to each saved object. When the date is re-loaded into Umple, these umpleObjectID codes are used to ensure that the circular references are regenerated correctly.

Examples: These demonstrate how use toJson() and fromJson() to serialize and de-serialize Json.

This example shows a simple class implementing JSON Serialization

view plain

```
01. suboption "genJson";
02. class Bank {
03.   String bankName;
04.   int numAccount;
05.
06.   public static void main(String[] args) Java{
07.     Bank boc=new Bank("BankCanada",100000);
08.     System.out.println(boc.toJson());
09.   }
10. }
11. // @@@skipcompile   Temporary to bootstrap
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

This example shows a more sophisticated model implementing JSON serialization and deserialization; it generates and outputs the Json, then reloads it, then outputs it again. The results are the same, except for the

UmpleObjectIDs.

view plain

```
suboption "genJson";
class University {
  singleton;
  lazy name;
  lazy Float tuition;
  1 -- * Course;
}

class Person {
  Integer id;
  name;
}

class Student {
  isA Person;
  Float tuitionPaid;
}

class PartTimeStudent {
  isA Student;
  Boolean isExemptFromFees;
}

class Employee {
  isA Person;
  Double salary;
  lazy job;
  * -- * CourseSection teaches;
}

association {
  * Student -- 1 University;
}

class Course {
  title;
}

class CourseSection {
  code;
  * -- 1 Course;
```

```
42. }
43.
44. associationClass Registration {
45.   * Student;
46.   * CourseSection;
47.   lazy grade;
48. }
49.
50. class University {
51.   public static void main (String[] args) Java{
52.
53.     University u = new University();
54.     u.setName("UOttawa");
55.     u.setTuition(1000.00f);
56.
57.     Student s1 = new Student(101, "Tim", 800.0f, u);
58.     Student s2 = new Student(102, "Jane", 2000.0f, u);
59.     Student s3 = new Student(103, "Ali", 750.0f, u);
60.
61.     PartTimeStudent s4= new PartTimeStudent(104, "Marie", 600
62.
63.     Employee e1 = new Employee(202, "Julia", 6585.45);
64.     Employee e2 = new Employee(203, "Robin", 8785.11);
65.     e2.setJob("Advisor");
66.
67.     Course ca= new Course("Architecture",u);
68.     Course cb= new Course("Biology",u);
69. CourseSection csa1= new CourseSection("A1",ca);
70.     e1.addTeache(csa1);
71.     CourseSection csa2= new CourseSection("A2",ca);
72.     e1.addTeache(csa1);
73.     CourseSection csb1= new CourseSection("B1",cb);
74.     e2.addTeache(csa1);
75.     e1.addTeache(csa1);  // two teachers for this
76.
77.     new Registration(s1,csa1);
78.     new Registration(s1,csb1);
79.     new Registration(s2,csa2);
80.     new Registration(s2,csb1);
81.     new Registration(s2,csa1);
82.     new Registration(s3,csb1);
83.     new Registration(s4,csa2);
84.     String jsonString = u.toJson();
85.     University u2 = University.fromJson(jsonString);
86.
```

```
    System.out.println("----- fromJson object -------\n"+u2);

    System.out.println("----- fromJson --->  toJson ------");
    System.out.println(u2.toJson());

  }
}
// @@@skipcompile   Temporary to bootstrap


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

This example shows an object with a nested state machine being saved and reloaded; the reloaded object continues from the state it was saved in.

view plain

```
suboption "genJson";
class Order {
  String orderId;
  status {
    Placed {
      pay -> Paid;
    }
    Paid {
      ship -> Shipping;
      Packing { pack -> ReadyToShip; }
      ReadyToShip {}
    }
    Shipping {}
  }

  public static void main(String[] args) Java{
    Order order = new Order("A-17");
    order.pay();
    order.pack();
    String jsonString = order.toJson();
    System.out.println(jsonString);

    // The reloaded order is in Paid.ReadyToShip and continues fro
    Order reloaded = Order.fromJson(jsonString);
```

```
25.     System.out.println(reloaded.getStatusFullName());
26.     reloaded.ship();
27.     System.out.println(reloaded.getStatusFullName());
28.   }
29. }
30.
31.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Umplification - An Introduction

Umplification involves transforming step by step a base language program to an Umple program; each step is a refactoring. The starting point and the ending point of each transformation will be a system rendered in a textual form combining model elements in the Umple syntax and base language code.

Umplification involves increasing the proportion of modeling elements.

The key insight is that a pure Java program can be seen as a special case of an Umple program. A pure model can also be seen as such a special case. So Umplification involves repeatedly modifying the Umple to add abstractions, while maintaining the semantics of the program, and also maintaining to the greatest extent possible such elements as layout, algorithmic methods and comments.

Two main features differentiate Umple from existing reverse engineering techniques.

1. The **Tranformations** required are intended to be applied **incrementally** by a programmer who has a body of legacy code and wants to gradually transform it into Umple, preserving much of the layout, comments and other aspects of the original code if possible.
2. The transformations required are at the same time **code-to-model**,**model-to-model** and **code-to-code**.

Umplification Process

To start, source files with language L (e.g. Java, C++) code are initially renamed as Umple files, with extension **.ump**.

Then, iteratively, small transformations are performed that gradually add UML abstractions. Each iteration should involve testing to check that the program's semantics are preserved.

Umplification can be performed automatically by a reverse engineering technology that can parse a program in any native programming language and extract the Umple model from it.

Umplification Tooling

The **Umplificator** is a reverse engineering tool for the incremental automatic detection of opportunities for refactorings and model transformations in source code consisting of Umple or an Umple base language such as Java.

To transform a program written in Java into Umple you can use one of the following tools:

- **Eclipse**: To use the Umplificator with Eclipse, you need to load the the Umplificator plugin (cruise.umplificator.eclipse.X.X.X.jar). You can build this from sources or contact the Umple team for a fresh build.

- **Command-line based compiler**: This can be run as "java -jar umplificator.jar *.java".
  Note that it is also possible to umplify files with extension ".ump".

Interfacing to External Code

This example shows some of the techniques that can be used when writing Umple code that will be connected to code written separately, such as code in a GUI library. Use of the:

- internal keyword to specify an attribute for which getters and setters will not be generated (i.e. treated as a standard instance variable)
- external keyword to indicate a class that has been compiled separately.
- isA keyword to have a class implement an interface
- before and after keywords to inject code to link two attributes
- 'after constructor' to inject additional constructor code

Example

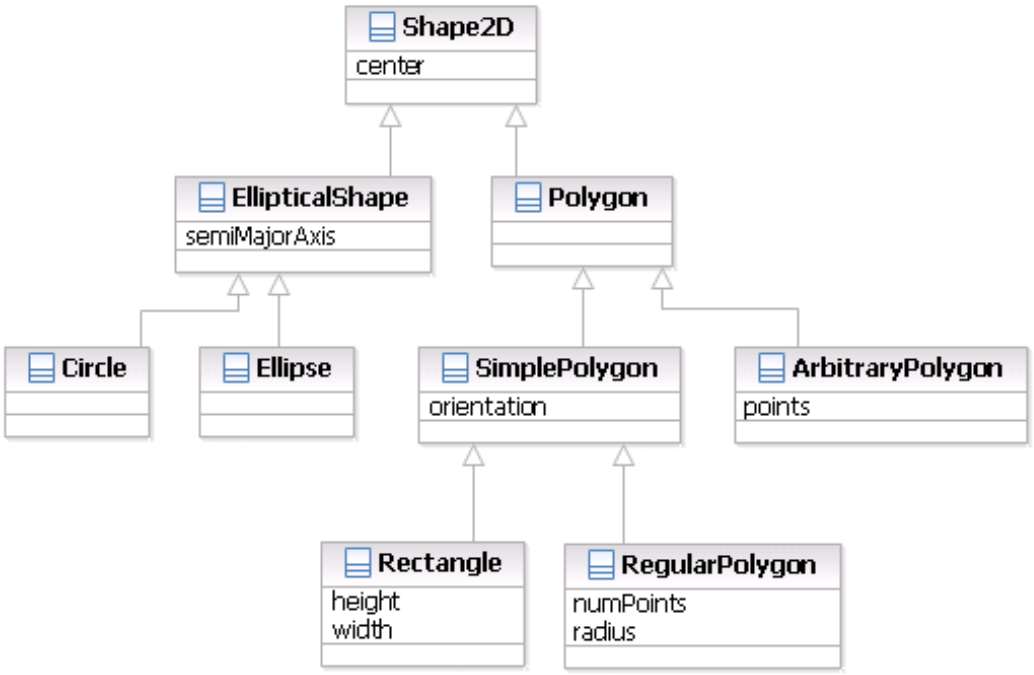
view plain

```
01. /*
02. This example demonstrates an example of using the
03. internal, depends, external, before and after
04. keywords to interface umple code to code in
05. another system.
06.
07. The internal keyword is suitable for situations
08. where the developer doesn't want setter/getters
09. to be generated. GUI windows are an example of
10. such a case. A GUI window/unit contains several
11. components but the rest of the program may only
12. need the value stored within those components. So
13. it's a good practice to hide UI components from
14. classes other than the containing one.
15.
16. */
17.
18. // required to make HelloInternals class a JFrame
19. external JFrame{}
20.
21. class HelloInternals {
22.   // HelloInternals extends JFrame
23.   isA JFrame;
24.
25.   // importing required classes
26.   depend javax.swing.JFrame;
27.   depend javax.swing.JLabel;
28.
```

```
29.   /* messageLabel is a component of the frame;
30.   often we don't want sub-components of a GUI unit
31.   to be settable/gettable. By making them internal
32.   Umple will avoid generating setter/getter for
33.   messageLabel. Using lazy Umple will avoid adding
34.   a constructor parameter for this component */
35.   lazy internal JLabel messageLabel;
36.
37.   // the contents of messageLabel
38.   String message;
39.
40.   /* before getting the message, this code updates
41.   the message attribute using the text from
42.   messageLabel */
43.   before getMessage {
44.     message=messageLabel.getText();
45.   }
46.
47.   /* after setting the messae, this code updates
48.   messageLabel to contain the newly updated
49.   message */
50.   after setMessage {
51.     messageLabel.setText(message);
52.   }
53.
54.   // using after constructor is a good way of
55.   // initiating a GUI unit
56.   after constructor {
57.     getContentPane().setLayout(null);
58.
59.     messageLabel=new JLabel(message);
60.     messageLabel.setBounds(10, 10, 200, 20);
61.     add(messageLabel);
62.
63.     pack();
64.     setSize(250, 200);
65.   }
66. }
67.
68.
69.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

2D Shapes System



Example

view plain

```
01. // 2D Shapes class hierarchy - sample UML class
02. // diagram in Umple
03. // From Book by Lethbridge and Laganiere, McGraw Hill 2004
04. // Object-Oriented Software Engineering:
05. //   Practical Software Engineering using UML and Java
06. // See https://www.site.uottawa.ca/school/research/lloseng/
07.
08. //Namespace for core of the system.
09. namespace Shapes.core;
10.
11. class Shape2D {
12.  center;
13. }
14. //Abstract
15. class EllipticalShape {
16.  isA Shape2D;
17.  semiMajorAxis;
18. }
19. //Abstract
```

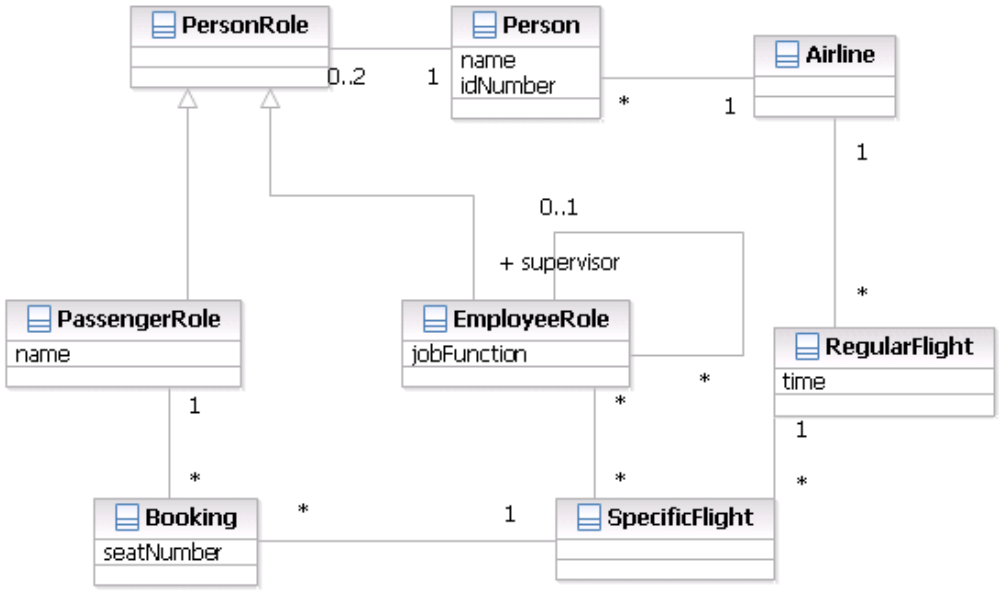
```
20. class Polygon {
21.  isA Shape2D;
22. }
23. class Circle {
24.  isA EllipticalShape;
25. }
26. class Ellipse{
27.  isA EllipticalShape;
28. }
29. class SimplePolygon {
30.  orientation;
31.  isA Polygon;
32. }
33. class ArbitraryPolygon {
34.  points;
35.  isA Polygon;
36. }
37. class Rectangle {
38.  isA SimplePolygon;
39.  height;
40.  width;
41. }
42. class RegularPolygon {
43.  numPoints;
44.  radius;
45.  isA SimplePolygon;
46. }
47.
48. class Shape2D {
49.   public static void main(String [] argc) Java {
50.     Shape2D s = new
51.       RegularPolygon("0,0","0","3","100");
52.     System.out.println(s);
53.   }
54.
55.   public static void main(String [] argc) Python{
56.     s = RegularPolygon("0,0","0","3","100")
57.     print(str(s))
58.   }
59. }
60.
61.
```

Load the above code into classic UmpleOnline

Load the above code into UmpleOnline V2

Airline System

A simple airline system that manages flight and passenger information



Example

view plain

```
01. // Airline system - sample UML class diagram in Umple
02. // From Book by Lethbridge and Laganiere, McGraw Hill 2004
03. // Object-
    Oriented Software Engineering: Practical Software Engineering us
04. // See https://www.site.uottawa.ca/school/research/lloseng/
05.
06. namespace Airline;
07.
08. class Airline{
09.   1 -- * RegularFlight;
10.   1 -- * Person;
11. }
12.
13. class RegularFlight{
14.   Time time;
15.   unique Integer flightNumber;
16.   1 -- * SpecificFlight;
17. }
18.
```

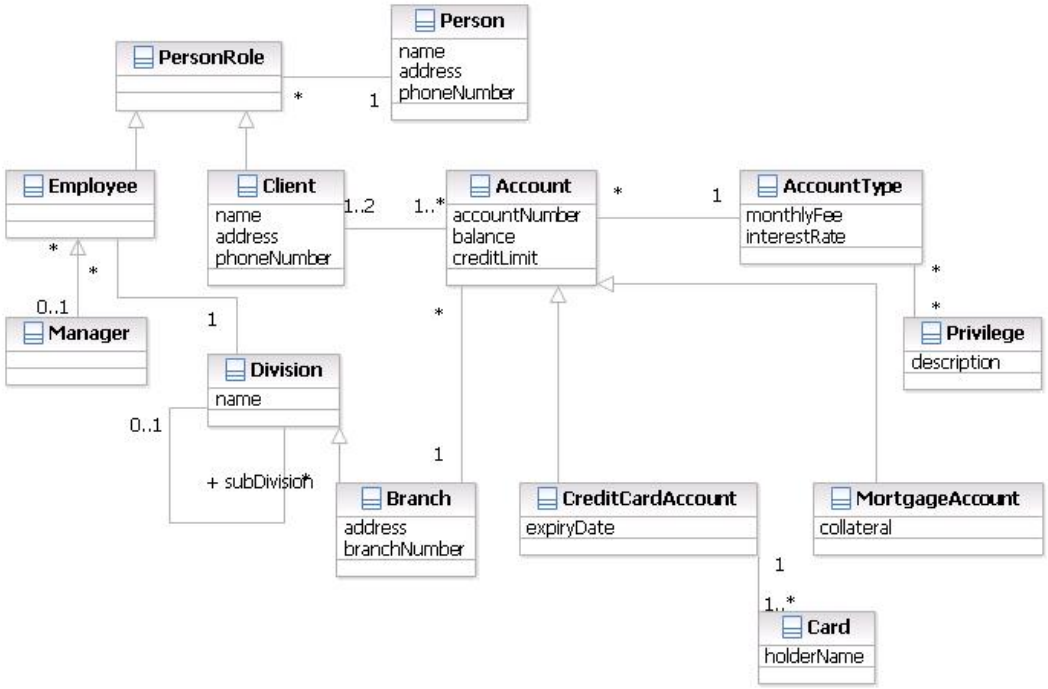
```
19. class SpecificFlight{
20.   unique Date date;
21. }
22. 
23. class PassengerRole
24. {
25.   isA PersonRole;
26.   immutable String name ;
27.   1 -- * Booking;
28. }
29. 
30. 
31. class EmployeeRole
32. {
33.   String jobFunction ;
34.   isA PersonRole;
35.   * -- 0..1 EmployeeRole supervisor;
36.   * -- * SpecificFlight;
37. }
38. 
39. class Person
40. {
41.   settable String name;
42.   Integer idNumber;
43.   1 -- 0..2 PersonRole;
44. }
45. 
46. class PersonRole{}
47. 
48. class Booking{
49.   String seatNumber;
50.   * -- 1 SpecificFlight;
51. }
52. 
53. class Airline {
54.   // Sample output to illustrate manipulating the model
55.   depend java.sql.*;
56.   public static void main(String [] argc) {
57.     Airline a = new Airline();
58.     RegularFlight f100 = new RegularFlight(
59.       Time.valueOf("10:15:00"),100,a);
60.     SpecificFlight f100jan4 = new SpecificFlight(
61.       java.sql.Date.valueOf("2025-01-04"),f100);
62.     System.out.println(f100);
63.     System.out.println(f100jan4);
```

```
  }
}


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Banking System



Example

view plain

```
01. // Sample UML class diagram for a banking system, written in Um
02.
03. //Namespace for core of the system.
04. namespace BankingSystem.core.humanResources;
05. class PersonRole{}
06.
07. class Person{
08.  name;
09.  address;
10.  phoneNumber;
11.
12.  1 -- * PersonRole;
13. }
14.
15. class Employee{
16.  isA PersonRole;
17. }
18.
19. class Client
20. {
```

```
 isA PersonRole;
 name;
 address;
 phoneNumber;
 1..2 -- 1..* Account;
}

class Manager {
 isA Employee;
 0..1 -- * Employee;
}

//Accounts,  priviledges, etc.
namespace BankingSystem.core.intangableResources;
class Account{
 Integer accountNumber;
 Float balance;
 Float creditLimit;
 * -> 1 AccountType;
}

class AccountType
{
 Float monthlyFee;
 Float interestRate;

 * -- * Privilege;
}

class Privilege
{
 description;
}

class CreditCardAccount{
 isA Account;
 Date expiryDate;

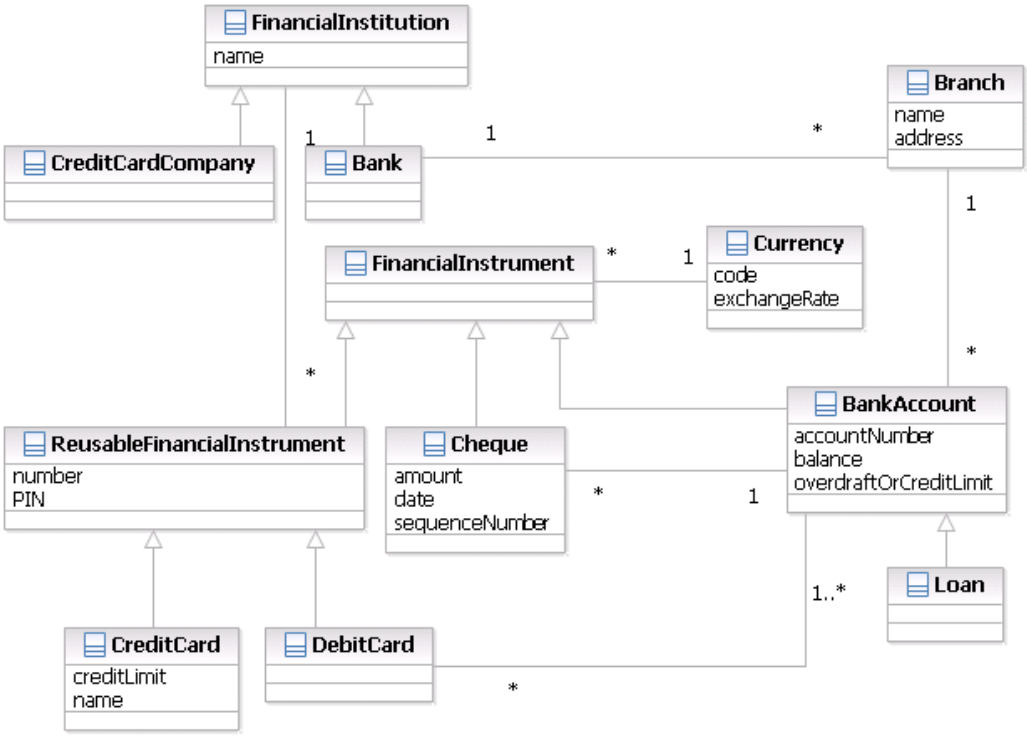
 1 -- 1..* Card;
}

class MortgageAccount {
 isA Account;
 collateral;
}
```

```
66.
67.  //Anything physically tangable
68.  namespace BankingSystem.core.tangableResources;
69.  class Card
70.  {
71.   holderName;
72.  }
73.
74.  class Branch {
75.   isA Division;
76.   address;
77.   branchNumber;
78.
79.   1 -- * Account;
80.  }
81.
82.  class Division{
83.   name;
84.
85.   1 -- * Employee;
86.   0..1 -- 0..* Division subDivision;
87.  }
88.
89.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Banking System (b)



Example

view plain

```
01. /*
02.   Banking System - sample UML class diagram written in Umple
03.   Last updated: February 21, 2011
04. */
05. //Namespace for core of the system.
06. namespace BankingSystem.core;
07.
08. class FinancialInstitution {
09.  name;
10.  1 -- * ReusableFinancialInstrument;
11. }
12.
13. class CreditCardCompany{
14.  isA FinancialInstitution;
15. }
16.
17. class Bank{
18.  isA FinancialInstitution;
19.  1 -- * Branch;
```

```
}

class FinancialInstrument{
}

class ReusableFinancialInstrument{
 isA FinancialInstrument;
 number;
 pin;
 * -> 1 Currency;
}

class CreditCard {
 isA ReusableFinancialInstrument;
 creditLimit;
 name;
}

class DebitCard {
 isA ReusableFinancialInstrument;
}

class Cheque {
 isA FinancialInstrument;
 amount;
 Date date;
 sequenceNumber;
}

class BankAccount{
 isA FinancialInstrument;
 accountNumber;
 balance;
 Float overdraftOrCreditLimit;

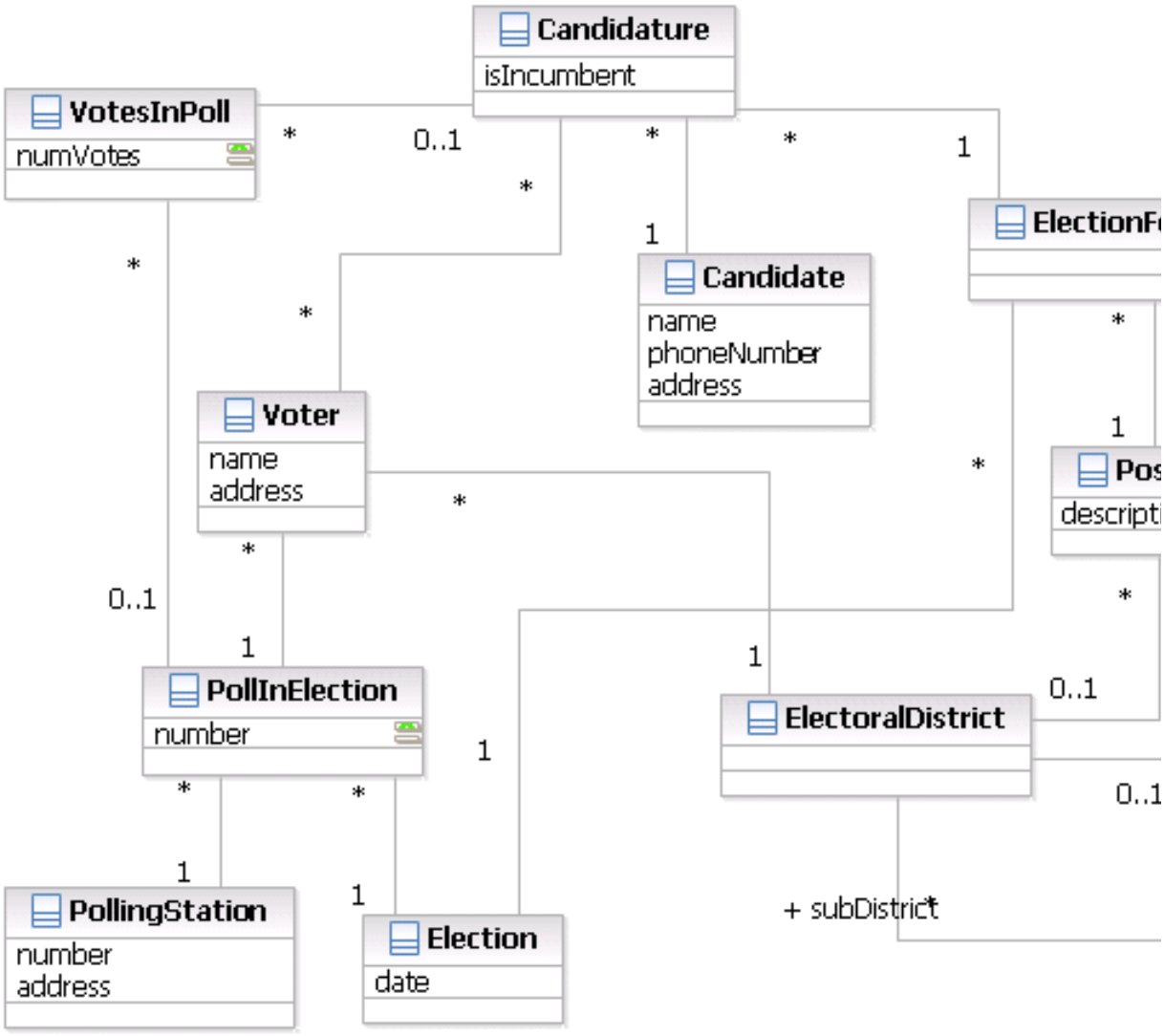
 1..* -- * DebitCard;
 1 -- * Cheque;
}

class Currency
{
 code;
 exchangeRate;
}
```

```
65. class Branch
66. {
67.  name;
68.  address;
69.  1 -- * BankAccount;
70. }
71.
72. class Loan{
73.  isA BankAccount;
74. }
75.
76.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Election System



Example

view plain

```
01. // UML class diagram for a system for managing elections, writter
02. namespace electorial;
03.
04. // association should be ->
05. class PollingStation {
06.  Integer number;
07.  address;
08.
09.  1 -- * PollInElection;
```

```
}

class ElectedBody{
 description;

 1 -- * Position;
}

 //Different elections may have different sets of polls
class Election{
 Date date;

 1 -- * PollInElection;
 1 -- * ElectionForPosition;
}

class ElectionForPosition{
 1 -- * Candidature;
}

//Eg. Mayor, Councilor. AKA seats
//
A position can have different elections for it at different times, eg.
class Position {
 description;
 1 -- * ElectionForPosition;
}

//
We need candidature class since a candidate can run for different
//positions and for the smae positions at subsequent elections
class Candidature {
 internal Boolean isIncumbent = false;

 public void markAsIncumbent()
 {
   isIncumbent = true;
 }

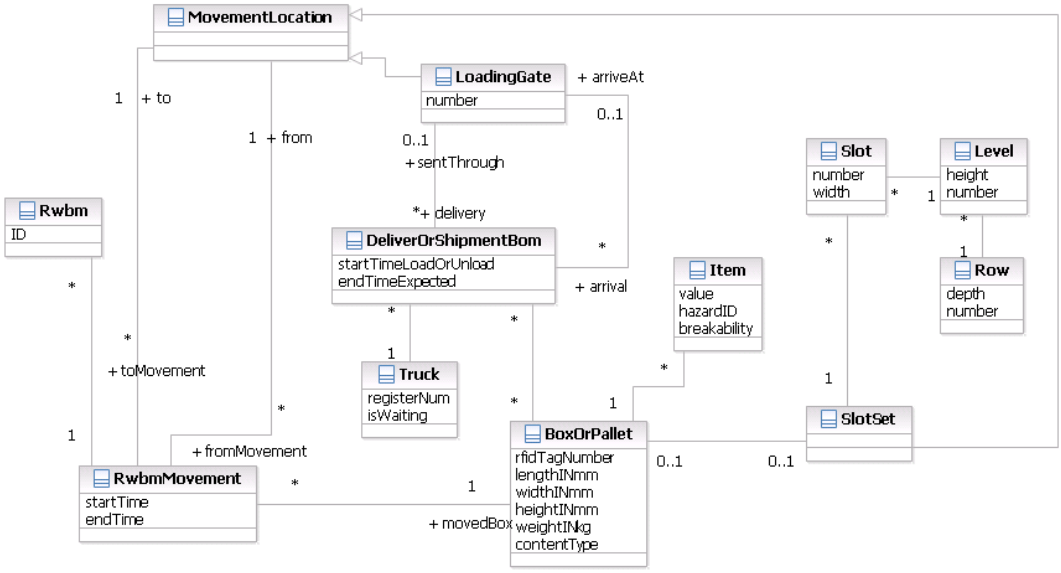
 public String toString()
 {
   return isIncumbent ? "Incumbent" : "Candidature";
 }
}

```

```
53. class Candidate {
54.  name;
55.  Integer phoneNumber;
56.  address;
57. 
58.  1 -- * Candidature;
59. }
60. 
61. class PollInElection {
62.  Integer number;
63. 
64.  1 -- * Voter;
65. }
66. 
67. associationClass VotesInPoll{
68.  * Candidature;
69.  * PollInElection;
70.  Integer numVotes;
71. }
72. 
73. class Voter{
74.  name;
75.  address;
76.  * -- * Candidature;
77. }
78. 
79. class ElectoralDistrict{
80.  1 -- * Voter;
81.  0..1 -- * Position;
82.  0..1 -- * ElectoralDistrict subDistrict;
83. }
84. 
85. 
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Warehouse System



Example

view plain

```
01. // UML class diagram of a system for managing a warehouse
02. namespace warehouse;
03.
04. class MovementLocation{}
05.
06. class SlotSet{
07.  isA MovementLocation;
08.  0..1 -- 0..1 BoxOrPallet;
09.  1 -- * Slot;
10. }
11.
12. class Slot{
13.  Integer number;
14.  Double width;
15. }
16.
17. class Level{
18.  Integer height;
19.  Integer number;
20.  1 -- * Slot;
21. }
22.
23. class LoadingGate{
```

```
24.
25.  Integer number;
26.  isA MovementLocation;
27. }
28.
29. class Truck{
30.  Integer registerNum;
31.  Boolean isWaiting;
32.
33.  1 -- * DeliverOrShipmentBom;
34. }
35. class Rwbm{
36.  Integer id;
37. }
38.
39. class Item{
40.  Double value;
41.  Integer hazardID;
42.  Integer breakability;
43. }
44.
45. class Row{
46.  Integer depth;
47.  Integer number;
48.
49.  1 -- * Level;
50. }
51.
52. //Note that here we have the choice to use
53. // Time or Double, if the customer requires Double due to
54. // some interoperability issues with a legacy syste, we can easily
55. // accomodate them and use Double instead of Time.
56. class RwbmMovement{
57.  Double startTime;
58.  Double endTime;
59.
60.  * toMovement -- 1 MovementLocation to;
61.  * fromMovement -- 1 MovementLocation from;
62.  * -- 1 BoxOrPallet movedBox;
63.  1 -- * Rwbm;
64. }
65.
66. // Both associations should be ->
67. class DeliverOrShipmentBom {
68.  Double startTimeLoadOrUnload;
```

```
69.   Double endTimeExpected;
70.   * delivery -- 0..1 LoadingGate sentThrough;
71.   * arrival -- 0..1 LoadingGate arriveAt;
72.  }
73.
74.  class BoxOrPallet{
75.   Integer rfidTagNumber;
76.   Double lengthINmm;
77.   Double widthINmm;
78.   Double heightINmm;
79.   Double weightINkg;
80.   String contentType;
81.
82.   * -- * DeliverOrShipmentBom;
83.   1 -- * Item;
84.  }
85.
86.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Canal System

Canal system class diagram.

See also the state diagram for the canal lock.

The following are the requirements from which this was derived:

The Canal Monitoring and Control System is used to automate canal locks in response to water craft that want to pass through a canal system. The system can be installed in any network of canals. Each canal is divided up into segments. Each segment has two ends which can either be a bend, a lock gate, a low bridge, or a system entry/exit point (where boats can move to or from other waterways such as rivers and lakes). A special kind of canal segment is a lock. The water level in a lock can be raised or lowered so its height matches the height of an adjacent segment. Several locks can exist side by side (as in the Rideau Canal next to Parliament Hill in Ottawa). A low bridge is a location where the bridge needs raising for a boat to pass. A bend is simply a place where the canal changes direction - keeping track of these helps simplify mapping of the system.

The system keeps track of the GPS co-ordinates of each segment end, the height of the water above sea level in each segment (which can change in locks), and optionally a name given to a series of locks or segments.

Each water craft using the system must have a transponder. The transponder includes a GPS unit and transceiver. When the captain of a craft wants to travel through the system, he enters his plannned destination in the transponder. The size of the boat is also tracked. The transponder regularly transmits the location of the craft to the control system so the control system can monitor traffic.

When a craft reaches a gate, the system knows it is there and needs to pass through since it has received the data about the desired destination. The control system takes care of opening the gate, raising or lowering of the water level, and then opening the next gate, etc. When a boat reaches a low bridge, a similar procedure takes place, except that raising or lowering the water doesn't happen.

When there are a lot of boats, the system makes several of them queue up so they can pass through a lock or low bridge at the same time. The system has, however, to ensure that only the right number and size of boats are put into a lock at once, so the lock doesn't become overly full (the system knows the size of each lock). Complicating matters is the fact that there will be boats travelling in both directions.

Example

view plain

```
01. // UML class diagram that models a canal as a
02. // network of segments, with Locks. See the
03. // State Machine section for a state machine for
04. // each lock
05. class CanalNetwork {
06.   name;
07.   0..1 -- * CanalNetwork subNetwork;
08.   0..1 -- * Craft activeVessels;
09.   * -- * SegEnd;
10. }
11.
12. class SegEnd {
13.   name;
14.   GPSCoord location;
15. }
16.
17. class Segment {
18.   Float waterLevel; // m above sea level
19.   1..* -- 2 SegEnd;
20. }
21.
22. class Lock {
23.   isA Segment;
24.   Float maxLevel;
25.   Float minLevel;
26. }
27.
28. class Bend {
29.   isA SegEnd;
30. }
31.
32. class EntryAndExitPoint {
33.   isA SegEnd;
34. }
35.
36. class MooringPoint {
37.   isA SegEnd;
38. }
39.
40. class Obstacle {
41.   isA SegEnd;
```

```
  0..1 downstreamObstacle -- * Craft upStreamQueue;
  0..1 upstreamObstacle -- * Craft downStreamQueue;
}

class LowBridge {
  isA Obstacle;
}

class LockGate {
  isA Obstacle;
}

class Craft {
  lazy GPSCoord location;
}

class Trip {
  0..1 -> 1..* SegEnd;
  0..1 -- 1 Craft;
}

class Transponder {
  id;
  0..1 -- 0..1 Craft;
}

class GPSCoord {
   Float lattitide;
   Float longitude;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Canal Lock State Machine

Canal Lock State Machine

See also the class diagram and requirements for the canal system.

Example

view plain

```
01. // UML state machine diagram for a canal lock,
02. // represented in UML
03.
04. class Lock
05. {
06.   Boolean boatGoingDown = false;
07.   Boolean boatGoingUp = false;
08.   Boolean boatBlockingGate = false;
09.
10.   lockState {
11.     BothDoorsClosedLockFull {
12.       // Waiting for boat
13.       boatRequestsToEnterAndGoDown
14.         -> OpeningUpperGate;
15.       boatRequestsToEnterAndGoUp
16.         -> LoweringWater;
17.     }
18.
19.     OpeningUpperGate {
20.       upperGateFullyOpen -> UpperGateOpen;
21.     }
22.
23.     UpperGateOpen {
24.       entry / {setBoatGoingUp(false);}
25.       boatInLockRequestingToGoDown
26.         -> / {setBoatGoingDown(true);}
27.         ClosingUpperGate;
28.       after(180000) [!boatBlockingGate]
29.         -> ClosingUpperGate;
30.     }
31.
32.     ClosingUpperGate {
33.       upperGateFullyClosed [boatGoingDown]
34.         -> LoweringWater;
35.       upperGateFullyClosed [!boatGoingDown]
```

```
36.         -> BothDoorsClosedLockFull;
37.     }
38.
39.     LoweringWater {
40.       waterLevelMatchesDownStream
41.         -> OpeningLowerGate;
42.     }
43.
44.
45.     BothDoorsClosedLockEmpty {
46.       // Waiting for boat
47.       boatRequestsToEnterAndGoUp
48.         -> OpeningLowerGate;
49.       boatRequestsToEnterAndGoDown
50.         -> RaisingWater;
51.     }
52.
53.     OpeningLowerGate {
54.       lowerGateFullyOpen -> LowerGateOpen;
55.     }
56.
57.     LowerGateOpen {
58.       entry / {setBoatGoingDown(false);}
59.       boatInLockRequestingToGoUp
60.         -> / {setBoatGoingUp(true);}
61.           ClosingLowerGate;
62.       after(180000) [!boatBlockingGate]
63.         -> ClosingLowerGate;
64.     }
65.
66.     ClosingLowerGate {
67.       lowerGateFullyClosed [boatGoingUp]
68.         -> RaisingWater;
69.       lowerGateFullyClosed [!boatGoingUp]
70.         -> BothDoorsClosedLockEmpty;
71.     }
72.
73.     RaisingWater {
74.       waterLevelMatchesUpStream
75.         -> OpeningUpperGate;
76.     }
77.   }
78. }
79.
80.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Card Games

This system will allow players to play card games such as Whist and Oh Hell online.

The system should manage the cards in a deck of cards, the players (including who is dealer), the hands of cards each player has, the tricks, the pile of cards on the table. It should also manage the matches between players. Use some background research to improve these simple requirements.

Example

view plain

```
01. // Umple class diagram representing card games
02. // in the Oh Hell and Whist family
03.
04. class Card {
05.   suit;  // "hearts", "clubs", "diamonds", "spades"
06.   rank;  // A23456789JQK
07.   Boolean isJoker; // suit and rank are null for a Joker
08. }
09.
10. class CardSet {
11.   // Could be a complete deck or a trick
12.   1 -- * Card;
13. }
14.
15. class NotDealtCardSet {
16.   // This would be initialized with a complete 52-card set
17.   // at the start of each game
18.   // During dealing, the cards would be distributed to players
19.   isA CardSet;
20. }
21.
22. class Trick {
23.   // A group of cards with one contributed by each player, and
24.   // eventually won by a player
25.   // One is built after each player has played a card
26.   isA CardSet;
27. }
28.
29. class Player {
30.    // These are the people playing.
```

```
31.     name;
32.
33.     // A hand is dealt at the start of a game
34.     // The cards a player has in his or her hand
35.     0..1 -- * Card hand;
36.
37.     // The tricks won by a player this game
38.     1 -- * Trick;
39.
40.     // All hands and tricks are cleared at the end of each game
41.     // ready for dealing again
42.
43.     // The following is used only in Oh Hell
44.     Integer currentTricksBid;
45.   }
46.
47.   class ScoringTeam {
48.     // for Oh Hell, it is each player for himself, so there is one playe
49.     // for Whist there are partners, so there are two players
50.     1 -- 1..2 Player;
51.
52.     // The score that each team has accumulated so far
53.     // based on tricks taken or difference between tricks and bid
54.     Integer score;
55.   }
56.
57.   class CardsMatch {
58.    // A cards match is played in a number of games
59.    Boolean isWhist; // True if whist; false if Oh Hell
60.
61.    // The following determines the players
62.    // the first player in the first team deals first
63.    * -- 2..* ScoringTeam; // Exactly 2 if Whist
64.
65.    Integer scoreToWin; // The score agreed to in order to declare w
66.
67.    1 --
       * Game games; // Games continue until a player or team wins
68.    // Assume that the last game in games is the current one
69.   }
70.
71.   class Game {
72.     trumps; // a suit (if any) that is the trumps for this game
73.
74.     // One player is the dealer each game
```

```
75.     // The dealer deals out all or most of the cards at the start
76.     // Players take turn laying down cards to form the next trick
77.     * -- 1 Player dealer;
78.
79.     // As the cards are laid down the following set is created.
80.     // After all players have contributed a new trick is created from
81.     // following and awarded to a player
82.     0..1 -- * Card currentTrickBeingBuilt;
83.
84.     * gameLed --
        0..1 Player currentLead; // the player next to lay down a card
85.   }
86.
87.
88.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Pizza Delivery

Requirements for this system are as follows:

A take-out pizza restaurant wants to set up an online ordering system. A customer must have an account to use the system. When the customer creates his or her account, the following information is stored: Email address (which becomes the user id), contact phone number, password, name, address, preferred credit card number, and credit card expiry date. When the customer creates an order the following information is stored: The time the order was placed, the address for delivery, the contact phone number, the total price, the credit card number charged, the expiry date of the credit card, the items ordered and the total price. An item can be pizza or drinks.

For each pizza item, the information stored will include the kind of pizza (thin crust, thick crust or gluten-free crust), the size (small, medium, large), the list of toppings (e.g. cheeze, bacon, vegetables, etc.), the number of items like this (e.g. 10 would mean 10 identical pizzas) and the total price for this pizza item. For each drink item, the information stored is type, size, number, and total price. The system also records each delivery: Associated with each delivery is the name of the delivery driver; the time the driver picked up the order(s) and the time each order was delivered. A driver may take more than one order on a delivery run.

Example

view plain

```
01. // UML Class diagram representing a system for taking online ord
02.
03. class Account
04. {
05.   emailAddress;
06.   contactPhoneNumber;
07.   password;
08.   name;
09.   address;
10.   preferredCredCard;
11.   expiryDate;
12.   0..1 -- * Order;
13. }
14.
15. class Order
16. {
```

```
  timePlaced;
  contactPhoneNumber;
  Float totalPrice;
  creditCardCharged;
  expiryDate;
  1 -- * OrderItem;
}

class OrderItem
{
  Integer number;
  Float totalPrice;
}

class PizzaOrder
{
    isA OrderItem;
kind;
  * -> * ToppingType toppings;
  * -> 1 StandardPizzaSize;
}

class ToppingType
{
  description;
}

class StandardPizzaSize
{
  sizeLabel;
}

class DrinkOrder
{
    isA OrderItem;
    drinkSize;
size;
}

class Delivery
{
  Time timePickedUp;
  Time timeDelivered;
  * -- 0..1 Driver;
  0..1 -- 1..* Order;
```

```
62. }
63.
64. class Driver
65. {
66.   name;
67. }
68.
69.
70.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Community Association

Requirements for this system are as follows:

You are in charge of the website of a community. Residents in the community can sign up to join the community association for a fee of $30. If they do, then they can vote for the executive of the association, can use facilities and attend certain events for free. Only one membership is required per residence and it must be renewed by the end of October each year (it is valid for one year). For each member the system stores the street address, apartment number (if there is one) the email address, the telephone number, and the names of the residents. Anyone under 18 is flagged, as there are special events for these people. The community runs a rink in the winter and three tennis courts in the summer. Members of the community association are given priority to sign up for blocks of time in these facilities, to a maximum of four hours per week. Residents who are not members can sign up 24 hours before their booked time, if there are still any free time slots. When a non-member signs up in this way, they need to give their name, street address and email address. Some time slots in the rink and tennis courts are left unbooked for free 'first-come-first-served' access.

Example

view plain

```
01. // UML class diagram in Umple showing a simple
02. // system for managing the data maintained by a
03. // community association
04.
05. class Facility
06. {
07.   id;
08.   type;
09.   openPeriod;
10.   1 -- * Booking booked;
11. }
12.
13. class Resident
14. {
15.   name;
16.   Boolean under18;
17.   emailAddress;
18.   telephoneNumber;
19.   executivePosition;
```

```
20. }
21.
22. class CommunityResidence
23. {
24.   streetAddress;
25.   Integer apartmentNumber;
26.   feePaidToDate;
27.   1 -- * Resident;
28. }
29.
30. class Event
31. {
32.   Date date;
33.   Time time;
34.   Float fee;
35.   name;
36. }
37.
38. class Booking
39. {
40.   Date date;
41.   Time startTime;
42.   Time endTime;
43.   Float feePaid;
44.   Boolean isReservedForFCFS;
45.   * -- 0..1 Resident;
46. }
47.
48. class Under18Event
49. {
50.   isA Event;
51. }
52.
53. class Rink
54. {
55.   isA Facility;
56. }
57.
58. class TennisCourt
59. {
60.   isA Facility;
61. }
62.
63. class CommunityAssociation
64. {
```

```
65.     singleton;
66.     1 -- * CommunityResidence;
67.     1 -- * Facility;
68.     1 -- * Event;
69.   }
70.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Co-Op Education

Requirements for a University System to Manage Co-Op Programs

The co-op system works as follows. Employers contact the university and list potential jobs. Each job has a description, employer, location, start and end date, and a list of programs (e.g. SEG, CSI, MCG) for which it is applicable.

Each student in the system has a name, student number, list of available time slots (when they are not in courses, or not doing interviews), resume and transcript.

Students in various programs apply for jobs. Students choose a set of jobs they are interested in. The system shows the list of interested students to employers, and employers pick a set of students to interview. The system sets interview times that match the employee's and student's availability (set of available time slots).

At the end of the process, the system arranges for the employer to offer each job to a student. If the first student does not accept, the system arranges for other students (the employer's second, third choices, etc.) to be offered the job.

Example

view plain

```
01. // UML class diagram in Umple showing the data managed by
02. // a co-
    operative education system that has to place students with emplo
03.
04. class Program
05. {
06.   name;
07.   1..* -- * Student;
08. }
09.
10. class Application
11. {
12.   * -- 1 Student;
13.   * -- 1 Job;
14.   1..* -- 0..1 Interview;
15.   1 -- 0..1 Offer;
16. }
17.
```

```
18. class Job
19. {
20.   description;
21.   location;
22.   Date startDate;
23.   Date endDate;
24.   * -- 1..* Program;
25. }
26. 
27. class Employer
28. {
29.   name;
30.   1 -- * Job;
31.   0..1 -> * TimeSlot;
32. }
33. 
34. class Interview
35. {
36.   location;
37.   0..1 -> 0..1 TimeSlot;
38. }
39. 
40. class TimeSlot
41. {
42.   startTime;
43.   endTime;
44.   status;
45. }
46. 
47. class Student
48. {
49.   Integer stNum;
50.   1 -- 1 Transcript;
51.   0..1 -> * TimeSlot;
52. }
53. 
54. class Resume
55. {
56.   text;
57.   0..1 -- 1 Student;
58. }
59. 
60. class Transcript
61. {
62.   text;
```

```
63. }
64.
65. class Offer
66. {
67.   ranking;
68.   Boolean accepted;
69. }
70.
71.
72.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Vending Machine Classes

The following are basic requirements for a vending machine.

A vending machine dispenses a variety of products. Each product is held in one or more dispensers, identified by slot and row in the machine. The system keeps track of the number of each product type in each dispenser, so it knows when a dispenser becomes empty.

The vending machine has several coin holders. Each coin holder can contain a specific type of coin worth a particular number of cents, and with a certain weight and diameter. The system tracks the number of coins in each holder, so it can determine whether it can give change, and what coins it should give as change when somebody buys a product. Each holder also has a maximum capacity. The system keeps an electronic record of each successful purchase (transaction); it records the product dispensed from a dispenser and the number of coins added (i.e. paid by the customer) or deleted (i.e. given as change) from coin holders.

Example

view plain

```
01. // UML class diagram of a vending machine
02. class VendingMachine {
03.    singleton;
04.    1 -- * Dispenser slots;
05. }
06.
07. class Dispenser {
08.   Character row;
09.   Character column;
10. }
11.
12. // It is possible for a dispenser to have
13. // one product type 'in front' and another in behind
14. class ProductInDispenser {
15.    * -- 1 Dispenser;
16.    * -- 1 ProductType;
17.    Integer numberProductsOfThisTypeLeft;
18.    }
19.
20. class ProductType {
21.    Integer pricePerUnit; // cents
22. }
```

```
23.
24. class VendingMachine {
25.   1 -- * CoinHolder;
26. }
27.
28. class CoinType {
29.   Integer value; // cents
30.   Integer weight; // grams
31.   Integer diameter; // micrometers
32. }
33.
34. class CoinHolder {
35.   * -- 1 CoinType canHold;
36.   Integer numCoinsCapacity;
37.   Integer currentNumberOfCoins;
38. }
39.
40. // Every product purchase is recorded
41. class ProductTransaction {
42.   * -- 1 ProductInDispenser;
43.   1 --
    * CoinHolderTransaction; // coins entered and change given
44. }
45.
46. // The number of coins transferred to or from a CoinHolder
47. class CoinHolderTransaction {
48.   Integer numCoinsInOrOut; // negative if used to provide chang
49.   * -- 1 CoinHolder sourceOrDestination;
50. }
51.
52. //Positions used for diagram display
53. class VendingMachine
54. {
55.   position 50 29 119 45;
56.   position.association Dispenser:slots__VendingMachine 120,23
57.   position.association CoinHolder__VendingMachine 27,45 5,0;
58. }
59.
60. class Dispenser
61. {
62.   position 290 29 149 80;
63. }
64.
65. class ProductInDispenser
66. {
```

```
  position 166 157 291 63;
  position.association ProductInDispenser__ProductType 10,63 3
  position.association Dispenser__ProductInDispenser 217,0 56,
}

class ProductType
{
  position 120 254 166 63;
}

class CoinType
{
  position 45 531 144 97;
}

class CoinHolder
{
  position 29 362 236 80;
  position.association CoinHolder__CoinType:canHold 40,80 90,0
}

class ProductTransaction
{
  position 337 277 136 45;
  position.association CoinHolderTransaction__ProductTransactio
  position.association ProductInDispenser__ProductTransaction 3
}

class CoinHolderTransaction
{
  position 283 501 201 63;
  position.association CoinHolder:sourceOrDestination__CoinHol
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

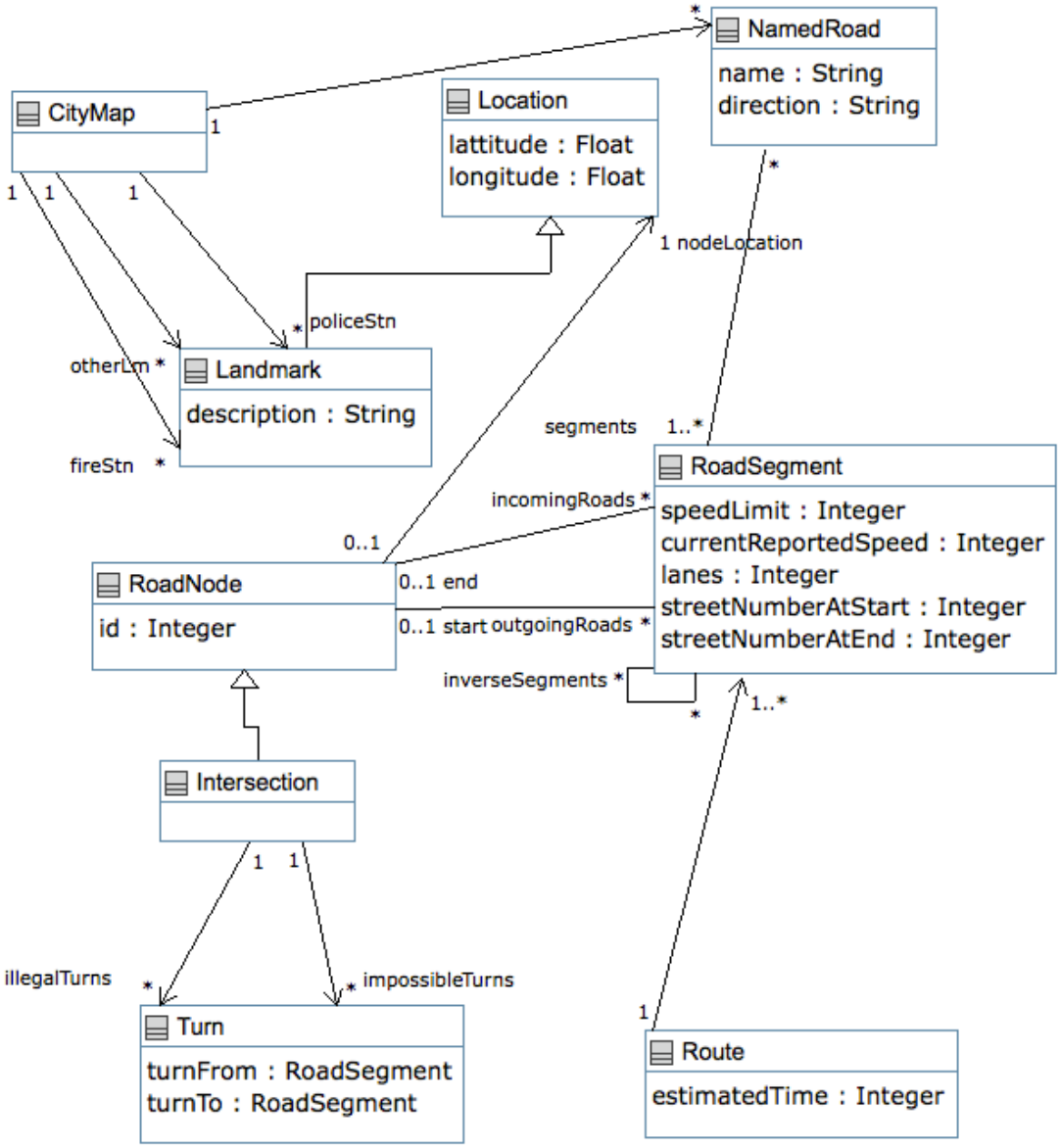
```
// UML state machine diagram of a vending machine
// There is also a class diagram example separately
```

```
03. class VendingMachine
04. {
05.   controlUnit {
06.     ReceivingMoney {
07.       enterCoin  -> DeliveringItem;
08.     }
09.
10.     DeliveringItem {
11.       deliveryComplete -> ReturningMoney;
12.     }
13.
14.     ReturningMoney {
15.       returningComplete -> Waiting;
16.     }
17.
18.     Waiting {
19.       pressSelection  -> ReceivingMoney;
20.     }
21.   }
22. }
23.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Routes And Locations

This example has been built for the [Comparing Modeling Approaches Workshop]



Example

view plain

```
01. // RoutesAndLocations.ump  Class diagram represented in Umple
02. //
03. // Author: Timothy C. Lethbridge
04.
```

```
05. namespace routesAndLocations;
06.
07. /*
08.
09. This file describes reusable model elements that are used in the
10. bCMS Crisis Management System. It is necessary to have a rudim
11. Graphical Information System such as described in this submodel
12. of bCMS in order to plan routes and to track where crises are. Thi
13. has the necessary classes.
14.
15. */
16.
17. /*
18.  * A CityMap contains the precompiled streets and landmarks
19.  * This must be read in from the database on system initialization
20.  */
21. class CityMap {
22.   singleton;
23.   0..1 -> * Landmark fireStn;
24.   0..1 -> * Landmark policeStn;
25.   0..1 -> * Landmark otherLm;
26.   0..1 -> * NamedRoad;
27. }
28.
29. /*
30.  * A Location describes a place such as an intersection, landmark,
31.  */
32. class Location {
33.   Float lattitude;
34.   Float longitude;
35. }
36.
37. class Landmark {
38.   isA Location;
39.
40.   // Name of landmark (name of fire station, business, address)
41.   description;
42.   landmarkType { fireStation { } policeStation { } touristDestinati
43. }
44.
45. /*
46.  * RoadNodes are places where RoadSegments connect. A node w
47.  * and one outgoing is used to handle changes in direction, e.g. a
48.  * a corner, changes in speed limit, changes in number of lanes, a
49.  * When there is more than one outgoing or incoming, the node r
```

```
 * where traffic flow can split and join.
 * A crossroads is one type. Entry into a roundabout would be anc
 * Highway merges and exits are other kinds as are parking lots a
 * into or out of fire stations..
 *
 * RoadSegments leaving the city lead to no RoadNode
 * RoadSegments entering the city come from no RoadNode
 */
class RoadNode {
  Integer id;
   // Could be at a landmark
  0..1 -> 1 Location nodeLocation;
  0..1 end -- * RoadSegment incomingRoads;
  0..1 start -- * RoadSegment outgoingRoads;
}

class Intersection {
  isA RoadNode;
  // illegal, but possible turns for police and fire
  0..1 -> * Turn illegalTurns;
  // Impossible turns , e.g. because of barriers, turning radius
  0..1 -> * Turn impossibleTurns;
}

/*
 * Turns are used to model illegal turns, e.g. turning left
 * when there is 'no left turn' allowed, or 'no U turn' allowed.
 */
class Turn {
  RoadSegment turnFrom;
  RoadSegment turnTo;
}

/*
 * A named road might have the name of a street, the number of
 * a highway, etc. A RoadSegment can have several names becau
 * for example, sometimes several numbered highways share a se
 */
class NamedRoad {
  name;

  // Most roads have two directions, e.g. North and South
  direction;

  // Listed in order
```

```
  * -- 1..* RoadSegment segments;
}

/*
 * A RoadSegment represents a section of road on which a vehicle
 * The ends of each segment have been modeled using RoadNode
 * Note that distance can be calculated from the speed limit, and
 */
class RoadSegment {
  Integer speedLimit;

  // Indicator of real-time congestion; 0=blocked
  Integer currentReportedSpeed;
  Integer lanes;

  // The following are used to determine addresses
  Integer streetNumberAtStart;
  Integer streetNumberAtEnd;
  * -> * RoadSegment inverseSegments;
}

/*
 * A route is a plan to get from one location to another
 * It is built by algorithms that traverse the nodes and arcs of the
 * taking into account speed, congestion, etc.
 */
class Route {
  // Ordered list of segments
  0..1 -> 1..* RoadSegment;

  // Time seconds at current flow speeds
  Integer estimatedTime;
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Coordination State Machine

This example has been built for the Comparing Modeling Approaches Workshop

Note that this file is too large for the 'Open In UmpleOnline' to work in most browsers. You will need to copy the text to the clipboard and paste it into UmpleOnline.

Example

view plain

```
01. // CoordinationStateMachine.ump
02. // This UML state machine diagram in Umple is for the
03. // bCMS car crash management system
04. // See http://cserg0.site.uottawa.ca/cma2012/CaseStudy.pdf
05. // and http://www.cs.colostate.edu/remodd/v1/content/
    repository-model-driven-development-remodd-overview
06. // Author: Timothy C. Lethbridge
07.
08. /*
09.  * State machine for co-ordination of the
10.  * crisis, as seen from one SC. If
11.  * communication is operating properly the
12.  * other coordinators' state machines will
13.  * generally be in the same states and
14.  * substates.
15.  *
16.  * Modelled given Section 4 of the requirements
17.  * This is a mixin to class crisis.
18.  */
19. class Crisis {
20.
21.   // Number of ms allowed to negotiate routes,
22.   // establish credentials
23.   Integer timeoutInMs = 20000;
24.
25.   // True if we are the initiator of the crisis
26.   // request, false otherwise
27.   Boolean thisSideInitiator;
28.   Boolean thisSideInitiatorOfTermination;
29.
30.   Boolean routeAgreedYet;
31.
```

```
32.   // True of we are the PSC, false of we are
33.   // the FSC
34.   Boolean police;
35.
36.   // Top level state machine
37.   crisisCoordinationStage {
38.
39.     // 0. When initiated the crisis is in
40.     // noCrisis state
41.     // It has been created but not yet
42.     // populated, awaiting initiation
43.     // of coordination and data..
44.     noCrisis {
45.       entry / {setThisSideInitiator(false);}
46.       // We are initiating and requesting the
47.       // other side to respond
48.       initiateCrisis / {
49.         setThisSideInitiator(true);
50.         sendInitiateCrisisRequestToOtherSC();
51.         } -> establishingCommunication;
52.
53.       // We are responding to a request from
54.       // the other SC
55.       receiveInitiateCrisisRequest ->
56.         establishingCommunication;
57.     }
58.
59.     // 1. Establish communication and
60.     // identification when a crisis begins
61.     establishingCommunication {
62.       entry / {sendSecureCredentials();}
63.       secureCredentialsConfirmed ->
64.         managingCrisis;
65.
66.       // Return to idle after some delay if
67.       // other side initiated and did not
68.       // respond. It might have been a
69.       // hacker. Alternatively if there is a
70.       // real crisis, normal manual processes
71.       // would happen.
72.       after (timeoutInMs)
73.         [!isThisSideInitiator()] -> noCrisis;
74.     }
75.
76.     // Managing crisis involves continually
```

```
77.     // exchanging information
78.     // Negotiating or renegotiating routes and
79.     // other crisis details
80.     // These are handled concurrently.
81.     managingCrisis {
82.       // A crisis is underway. We initially
83.       // start with empty data
84.       // which will be populated over time
85.       entry / {initiateEmptyCrisis(); }
86.
87.       // 2,4,5,6. Exchanging details about what
88.       // each knows about the crisis
89.       // So the information each coordinator
90.      // has is the same.
91.       exchangingDetails {
92.         // Requirement Scenario 2
93.         ourUpdateToCrisisData
94.           / {sendCrisisData();}
95.           -> exchangingDetails;
96.         receiveCrisisData
97.           / {updateCrisisData();}
98.           -> exchangingDetails;
99.
100.        // Requirement Scenario 4
101.        ourVehicleDispatched
102.          / {sendVehicleDispatch();}
103.          -> exchangingDetails;
104.        receiveVehicleDispatched
105.          / {updateTheirDispatch();}
106.          -> exchangingDetails;
107.
108.        // Requirement Scenario 5
109.        ourVehicleArrived
110.          / {sendVehicleArrived();}
111.          -> exchangingDetails;
112.        receiveVehicleArrived
113.          / {updateTheirArrival();}
114.          -> exchangingDetails;
115.
116.        // Requirement Scenario 6
117.        ourVehicleMetObjective
118.          / {sendVehicleMetObjective();}
119.          -> exchangingDetails;
120.        receiveMetObjective
121.        / {updateTheirMetObjective();}
```

```
122.         -> exchangingDetails;
123.
124.         // Requirement 5.a - breakdown
125.         breakdown / {dispatchAndUpdateOther();}
126.           -> exchangingDetails;
127.
128.         // Requirement 5.b -
129.         // congestion/blockage
130.         // TO DO
131.
132.         // Requirement 5.c escalate crisis -
133.         // renegotiate routes
134.         // TO DO
135.
136.
137.       } // End of exchangingDetails
138.         //concurrent substate
139.       ||
140.
141.       // 7. Both parties must agree to close
142.       // the crisis; either can initiate
143.       crisisEndManagement {
144.
145.         // Normal substate of crisisEndManagement -
    no end in sight yet
146.         ongoingCrisis {
147.
148.           // We could initiate termination
149.           initiateTermination / {
150.             setThisSideInitiatorOfTermination(true);
151.             sendTerminationRequestToOtherSC();
152.             } -> waitingForTerminationConfirmation;
153.           receiveTerminationRequestFromOtherSC
154.           ->
155.             waitingForUserTerminationAgreement;
156.         }
157.
158.         waitingForUserTerminationAgreement {
159.           do {confirmWithUserToTerminate();}
160.           confirmTermination -> tearDown;
161.           after (timeoutInMs) -> ongoingCrisis;
162.         }
163.
164.         // Substate of where we are waiting for
165.         // the other end to agree
```

```
166.         waitingForTerminationConfirmation {
167.           receiveTerminationConfirmation
168.             -> tearDown;
169.
170.           // If the other side has not agreed,
171.           // we keep the crisis active
172.           after (timeoutInMs) -> ongoingCrisis;
173.         }
174.
175.         // End of crisis
176.         tearDown {
177.           entry / {deleteCrisis();}
178.           -> noCrisis;
179.         }
180.       } // End of crisisEndManagement
181.         // concurrent substate
182.       ||
183.
184.       // 3. Negotiating route plan
185.       negotiatingRoutePlan {
186.
187.         // Negotiation happens in parallel with
188.         // reporting timeout
189.         negotiation {
190.           entry / {setRouteAgreedYet(false);}
191.           informOfNumberOfVehicles
192.             / {sendNumberOfVehicles();}
193.             -> negotiation;
194.           receiveNumberOfVehicles [isPolice()]
195.             -> planningRoute;
196.           receiveRouteProposal [!isPolice()]
197.             -> approvingRoute;
198.
199.           // Requirement 3.3.a2.a1
200.           receiveNoAgreeableRoute
201.             [!isPolice()] -> noRouteAgreement;
202.
203.           // The PSC plans the route -- only
204.           // PSC can be in this state
205.           planningRoute {
206.             do {planRoute();}
207.             routePlanComplete
208.               / {sendPlanToFSC(); }
209.               -> awaitingRouteApproval;
210.
```

```
211.          // Requirement 3.3.a2.a1 - no more
212.          // possible routes
213.          noMoreRoutes
214.           / {sendNoMoreRoutesToFSC(); }
215.           -> noRouteAgreement;
216.        }
217.
218.        // The FSC approves the route  --
219.        // only FSC can be in this state
220.        approvingRoute {
221.          do {userConfirmRouteAcceptable();}
222.          routeAcceptable
223.           / {sendApprovalToPSC();}
224.           -> routeAgreed;
225.          routeUnacceptable
226.           / {sendDisapprovalToFSC();}
227.           -> negotiation;
228.        }
229.
230.        // The PSC awaits for approval from
231.        // the FSC. Only the PSC can be in
232.        // this state
233.        awaitingRouteApproval {
234.          receiveApprovalFromFSC
235.           -> routeAgreed;
236.
237.          // Requirement 3.3.a - FSC
238.          // disagrees
239.          receiveDisapprovalFromFSC / {
240.            addRouteToDisapprovedChoices();
241.           } -> planningRoute;
242.        }
243.
244.        routeAgreed {
245.          entry / {setRouteAgreedYet(true);}
246.        }
247.
248.        noRouteAgreement {
249.          // requirement 3.3.a2.a1
250.        }
251.      } // End of Negotiation concurrent
252.        // substate of negotiatingRoutePlan
253.      ||
254.      managingTimeliness {
255.        timeAcceptable {
```

```
256.           // Requirement 3.a1. Negotiations
257.           // are taking excessive time
258.           after (timeoutInMs)
259.             [!isRouteAgreedYet()]
260.             -> timeUnacceptable;
261.         }
262.         timeUnacceptable {
263.           // Although negotiations may yet
264.           // complete, we are going to send
265.           // out our own vehicles because too
266.           // much time has passed
267.           do {
268.             promptAndLogReasonForTimeout(); }
269.         }
270.       } // End of managingTimeiness
271.         //concurrent substate of
272.         // negotiatingRoutePlan
273.     } // End of NegotiatingRoutePlan
274.       //concurrent substate
275.   } // End of Managing Crisis top level state
276.  } // End of crisisCoordinationStage state
277.    // machine
278. } // End of state machine mixin for
279.  // Crisis
280.
281.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Important tips

The tips below will appear as the tips of the day in UmpleOnline. You can read them all here.

You can specify an association in a class or else by itself with the *association* keyword.

An association between two classes can be specified in either class (see an example in the user manual) or separately from either class, using the *association* keyword

You can generate code that ensures a class is a *singleton* (has only one instance).

If you specify *singleton;* in any class, then the singleton pattern will be applied, meaning that there will only be one instance.

Mark an attribute as *lazy* if you do not want it to create a constructor argument.

By default each attribute in a class becomes a constructor argument (along with associations that have a multiplicity of 1 at their other end).

If you want to avoid this, then add the keyword *lazy* before the association. The attribute will then be initialized with a null, empty or zero value.

The default data type for attributes in Umple is *String*.

In Umple you can specify a data type before any attribute. But if you do not, then attribute will default to type *String*.

Core datatypes in Umple include String, Integer, Boolean, Date, Time.

If you use the datatypes String, Double, Float, Integer, Boolean, Date or Time as the types of attributes, then Umple generates code tailored to the target language.

For example, when generating Java, an Integer attribute is implemented as an int variable. This is important to know, since the methods Umple generates to access those variables will use the Java type. If you specify `Integer id;` in Umple, and generate Java, then the API will have a method with signature *`setId(int)`*

Why does Umple then bother with an Integer datatype if it will be converted to int when generating Java? Because we want key types to be independent of the target programming language.

With UmpleOnline, you can create *tasks* to assign to others.

Under Tasks, you can create numerous assignments with different names, instructions, Umple models, and completion URLs (such as feedback quizzes) with the option of detailed logging. Simply create a task and distribute the participant URL or ask the responders to look up the name of the task under "Load Task" in order to start their submissions.

To acquire all the submissions, click "Request all Directories as a zip" under the task editing panel and all of the submissions will be saved on your local computer.

Secondary tips

In the Umple User Manual, you can select *Combined version* at the bottom.

At the bottom-left of each user manual page is a link called Combined Version. If you select this you can then use the built-in search function in your browser (control-f or command-f) to find information. Beware though that the combined version web page is extremely big. There is also a pdf version of the manual available.

In Umple textual code, the type comes *before* an attribute name, but in UML it comes *afterwards*.

Umple follows the Java, C and C++ approach of specifying the type of an attribute: The type is specified followed by the name. For example:

```
Integer count;
```

But class diagrams created in Umple follow the *UML* convention whereby the type follows the name of the attribute, with a colon separating them. So the above example in a diagram appears as:

```
count: Integer
```

Tertiary tips

You can work on *multiple model tabs* in UmpleOnline.

In UmpleOnline you are able to write code in multiple tabs. The tabs appear above the text-editing area in the top-left. To add a new tab, click on the + symbol.

You can rename a tab by double-clicking on its name, and typing the new name.

Consider the tabs to be separate files. You can arrange for the model in a tab to *know about* the model in another tab you have named *Utilities* by including the expression `use utilities.ump`

You can select *Hide Tabs* to hide model tabs if you don't want to use them, and *Show Tabs* to show them again.

Note that tabs can also be shown in the generated code area; these are entirely separate

You can change the width of the text and diagram panels in UmpleOnline.

If you hover the cursor over the right edge of the text panel (just to the left of the menu) or the left edge of the diagram panel (just to the right of the menu), the cursor will change. When the cursor has changed you can press the mouse/trackpad and drag left or right to change the relative portion of the screen devoted to either panel.

Feel free to work on either the text or the diagram panel according to your preferences.

In UmpleOnline, by toggling the T and D buttons above the text editor, you can choose to hide and show the text panel (T) or the diagram panel (D).

Hiding the diagram panel temporarily can sometimes speed up your work, although you will receive less feedback about your model. When the diagram is visible it updates automatically 3 seconds after you stop typing new code in the model panel.

Hiding the text panel temporarily can help you display a large diagram on the screen.

Hovering over an item in UmpleOnline will allow you to see more info about it.

By hovering over a button or some text in UmpleOnline, a message will be displayed with extra information about that item.

Privacy and Risks

Risks in using Umple and UmpleOnline

License: All use of Umple and UmpleOnline is subject to the Umple MIT license. Please read it carefully, since it disclaims liability. This is not because we don't want to be 'good engineers' and take responsibility for work, rather it is because we are following the open source model, which allows a wide variety of people to modify Umple.

Risk due to support by hosting organizations: UmpleOnline is hosted by the Digital Research Alliance of Canada, and maintained by the University of Ottawa with funds from research grants and donations. Should these supports cease and donations not suffice, then support for Umple may cease unless others take over responsibility. Similarly, the code is hosted on Github. If Github decides to cease hosting projects for free, the Umple code would need to be hosted somewhere else.

Risk of deprecation, missing features and defects: It is possible, although unlikely, that Umple code which works today may cease working in the future. Development is performed in the context of research. You will therefore find incomplete features, which may have bugs. We encourage you to report new bugs (and fix them) and to realize that you may need to work around the existing ones if you use experimental features. That said, we do use test-driven development to maintain what we believe is a high level of quality for the core features.

Limitations of UmpleOnline: The purpose of UmpleOnline is to allow people to explore Umple and model-oriented programming, particularly in an educational context. UmpleOnline is not intended to be a full-fledged tool for either commercial or open source; this is one of the reasons why it is only capable of working with a small number of Umple files per user session. If you want to do serious development in Umple, with many files, you should download it and run it on the command line, or some other supported IDE.

Not certified for safety critical or mission critical use: At the current time Umple-generated code **should not be used for mission-critical or safety critical uses**, including software for any device that may pose a safety risk if it performs incorrectly, or software that would cause economic damage if it failed. We intend that, in time, Umple and tools like it will in fact help improve safety and reliability. But at the current time we have not subjected Umple to the rigorous validation it needs for such uses, and there are known issues that would preclude such current use.

Need to apply best practices: Should you choose to use Umple for production use, it is **critical that you follow rigorous software engineering practices** including (but not limited to): Requirements analysis, careful design, code inspection, and thorough automated testing. See here for a list of Umple best practices.

Privacy: Use of Cookies

UmpleOnline stores a copy of your most recent edited Umple code and various settings in cookies. This protects against losing data by accidental closure of the browser. Upon starting UmpleOnline again, the user will be presented with an option to 'Restore saved state' by loading the model and settings from such cookies. This does mean that someone else might be able to find out what you were editing if they had access to your computer. You should not, therefore, use UmpleOnline if you are concerned about such access.

Privacy: Saving of data in UmpleOnline

Models entered in UmpleOnline are automatically stored on servers managed by the Digital Research Alliance of Canada. Each time you type and pause for three seconds, each time you make an edit to a diagram, and each time you generate code, your data is saved. The data includes one or more .ump files, plus the data you have generated from those files (Java code, Python code, etc.).

Data saved automatically in this way remains stored in the cloud by the Digital Research Alliance of Canada (backed up periodically to the University of Ottawa) for two days or slightly more. This is so you can continue an editing session, even if you walk away from your computer for an extended period. We have an automated process that will normally delete such data after a few days. However we reserve the right to record general statistics about the size of models and other uses of Umple tools before we delete such data.

While a model is stored, UmpleOnline also keeps earlier versions of its .ump files alongside it, so that you can go back to them using 'Restore Earlier Version'. These earlier versions are deleted along with the model.

If you choose 'Save as URL' then your model is stored for an extended period, subject to deletion rules described below. Your files can be edited and deleted by anyone to whom you give the URL, or by anyone who guesses the URL (there is less than a 1 in a trillion chance of this though).

If we detect abuse of UmpleOnline, we reserve the right to attempt to

track the user using such tools as the originating IP address, and to block access from such an address or address range.

We will occasionally use external tools such as SurveyMonkey to conduct surveys of users, links to which may appear from time to time on the screen. People who complete such surveys would then be subject to the privacy rules of such external tools. Users are requested to give informed consent prior to taking such a survey, and such informed consent is first approved by the University of Ottawa's Research Ethics Board.

Sharing of personal data in UmpleOnline

The only data saved by UmpleOnline is the model you create, either graphically or textually or both. There is currently no login mechanism so there is no userid, name or other personal data associated with your model. We may impose a login requirement in the future, but in that case we would only store the minimum of data (loginID, your name, an encrypted salted password, and an email address for account confirmation, to contact you and to allow for password reset).

You may, however, embed (at your choice) confidential information in the code or models you write in UmpleOnline. It is important for you to realize that this information is accessible to the site maintainers and to anyone to whom you give the URL of your data.

Since no userid is currently associated with UmpleOnline models, we have no way of determining who has saved which models at the current time. We cannot guarantee to be able to recover any file you may have 'lost'. Nor can we determine whether anyone else has looked at or modified your files.

Data deletion

Models and associated generated outputs are always deleted after a few days unless explicitly saved as a permanent URL. If the user generates a permanent URL our normal policy is to keep the data for two years after the last time it has been edited. But this is not guaranteed, for the reasons mentioned below.

You may delete your own model in UmpleOnline: Simply select all the text and delete it. Note that earlier versions of the model can still be restored until the model itself is deleted. To delete all records of generated code, it is suggested that you replace your model by a single line of code (such as class X {} and then generate code from it (generate code in all formats you have previously generated).

Staff at the University of Ottawa reserve the right to delete models for any of the following reasons:

- There is objectionable content, including but not limited to, code for anything illegal. We may occasionally scan for such content using manual and automatic means.
- Using the site to to store something other than Umple code (e.g. using it to store other forms of data in the form of code comments)
- Our servers become full or over-taxed. In this case we will make an effort to delete large models that are also old, before removing recently updated and smaller models. Ultimately, we cannot guarantee permanent storage of any model; we just intend to maintain models for as long as we can.
- Failure of the system in any way.

Access and use by others

If somebody is able to guess the URL of your model, or you give it to them, then they can modify and delete your model. Important models should therefore be saved using other means. Instructions for how to do that are here.

Anonymity

There is currently no login mechanism to UmpleOnline so there is no way to trace users, as stated earlier. This may change in the future.

Umple Grammar

Below is the complete grammar for Umple . This page is generated automatically from the grammar used to parse Umple code.

- Rules are in blue. You may click on a usage of a rule (underlined and in [[ ]]) anywhere in this manual in order to see the definition.
- Terminal symbols are in red. These keywords and symbols that are key to Umple's syntax.
- Identifiers are in green and are surrounded by [ ]. These are places where you can supply a sequence of alphanumeric characters to name a class, attribute or some other entity.
- Arbitrary input is in yellowish green and surrounded by [**]. Umple ignores everything until it finds the symbol that terminates the input. This is used for comments and to embed code in a language like Java or PhP. Umple will pass this text to the parser of the underlying language.
- Comments about the grammar itself are in brown and preceded by //
- symbols controlling how the grammar is parsed are in black.
  - * means zero or more
  - | means 'or'
  - ? means optional
  - Other black items are of interest only to experts

Refer to the Grammar Notation section for more explanation of the EBNF syntax used here.

```
// The master source of this first part of the Umple grammar
is available at
// https://github.com/umple/umple/blob/master/cruise.umple/src/
umple_core.grammar
// The html rendering is generated from the master and appears
in the Umple User manual
// at page https://umple.org/grammar

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source
license found at:
// https://umple.org/license

// The core of umple is a "program". This is the grammar's
'start symbol'
// Comments and lone semicolons are ignored
```

```
program- : [[useProgram]] | [[umpleProgram]]


umpleProgram- : ( [[comment]] | [[directive]] | ; )*


// Directives are the top-level items in an umple file. See
manual page TypesofDirectives
// A directive is either used to configure the system or else
is
// an actual entity of the system to be modelled or generated
directive- : [[checkForUnintendedBracket]]
    | [[glossary]]
    | [[generate]]
    | [[suboption]]
    | [[distribute]]
    | [[generate_path]]
    | [[filter]]
    | [[useStatement]]
    | [[namespace]]
    | [[requirement]]
    | [[reqImplementation]]
    | [[tracerDirective]]
    | [[entity]]
    | [[debug]]
    | [[strictness]]
    | [[toplevelExtracode]]
    | [[toplevelException]]


// A glossary item is used to fine tune pluralization during
code generation
glossary : glossary { [[word]]* }

word : [singular] : [plural] ;

distribute : distributable [=distributeTech:RMI
    |WS]? [=distributePattern:0
    |1
    |2
    |3]? [=distributeVal:on
    |off
    |forced] ;
```

```
// A high level of strictness will cause warnings to be issued
when base language code
// is found, where it might not have been intended. Strictness
can also be used
// To either suppress certain messages, or to declare that
they should be present.
// NOTE: This is currently under development
strictness- : strictness
  ( [=strictness:modelOnly|noExtraCode|none]
    | [[strictnessMessage]]
    | [[strictnessDisableAuto]]
  ) ;

strictnessMessage : [=message:allow|ignore|expect|disallow]
[messageNumber]

strictnessDisableAuto : disable [**expression]


// The generate clause can be used to generate multiple
outputs
// The --override is used to say that subsequent generate
statements will be ignored
generate : generate [=language:Java
    |Nothing
    |Php
    |RTCpp
    |SimpleCpp
    |Ruby
    |Python
    |Cpp
    |Json
    |FeatureModelJson
    |StructureDiagram
    |Yuml
    |Violet
    |Umlet
    |Simulate
    |TextUml
    |Scxml
    |GvStateDiagram
    |GvClassDiagram
    |GvFeatureDiagram
    |GvClassTraitDiagram
```

```
    |GvEntityRelationshipDiagram
    |Alloy
    |NuSMV
    |NuSMVOptimizer
    |Papyrus
    |Ecore
    |Xmi
    |Xtext
    |Sql
    |StateTables
    |EventSequence
    |InstanceDiagram
    |Umple
    |UmpleSelf
    |USE
    |Test
    |SimpleMetrics
    |PlainRequirementsDoc
    |Uigu2
    |ExternalGrammar
    |Mermaid]
  ( [=suboptionIndicator:-s
    |--suboption] " [**suboption] "
  )* ;


generate_path : generate [=language:Java
    |Nothing
    |Php
    |RTCpp
    |SimpleCpp
    |Ruby
    |Python
    |Cpp
    |Json
    |FeatureModelJson
    |StructureDiagram
    |Yuml
    |Violet
    |Umlet
    |Simulate
    |TextUml
    |Scxml
    |GvStateDiagram
```

```
    |GvClassDiagram
    |GvFeatureDiagram
    |GvClassTraitDiagram
    |GvEntityRelationshipDiagram
    |Alloy
    |NuSMV
    |NuSMVOptimizer
    |Papyrus
    |Ecore
    |Xmi
    |Xtext
    |Sql
    |StateTables
    |EventSequence
    |InstanceDiagram
    |Umple
    |UmpleSelf
    |USE
    |Test
    |SimpleMetrics
    |PlainRequirementsDoc
    |Uigu2
    |ExternalGrammar
    |Mermaid] " [**output] " [=override:--override
    |--override-all]?
  ( [=suboptionIndicator:-s
    |--suboption] " [**suboption] "
  )* ;


suboption : suboption "[**value] " ;


// Use statements allow incorporation of other Umple files.
See UseStatements
useStatement : use [use] ( , [extraUse] )*


toplevelExtracode : top [top] [[moreCode]]


// Namespaces divide the code into logical groups. See
NamespaceDirectives
namespace- : namespace [namespace] [[namespaceoption]]? ;
```

```
namespaceoption : [=option:--redefine]


// The main top level elements to be found in an Umple file
entity- : [[mixsetIsFeature]]
     | [[requireStatement]]
     | [[mixsetDefinition]]
     | [[classDefinition]]
     | [[traitDefinition]]
     | [[fixml]]
     | [[interfaceDefinition]]
     | [[externalDefinition]]
     | [[associationDefinition]]
     | [[associationClassDefinition]]
     | [[stateMachineDefinition]]
     | [[templateDefinition]]
     | [[enumerationDefinition]]
     | [[toplevelCodeInjection]]


// Comments follow the same conventions as C-family languages.
UmpleComments
comment- : [[inlineComment]] | [[multilineComment]] |
[[annotationComment]]

inlineComment- : // [**inlineComment]

multilineComment- : /* [**multilineComment] */


reqIdentifier : [name]


requirement : req [reqIdentifier]
  (
  ( [[userStory]]
  )
    |
  ( [[useCase]]
  )
    |
  ( [[ears]]
  )
```

```
    |
  ( [[quality]]
  )
    |
  ( [[plainReq]]
  )
  )

quality- : [=reqLanguage:quality|Quality] { [**reqContent] }

plainReq- : [reqLanguage]? { [**reqContent] }

userStory- : [=reqLanguage:userStory|userstory] {
  ( [[userStoryTags]] | [**reqContent]
  ) }

userStoryTags- :
  ( who { [**whoContent] }
  )?
  ( when { [**whenContent] }
  )?
  ( what { [**whatContent] }
  )?
  ( why { [**whyContent] }
  )?

useCase- : [=reqLanguage:useCase|usecase] {
  ( [[useCaseTags]] | [**reqContent]
  ) }

useCaseTags- :
  ( [[userStoryTags]] | [[useCaseRelationship]] | [[useCaseStep]]
  )*

// A use case can incorporate the steps of other use cases, or
specialize another use case
useCaseRelationship- : [=useCaseRelationType:include|isA]
[useCaseRef]
  ( , [useCaseRef]
  )* ;

useCaseStep- :
  ( userStep [useCaseId] [[useCaseStepCondition]]? {
[**useCaseStepContent] }
```

```
  )
    |
  ( systemResponse [useCaseId] [[useCaseStepCondition]]? {
[**useCaseStepContent] }
  )

// A step is taken only when its condition holds; written like
a state machine guard
useCaseStepCondition- : [ [**useCaseCondition] ]

// An EARS (Easy Approach to Requirements Syntax) requirement.
See Requirements. The body is free text, or
// any of the clauses where { feature } while { precondition }
when { trigger } if { unwanted trigger } system { name }
// and shall { response }, which are read in the EARS order:
Where ..., while ..., when ..., if ..., then the ... shall ...
ears- : [=reqLanguage:ears] { [**reqContent] }

reqRef : [reqIdentifier]
  (( [qualityClassName] )
  )?

reqImplementation : implementsReq [[reqRef]] (, [[reqRef]])* ;


//Matches all forms of Java annotations including those with
parentheses and values in quotes
annotationComment- : [!annotationComment:@[a-zA-Z_][a-zA-
Z0-9_-]*(\(((\"(\w
    |\s
    |CLOSE_ROUND_BRACKET*")
    |\w
    |\s
    |=
    |,
    |\.)*\))?]


// The Debug statement turns on debugging in code generation.
For developer use only
debug- : [=debug] ;


// Instructs a class or method to be abstract
```

```
abstract- : [=abstract] ;

// The master of this second part of the Umple grammar is
available at
// https://github.com/umple/umple/blob/master/cruise.umple/src/
umple_patterns.grammar

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source
license found at:
// https://umple.org/license

softwarePattern- : [[isA]] | [[singleton]] | [[immutable]] |
[[keyDefinition]] | [[codeInjection]]


// Generalization and inheritance isAclause
isA- : [[singleIsA]] | [[multipleIsA]]

singleIsA- : isA [[isAName]] ( , isA [[isAName]] )* ;

multipleIsA- : isA [[isAName]] ( , [[isAName]] )* ;

isAName- : " [**extendsNames] " [[gTemplateParameter]]?
    | [extendsName] [[gTemplateParameter]]?


gTemplateParameter : < [[AllInclusionExclusionAlias]] >


// A class that can have only one instance SingletonPattern
singleton- : [=singleton] ;


// A class that can't be modified when created ImmutablePattern
immutable- : [=immutable] ;


// For equality and hashing KeysforEqualityandHashing
keyDefinition- : [[defaultKey]] | [[key]]


// Aspect oriented code injection: BeforeandAfterStatements
codeInjection- : [[beforeCode]] | [[afterCode]]
```

```
toplevelCodeInjection- : [[toplevelBeforeCode]] |
[[toplevelAfterCode]]


parameterTypes : [parameterType] (, [parameterType])*

parameterListing : (
  ([[parameterTypes]]
  )? )


injectionOperation : [operationName] ([[parameterListing]])?


beforeCode : ( before | [=around] ) [[aspectBody]]

afterCode : after [[aspectBody]]


toplevelBeforeCode : ( before | [=around] ) {([className])(,
[className])*} [[aspectBody]]

toplevelAfterCode : after {
  ([className]
  )
  (, [className]
  )*} [[aspectBody]]


aspectBody- :
  ([=operationSource:custom
    |generated
    |all]
  )?
  ([!codeLabel:\S+:]
  )? [[injectionOperation]]
  ( , [[injectionOperation]]
  )*
  ([[codeLang]] [[codeLangs]]
  )? { [**code] }
  ( [[moreCode]]
  )*
```

```
// The master of this part of the Umple grammar is available
at
// https://github.com/umple/umple/blob/master/cruise.umple/src/
constraint/umple_constraints.grammar

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source
license found at:
// https://umple.org/license

// Constraints in Umple.
// This is currently under development. Constraint capability
is being
// developed in agile increments. The first step, described
below,
// allows limiting the values of attributes. Code generation
is not currently enabled.
// Constraints may appear in classes (including association
classes)
// as well as in states.
precondition : [ [name]? pre : [[constraint]] ]

postcondition : [ [name]? post : [[constraint]] ]


invariant : [ ([name] :)? ([[constraint]]) ]

constraintToken : [[constraint]]


// A constraint is an expression optionally be surrounded in
round brackets or negated
constraint- : [[constraintBody]] [[linkingOp]]? | [[constraintExpr]]
[[linkingOp]]?

//negativeConstraint : ( ! | not ) ( [[constraintName]] |
[[constraint]] ) fixed issue1536 'Umple does not parse
properly the guards'
negativeConstraint :
  ( ! | [!constraintNegSymbol:not\s+]
  )
  ( [[constraintName]] | [[constraint]]
  )
```

```
// A constraint body is a constraint expression (possibly with
a linking operator such as && or ||).
constraintBody : ( [[constraint]] )

linkingOp- :
  ( [=andOp:and|&&|&]
    | [[equalityOperator]]
    | [!orOp:(or|\Q||\E)]
  ) [[linkingOpBody]]

linkingOpBody : [[constraint]]

constraintExpr- : [[statemachineExpr]]
    | [[stringExpr]]
    | [[boolExpr]]
    | [[numExpr]]
    | [[associationExpr]]
    | [[genExpr]]
    | [[loneBoolean]]
    | { [!fakeContraint:[^\n]+] }


loneBoolean : [[constraintVariable]]


//must be a boolean //fixing is here!!!
boolExpr : [[constraintVariable]] [[equalityOperator]] [[boolLiteral]]
    | [[boolLiteral]] [[equalityOperator]] [[constraintVariable]]
    | [[boolLiteral]]

boolLiteral : [=literal:true|false]


//must be string
stringExpr : [[stringExprOperator]] | [[stringExprPlain]]

stringExprPlain- : [[constraintVariable]] [[equalityOperator]]
[[stringLiteral]]
    | [[stringLiteral]] [[equalityOperator]] [[constraintVariable]]

stringExprOperator- : [[stringComplexExpression]]
[[equalityOperator]] [[stringComplexExpression]]
```

```
    | [[stringComplexExpression]] [[equalityOperator]]
[[stringName]]
    | [[stringName]] [[equalityOperator]]
[[stringComplexExpression]]

stringLiteral- : " [**quote] " | ' [**quote] '

stringName- : [[stringLiteral]] | [[constraintVariable]]

stringComplexExpression : [[stringPlusOperator]]

stringPlusOperator- : [[stringName]]
  ( [=concat:+] [[stringPlusOperator]]
  )?


//basically the "other" catagory, contains everything that can
be equal to something else
genExpr : [[constraintVariable]] [[equalityOperator]]
[[constraintVariable]]


//for floats, doubles and ints
numExpr : [[numberExpression1]]
    | [[numberExpression2]]
    | [[numberExpression3]]
    | [[numberExpression4]]

numberExpression1- :
  ( [[arithmeticCall]]
    | [[numberName]]
  ) [[ordinalOp]]
  ( [[arithmeticCall]]
    | [[numberName]]
  )

numberExpression2- :
  ( [[arithmeticCall]]
    | [[constraintVariable]]
  ) [[equalityOperator]]
  ( [[arithmeticCall]]
    | [[numberLiteral]]
  )
```

```
numberExpression3- :
  ( [[arithmeticCall]]
    | [[numberLiteral]]
  ) [[equalityOperator]]
  ( [[arithmeticCall]]
    | [[constraintVariable]]
  )

numberExpression4- :
  ( [[arithmeticCall]]
    | [[numberLiteral]]
  ) [[equalityOperator]]
  ( [[arithmeticCall]]
    | [[numberLiteral]]
  )

numberLiteral- : [!number:-?[0-9]+([\\.][0-9]+)?]

numberName- : [[numberLiteral]] | [[constraintVariable]]

arithmeticCall : (
  ( [[lowArithmeticOperatorCall]]
    | [[highArithmeticOperatorCall]]
  ) )

lowArithmeticOperatorCall- :
  ( [[highArithmeticOperatorCall]]
    | [[arithmeticResolve]]
  )? [=operator:+|-|>>|<<]
  ( [[arithmeticCall]]
    | [[arithmeticResolve]]
  )

highArithmeticOperatorCall- : [[arithmeticResolve]]
[=operator:*|/|^|%]
  ( [[highArithmeticOperatorCall]]
    | [[arithmeticResolve]]
  )

arithmeticResolve- : ( [[arithmeticCall]] ) | [[numberName]]


equalityOperator- : [=equalsOp:==|equals] | [=notequalsOp:!=|/
```

```
=|=!|=/=]

ordinalOp- : [=greaterOp:greater|>=|=>|=>=]
    | [=lessOp:less|<=|=<|=<=]
    | [=moreOp:larger|>]
    | [=smallerOp:smaller|<]


associationExpr : [[constraintVariable]] [[associationOperators]]
[[associationLiteral]]

associationOperators- : [=firstOp:cardinality|has]
  ( [[ordinalOp]] | [[equalityOperator]] | [=all]
  )?

associationLiteral : [[constraintParameter]]
  ( , [[associationLiteral]]
  )?


statemachineExpr : [[constraintVariable]] [[statemachineOperators]]
[[statemachineLiteral]]

statemachineOperators- : [=isInOp:is|state] [=ignore:==|in]
    | [=isNotInOp:is|state]
  ( [=ignore:!=]
    | [=ignore:not] [=ignore:in]
  )

statemachineLiteral- : [=state]? [~stateName]


constraintVariable- : [[negativeConstraint]] | [[constraintName]]

constraintName : [=new]? [!name:[a-zA-Z_][a-zA-Z0-9_-]*]
[[constraintIndexOperator]]* [[constraintParameterList]]?
[[constraintScopeOperator]]?

constraintIndexOperator- : [ [!index:[0-9]+] ]

constraintScopeOperator : ( . | -> ) [[constraintVariable]]

constraintParameterList : (
  ([[constraintParameter]]
```

```
  ([=comma:,] [[constraintParameter]]
  )*
  )? )

constraintParameter : [=boolLit:true|false]
    | [[stringLiteral]]
    | [[constraintVariable]]
    | [[numberLiteral]]


modelConstraint- : [ model : [[modelConstraintBody]] ]

modelConstraintBody : ( [[modelConstraintBody]] )
[[modelLinkingOp]]?
    | [[modelExpr]] [[modelLinkingOp]]?

modelLinkingOp :
  ( [=and:&|&&|and|,] | [!or:
  ([|][|]?|or|;
  )]
  ) [[modelConstraintBody]]

modelExpr : [~source]? [[modelRelationOp]] [~target]

modelRelationOp- : [[modelRelationOpAssociation]]
    | [[modelRelationOpAttribute]]
    | [[modelRelationOpInheritance]]

modelRelationOpInheritance : [=subclass:isA]
    | [=subclass:subclass|child|>>]
  ( of
  )?
    | [=subclass:inherits]
  ( from
  )?
    | [=superclass:parent|superclass|<<]
  ( of
  )?

modelRelationOpAttribute : [=op:has]
  ( attribute | attr
  )? [=classification:named|of]?

modelRelationOpAssociation : [[modelRelationAssociationEnd]]?
```

```
[=op:--
    |->
    |<-
    |-<@>
    |<@>-] [[modelRelationAssociationEnd]]?

modelRelationAssociationEnd : [!bound:(\d+|[**])] ( .. [!
bound:(\d+|[**])] )?



// NOTE: Additional grammar parts deleted while testing is
ongoing.
// The master of this second part of the Umple grammar is
available at
// https://github.com/umple/umple/blob/master/cruise.umple/src/
umple_exceptions.grammar

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source
license found at:
// https://umple.org/license

misnamedAttribute : [name] ; | [type] [name] ;

malformedConstraint : [ [!stuff:[^\]]*] ]

malformedStatement1 : [!stuff:[^\\{\\};]*] ( [!stuff:[^\\{\
\};]*] )* ;

malformedStatemachine1 : [!stuff:[^\\{\\}\\(\\) ]*] {
[**innerstuff] }

malformedStatement2 : [!stuff:[^\\{\\};]*]
  ( [!stuff:[^\\{\\};]*]
  )* { [**innerstuff] } ;

malformedStatemachine2 : [!stuff:[^\\{\\}\\(\\)]*]
  ( [!stuff:[^\\{\\}\\(\\)]*]
  )* { [**innerstuff] }

malformedMethod : [!stuff:[^\\{\\}\\(\\)]*]
  ( [!stuff:[^\\{\\}]*]
  )* { [**innerstuff] }
```

```
exception : [[misnamedAttribute]]
    | [[malformedStatement1]]
    | [[malformedStatemachine1]]
    | [[malformedStatement2]]
    | [[malformedStatemachine2]]
    | [[malformedConstraint]]
    | [[malformedMethod]]


toplevelException : [[toplevelExceptionMain]]
  ( { [**extraStuff] }
  )?
  ( " [**quotedStuff] "
  )?
  ( ;
  )?

toplevelExceptionMain : [=access:public
    |private
    |protected]? [identifier] [=queued]? [name]?
[[inheritanceException]]? [name]*

inheritanceException :
  ( [=level:extends|implements]? [name]
  ( , [name]
  )*
  )+

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source
license found at:
// https://umple.org/license

// beginning of use.grammar

// This is essentially rewriting the grammar rules present in
the USE compiler, which exist in a similar format.

useProgram : model [id]
  ([[useGeneralClassDefinition]]
    | [[useInlineComment]]
    | [[useAssociation]]
  )*
```

```
useGeneralClassDefinition- : [[useClassDefinition]]

useClassDefinition : (abstract)? class [id] [[useAttributes]]?
end

useAttributes : attributes [[useAttributeDefinition]]*

useAttributeDefinition : [id] : [[useType]] (;)?

useType- : [[useSimpleType]]

useSimpleType : [id]

useInlineComment- : (-- [**inlineComment]) | (//
[**inlineComment])

useAssociation : association [name] between
[[useAssociationEnd]] [[useAssociationEnd]] end

useAssociationEnd : [name] [ [[useMultiplicity]] ]

useMultiplicity- : [[useMultiplicityRange]]

useMultiplicityRange : [[useMultiplicitySpec]]
  (.. [[useMultiplicitySpec]]
  )?

useMultiplicitySpec : [integerSpec:\d+|[**]]

// The master of this second part of the Umple grammar is
available at
```

```
// https://github.com/umple/umple/blob/master/cruise.umple/src/class/umple_classes.grammar

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source license found at:
// https://umple.org/license

// Classes are the most common elements in Umple.
// See user manual page ClassDefinition
classDefinition : class [name] { [[classContent]]* }


//The external keyword declares a class that is defined in a different
// compilation unit
externalDefinition : external [=interface]? [name] { [[classContent]]* }


// An Interface can only contain method. See interfaceDefinition
interfaceDefinition : interface [name] { [[depend]]* [[interfaceBody]] }


// Associations can be declared outside the body of classes.
// See user manual page IndependentlyDefinedAssociations
associationDefinition : association [name]? {
  ([[comment]]
    | [[reqImplementation]]
    | [[mixsetDefinition]]
    | [[association]]
  )* }


// Associations that would be many-many can also become full-fledged classes too
// See user manual page AssociationClassDefinition
associationClassDefinition : associationClass [name] { [[associationClassContent]]* }


// Enumerations can be declared outside the body of classes
// See user manual page EnumerationDefinition
```

```
enumerationDefinition : enum [name] { [enumValue](,
[enumValue])* }


// The following items can be found inside the body of classes
or association classes
classContent- : [[comment]]
     | [[reqImplementation]]
     | [[innerClass]]
     | [[mixsetDefinition]]
     | [[distributable]]
     | [[proxyPattern]]
     | [[strictness]]
     | [[classDefinition]]
     | [[trace]]
     | [[emitMethod]]
     | [[templateAttributeDefinition]]
     | [[primitiveDefinition]]
     | [[portDefinition]]
     | [[portBindingDefinition]]
     | [[position]]
     | [[displayColor]]
     | [[abstract]]
     | [[keyDefinition]]
     | [[softwarePattern]]
     | [[depend]]
     | [[symmetricReflexiveAssociation]]
     | [[attribute]]
     | [[testCase]]
     | [[genericTestCase]]
     | [[testSequence]]
     | [[testClassInit]]
     | [[stateMachine]]
     | [[activeMethodDefinition]]
     | [[inlineAssociation]]
     | [[concreteMethodDeclaration]]
     | [[constantDeclaration]]
     | [[modelConstraint]]
     | [[invariant]]
     | ;
     | [[enumerationDefinition]]
     | [[exception]]
     | [[extraCode]]
```

```
associationClassContent- : [[comment]]
    | [[reqImplementation]]
    | [[classDefinition]]
    | [[position]]
    | [[displayColor]]
    | [[invariant]]
    | [[softwarePattern]]
    | [[depend]]
    | [[association]]
    | [[inlineAssociation]]
    | [[singleAssociationEnd]]
    | [[attribute]]
    | [[stateMachine]]
    | [[enumerationDefinition]]
    | ;
    | [[extraCode]]

innerClass : [[innerStaticClass]] | [[innerNonStaticClass]]

innerStaticClass : static [[classDefinition]]

innerNonStaticClass : inner [[classDefinition]]


checkForUnintendedBracket : (
    | )
    | OPEN_CURLY_BRACKET
    | CLOSE_CURLY_BRACKET


// Interfaces: Note that if the format of an
abstractMethodDeclaration is not
// followed, then the body will extraCode and passed to the
base language
// See user manual page interfaceDefinition
interfaceBody- : [[interfaceMemberDeclaration]]*

interfaceMemberDeclaration : [[comment]]
    | [[reqImplementation]]
    | [[constantDeclaration]]
    | [[constantDeclarationDeprecated]]
    | [[abstractMethodDeclaration]]
```

```
    | [[position]]
    | [[displayColor]]
    | [[isA]]
    | [[interfaceTest]]
    | [[distributableInterface]]
    | [[exception]]
    | [[extraCode]]

distributableInterface : [[distributable]]

// Constants in interfaces (e.g. const String
ACONSTANT="aValue";)
// Note: in Classes const is a modifier
constantDeclarationDeprecated : constant [[typedName]] (=
[**value])? ;

constantDeclaration : [=internal]? const [[typedName]] (=
[**value])? ;



distributable- : [=distributable:distributable]
[=distributeTech:RMI|WS]? ;

proxyPattern- : [=proxyPattern:proxyPattern] ;

moreCode- : [[codeLangs]] { [**code] }

codeLangs- :
  ([=codeLang:Java
    |RTCpp
    |SimpleCpp
    |Cpp
    |Php
    |Ruby
    |Python
    |Alloy
    |UmpleSelf]
  ( , [=codeLang:Java
    |Alloy
    |RTCpp
    |SimpleCpp
    |Cpp
    |Php
```

```
    |Ruby
    |Python
    |UmpleSelf]
  )*
  )?


// Methods: The code in concrete methods is passed to the base
language
// See user manual page MethodDefinition
concreteMethodDeclaration : [=modifier:public|protected|
private]? [=static]? [=synchronized]? [=queued]? [type]?
[[methodDeclarator]] [[methodThrowsExceptions]]? [[methodBody]]
    | [=modifier:public|protected]? [=abstract] [type]?
[[methodDeclarator]] [[methodThrowsExceptions]]? ;

abstractMethodDeclaration : [type] [[methodDeclarator]] ;

//queuedMethodDeclaration- : queued [[methodDeclarator]]
[[methodThrowsExceptions]]? [[methodBody]]
methodBody- :
  ( [[codeLangs]] {
  ( [[precondition]]
    | [[postcondition]]
    | [[assertion]]|[[testCase]]
  )* [**code] }
  )+

methodDeclarator : [methodName] [[parameterList]]

methodThrowsExceptions : throws [~exception] ( ,
[~exception] )*

parameterList : (
  ([[parameter]]
  ( , [[parameter]]
  )*
  )? )

parameter : [[typedName]]


testCase : [=isTimed:before|after]? [=isConcrete:JUnit|
PhpUnit|RubyUnit]? [=isOverride:override]? test
```

```
[~testCaseName] {
  ( [[assertion]]
    | [[testAction]]
    | [[testInit]]
    | [[testInitAtt]]
    | [[testInitAttWithMethodCall]]
    | [[comment]]
    | [[reqImplementation]]
    | [[ifStatement]]
    | [[forStatement]]
  )* }

assertion : [=isTimed:before
    |after]? [=assertType:assertTrue
    |assertFalse
    |assertEqual
    |assertNull
    |assertMethod] [**code] ;

testAction : [~objectName]
  (
  (.
  )[~methodName]
  )? ( [[testActionParameterList]] ) ;

testMethodCall : [~objectName]
  (
  (.
  )[~methodName]
  )? ( [[testActionParameterList]] )

testActionParameterList : [[testParameter]]?

testParameter : [[pValue]] ( , [[pValue]] )*

testInit : [~identifier] [~objectName] ( [[testParameter]]? ) ;

testInitAtt : [~identifier]? [~attributeName] = [[pValue]] ;

testInitAttWithMethodCall : [~identifier]? [~attributeName] =
[[testMethodCall]] ;

//paraTestAction : [~methodName] OPEN_ROUND_BRACKET (
[[pValue]] ( , [[pValue]] ) )* CLOSE_ROUND_BRACKET
```

```
pValue- : ( [name] | "[name]" )

interfaceTest : test [~testName] ;

testSequence : testSequence [~sequenceName] {
[[testSequenceEntry]] }

testSequenceEntry : [~name] ( -> [~name] )* ;


//top-level initialization

testClassInit : test init { [**code] }



// Generic test case rules
genericTestCase : generic test [~testCaseName] (
[[genericElement]] ) {
  ( [**code]
  )* }

genericElement- : [elementType] [=genericElement:attribute
    |method
    |association] [[elementFix]]? [[genericMethodParameters]]?

genericMethodParameters- : [~elementType] ( , [~elementType]
)*

elementFix :
  (.
  ) [=fixType:prefix
    |suffix
    |regex] ( [fixValue] )


//testcase if statement
ifStatement : if ( [**condition] ) { [**code] }
[[ifElseStatement]]* [[elseStatement]]?

ifElseStatement : if else ( [**condition] ) { [**code] }

elseStatement : else { [**code] }
```

```
// for statement
forStatement : for ( [**forCode] ) { [**code] }



// Details of associations: See manual page AssociationDefinition

association : [=modifier:immutable]? [[associationEnd]]
[=arrow:--
    |->
    |<-
    |><
    |<@>-
    |-<@>] [[associationEnd]] ;

symmetricReflexiveAssociation : [[multiplicity]] self [roleName]
;

inlineAssociation : [=modifier:immutable]? [[inlineAssociationEnd]]
[=arrow:--
    |->
    |<-
    |><
    |<@>-
    |-<@>] [[associationEnd]] ;

inlineAssociationEnd : [[multiplicity]] [~roleName]? [[isSorted]]?

singleAssociationEnd : [[multiplicity]]
  (
  ( [otherEndroleName] [type] [roleName]
  )
    |
  ( [type] [~roleName]?
  )
  ) ;


associationEnd : [[multiplicity]] [type] [~roleName]? [[isSorted]]?

multiplicity- : [!lowerBound:\d+|[**]] .. [!upperBound:\d+|[**]]
```

```
| [!bound:\d+|[**]]

isSorted- : sorted { [priority] }


// Details of attributes. See user manual page AttributeDefinition
attribute : [[simpleAttribute]]
    | [[autouniqueAttribute]]
    | [[multivaluedAttribute]]
    | [[derivedAttribute]]
    | [[complexAttribute]]

simpleAttribute- : [~name] ;

autouniqueAttribute- : [=autounique] [~name] ;

multivaluedAttribute- : [=unique]? [=lazy]? [=ivar]?
[=modifier:immutable
    |settable
    |internal
    |defaulted
    |const
    |fixml]? [type]? [[list]] [~name]
  (= { [**value] }
  )? ;

derivedAttribute- : [=modifier:immutable
    |settable
    |internal
    |defaulted
    |const
    |fixml]? [[typedName]] =
  ( [[moreCode]]
  )+

complexAttribute- : [=unique]? [=lazy]? [=ivar]?
[=modifier:immutable
    |settable
    |internal
    |defaulted
    |const
    |fixml]? [[typedName]]
  (= [**value]
  )? ;
```

```
// Keys are used to define quality and hash codes, plus Sql
keys.
// See user manual page KeysforEqualityandHashing
defaultKey : key { }

key : key { [keyId] ( , [keyId] )* }


// Depend clause. See user manual page Dependclause
depend- : depend [depend] ;


// Anything inside a class that is not parsable by Umple is
passed on to the base language
// compiler. To raise warnings when this occurs, use the
strictness directive.
extraCode- : [**extraCode]


list- : [!list:\[\s*\]]

baseType- : [!baseType:[a-zA-Z0-9_$]+]

nestedType- : [[baseType]] ( < [[argType]] > )?

argType- :
  ( ? | [[nestedType]] [[list]]?
  )
  ( , [[argType]]
  )?

type : [[nestedType]] ( ... )?

typedName- : [[type]]? [[list]]? [~name]

// The master of this part of the Umple grammar is available
at
// https://github.com/umple/umple/blob/master/cruise.umple/src/trait/
umple_traits.grammar

// Copyright: All contributors to the Umple Project
```

```
// This file is made available subject to the open source
license found at:
// https://umple.org/license

traitDefinition : trait [name] [[traitParameters]]? {
[[traitContent]]* }

traitContent- : [[mixsetDefinition]]
     | [[testCase]]
     | [[requiredModelElements]]
     | [[comment]]
     | [[reqImplementation]]
     | [[traitDefinition]]
     | [[trace]]
     | [[position]]
     | [[displayColor]]
     | [[abstract]]
     | [[keyDefinition]]
     | [[softwarePattern]]
     | [[depend]]
     | [[symmetricReflexiveAssociation]]
     | [[attribute]]
     | [[stateMachine]]
     | [[inlineAssociation]]
     | [[concreteMethodDeclaration]]
     | [[abstractMethodDeclaration]]
     | [[constantDeclaration]]
     | [[invariant]]
     | ;
     | [[exception]]
     | [[extraCode]]

traitParameters : < [[traitFullParameters]]
  ( , [[traitFullParameters]]
  )* >

traitFullParameters : [~parameter]
  ([[traitParametersInterface]]
  )?
  ( = [~defaultType]
  )?

traitParametersInterface- : isA [~tInterface]( &
```

```
[~tInterface])*

requiredModelElements : require ([[requiredState]] |
[[requiredEvent]])

requiredState : [smName] ;

//.[~stateName](.[~stateName])* ;
requiredEvent : [smName] (
  ( [parameter]
  ( , [parameter]
  )*
  )? ) ;

//.[~stateName](.[~stateName])*


//iE = Include Exclude
AllInclusionExclusionAlias- : [[InclusionExclusionAlias]]
  ( , [[InclusionExclusionAlias]]
  )*

InclusionExclusionAlias- : [[functionIncludeExcludeAlias]] |
[[StateMachineIncludeExcludeAlias]]

functionIncludeExcludeAlias- : [[functionInExAlias]]
  ( , [[functionInExAlias]]
  )*

functionInExAlias- : [[functionAliasName]] | [[iEFunction]] |
[[traitAppliedParameters]]

iEFunction : [=modifier:+|-] [~methodName] [[iEParameterList]]

iEParameterList : (
  ( [parameter]
  ( , [parameter]
  )*
  )? )

functionAliasName : [=modifier:+]?
  (
  ( [~smName]
    | [!smPattern:\d+|[**]]
```

```
  ) .
  )? [~methodName] [[iEParameterList]] as [[IEVisibilityAlias]]


IEVisibilityAlias- : ([[IEVisibility]] [~aliasName]?) |
([~aliasName])

IEVisibility- : [=iEVisibility:public|private|protected]

traitAppliedParameters : [~pName] = [~rName]


StateMachineIncludeExcludeAlias- : [[ StateMachineInExAlias]]
  ( , [[StateMachineInExAlias]]
  )*

StateMachineInExAlias- : [[StateMachineAliasName]] |
[[iEStateMachine]]

//| [[StateMachineTransitionAlias]]
StateMachineAliasName :
  ([=iEStateMachineModifier:+]
  )? [~smName]
  ([[StateNames]]
  )? as [~smDesName]
  ([[DesStateNames]]
  )?

StateNames : . [~sName] ( [[StateNamesPassing]] )*

DesStateNames :[[StateNames]]

StateNamesPassing : [[StateNames]]

//iEStateMachine : [=iEStateMachineModifier:+|-] [~smName]
[[StateNames]]?
iEStateMachine : [=modifier:+|-] [~smName] ( [[StateNames]] (
( [[iEParameterList]] [[guardOption]]? )
    | ( . [[guardOption]] ) )? )?

guardOption : [[guard]] | []

//This is for events in state machines
//StateMachineTransitionAlias : ( [~smName] | [!smPattern:\d+|
```

```
[*]]) . [~eventName] [[iEParameterList]] as [~AliasName]
// The master of this part of the Umple grammar is available
at
// https://github.com/umple/umple/blob/master/cruise.umple/src/fixml/
umple_fixml.grammar

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source
license found at:
// https://umple.org/license

//This is Umple's grammar for parsing fixml documents (add
add-on feature)
fixml : [[fixmlDefinition]] | [[fixmComment]] | [[fixmlDoc]]

fixmlDoc : <! [**value] >

fixmComment : <?xml [[tagDefinition]]* ?>

fixmlDefinition : <FIXML> [[fixmlContent]]* </FIXML>

fixmlContent : [[endContent]] | [[ extendContent ]]

endContent : < [~name] ( [[tagDefinition]] )* />

extendContent : < [~name]
  ( [[tagDefinition]]
  )* >
  ( [[endContent]]
    | [[extendContent]]
    | [[attContent]]
  )* <
  ( /
  ) [~name] >

tagDefinition : [name] = "[**value]"

attContent : < [~name] > [**value:\<] < ( / ) [~name] >

// The master of this part of the Umple grammar is available
at
// https://github.com/umple/umple/blob/master/cruise.umple/src/
strcture/umple_structure.grammar
//@author Ahmed M.Orabi {@link ahmedvc@hotmail.com}
```

```
//@author Mahmoud M.Orabi {@link mahmoud_3rabi@hotmail.com}

primitiveDefinition : primitive [name] { [[primitiveBody]]* }

primitiveBody- : [[isA]] | [[constraint]]


// Port Definition
portName : ( [~classname] . )? [name]

portMultiplicity : [!portMultiplicity:\[[0-9]+\]]


typedPortName : [~type]? [~portName] [[portMultiplicity]]?


portDefinition : [[portClass]] | [[portDeclaration]]

portClass- : [=port] ;

portDeclaration : [=modifier:public
    |protected
    |private]? [=inverse:conjugated]? [!direction:
  (in
    |out
    |port
  )\s]
  ( [[typedPortName]]
  ) ;


// Port Connectors
portBindingDefinition :
  ( [~fromClassname] .
  )? [fromPort] ->
  ( [~toClassname] .
  )? [toPort] [[bindinHandler]]

bindinHandler : { [**code] } | ;


// Port Watchlist Definition
portWatch- :
  ( [[constraintList]] | [[comment]] | [[reqImplementation]]
```

```
  )*


// Active Method Definition
activeMethodDefinition : [[portWatch]]? [=modifier:public
    |protected
    |private]? [type]? [=activeType:atomic
    |synchronous
    |intercept]? [=active] [[activeMethodDeclarator]]
[[activeMethodBody]]+

activeMethodDeclarator : [~methodName] [[parameterList]]?


//activeMethodBody : [[activeDirectionHandler]] { (
[[activeTrigger]] )* [**code] }
activeMethodBody : [[activeMethodBodyContent]]
[[inverseDirectionHandler]]?

inverseDirectionHandler : -> [[portWatch]]?
[[activeMethodDeclarator]] [[activeMethodBodyContent]]

activeMethodBodyContent : {
  ( [[comment]] | [[reqImplementation]] | [[activeTrigger]] |
[**code]
  )* }


activeTrigger : [[hitchConstraint]]? [[constraintList]]? [=trigger:/]
[[activeTriggerBody]] [[thenDefinition]]? [[resolveDefinition]]?

activeTriggerBody- :
  ( [[deferredList]] | [[activeTriggerDefinition]]
  )

deferredList : [ [[activeTriggerDefinition]]
  ( , [[activeTriggerDefinition]]
  )* ]

activeTriggerDefinition- : [[anonymousTriggerBody]] | [[invoke]]


thenDefinition : .then ( [[anonymousTriggerBody]]? )
```

```
resolveDefinition : .resolve ( [[anonymousTriggerBody]]? )


hitchConstraint : [=clause:after|poll] ( [timer] )

constraintList : [ [[basicConstraint]]
  ( , [[basicConstraint]]
  )* ]

basicConstraint- : [[timeConstraint]] | [[messageConstraint]] |
[[constraint]]

timeConstraint : [=clause:latency|period|timeout] ( [timer] )

messageConstraint : [=clause:priority] ( [priorityValue] )

invoke :
  ( [~classname] .
  )? [name] (
  ( [parameter]
  ( , [parameter]
  )*
  )? ) ;

anonymousTriggerBody : { [**code] }


// The master of this part of the Umple grammar is available
at
// https://github.com/umple/umple/blob/master/cruise.umple/src/
template/umple_template.grammar
//@author Ahmed M.Orabi {@link ahmedvc@hotmail.com}
//@author Mahmoud M.Orabi {@link mahmoud_3rabi@hotmail.com}

// Template capabilities are a sublanguage of Umple.
// See user manual page: BasicTemplates

templateDefinition : template [name] { [[templateContent]]* }

templateContent- : [[comment]]
     | [[emitMethod]]
     | [[templateAttributeDefinition]]
     | [[trace]]
     | [[position]]
```

```
  | [[displayColor]]
  | [[abstract]]
  | [[keyDefinition]]
  | [[softwarePattern]]
  | [[depend]]
  | [[symmetricReflexiveAssociation]]
  | [[attribute]]
  | [[templateAttribute]]
  | [[stateMachine]]
  | [[inlineAssociation]]
  | [[concreteMethodDeclaration]]
  | [[constantDeclaration]]
  | [[invariant]]
  | ; |[[exception]]
  | [[extraCode]]


templateAttributeDefinition : [[templateName]]
[[templateAttribute]]

templateName :
  ( [~classname] .
  )? [name] [[templateParameters]]?

templateParameters : ( [[templateParameter]]
  ( , [[templateParameter]]
  )* )

templateParameter- : [parameter] | " [**parameter] "


emitMethod : [=modifier:public
    |protected
    |private]? [=static]? [type]? [=emit] [[methodDeclarator]]
[[templateList]]?;

templateList- : (
  ( [[templateName]]
  ( , [[templateName]]
  )*
  )? )
```

```
templateAttribute# : <<! [[templateAttributeContent]]* !>>

templateAttributeContent- : [[templateExpression]]
     | [[templateComment]]
     | [[templateCodeBlock]]
     | [[templateExactSpaces]]
     | [[templateInclude]]
     | [[templateText]]

templateText# : [**templateTextContent:<<(=|#|/[*]|$|@)]


templateComment# : <</* [**templateCommentContent] */>>

templateExpression# : <<=
  ( [[templateExpression]]
    | [**templateExpressionContent:<<
  (=|#|/[*]|$|@
  )]
  )+ >>

templateCodeBlock# : <<#
  ( [[templateExpression]] | [**templateLanguageCode:<<
  (=|#|/[*]|$|@
  )]
  )* #>>

templateExactSpaces# : <<$ [**templateExactSpacesContent] >>

templateInclude# : <<@ [[templateName]] >>

//templateOpen- : <
//templateClose- : !>>

// The master of this part of the Umple grammar is available
at
// https://github.com/umple/umple/blob/master/cruise.umple/src/
stateMachine/umple_state_machines.grammar

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source
license found at:
// https://umple.org/license
```

```
// State machine elements in Umple. See user manual page:
BasicStateMachines
stateMachineDefinition : statemachine [=queued]? [=pooled]?
[name] { [[state]]* }

stateMachine : [[enum]]
    | [[inlineStateMachine]]
    | [[referencedStateMachine]]
    | [[activeDefinition]]


activeDefinition : [=active] [[codeLangs]] [name]? [[moreCode]]+


//Issue 148
inlineStateMachine : [=queued]? [=pooled]? [~name] {
  ( [[comment]]
    | [[reqImplementation]]
    | [[state]]
    | [[trace]]
    | [[mixsetDefinition]]
    | [=||]
    | [[standAloneTransition]]
  )* }

referencedStateMachine : [name] as [definitionName]
  ( { [[extendedStateMachine]] } | ;
  )


//Issue 547 and 148
extendedStateMachine# :
  ( [[comment]]
    | [=changeType:+| - |- |*]? [[state]]
    | [[standAloneTransition]]
  )*


// An enum is a state machine that has no events
// stateName is prefixed with ~ to match alphanumeric names
only.
// This is needed to solve issue 399, which is cause when a
terminating } is parsed as part of the statename.
```

```
enum : [~name:key] { [~stateName]? (, [~stateName])* }

state : [=final]? [stateName] { [[stateInternal]]* }

//Issue 547 and 148
stateInternal-# : [[comment]]
    | [[reqImplementation]]
    | [=changeType:+| - |- |*]? [[stateEntity]]
    | [[standAloneTransition]]
    | [[concreteMethodDeclaration]]
    | [**extraCode]

stateEntity- : [=||]
    | [[mixsetDefinition]]
    | [[entryOrExitAction]]
    | [[autoTransition]]
    | [[transition]]
    | [[activity]]
    | [[state]]
    | [[displayColor]]
    | [[trace]]
    | ;

autoTransition : [[activity]]? [[autoTransitionBlock]]

// Autotransitions have no event. The transition is
immediately taken
// or taken after the do activity ends[[guard]]? ->
[[action]]?
// The action can come before or after the arrow
autoTransitionBlock- : [[guard]]?
  ( [[action]] -> | -> [[action]] | ->
  ) [stateName] ;

// A transition guard can come before or after the arrow
// The order of guard and event definition can also be
interchanged
transition :
  ( [[eventDefinition]] [[guard]]
```

```
    | [[guard]] [[eventDefinition]]
    | [=unspecified]? [[guard]]
    | [[eventDefinition]]
  )?
  ( [[action]] ->
    | -> [[action]]
    | ->
  ) [stateName] ;


//Issue148
standAloneTransition :
  ( [[eventDefinition]] [[guard]]
    | [[guard]] [[eventDefinition]]
    | [=unspecified]? [[guard]]
    | [[eventDefinition]]
  )? [~fromState]
  ( [[action]] ->
    | -> [[action]]
    | ->
  ) [~toState] ;


eventDefinition- : [[afterEveryEvent]] | [[afterEvent]] | [~event]
  ( [[parameterList]]
  )?


// The timer in a timed event can be an number, variable,
function() or function(num)
afterEveryEvent- : afterEvery ( [!timer:[+*/a-zA-
Z0-9_\.-]+(\([0-9\.]*\))?] )

afterEvent- : after ( [!timer:[+*/a-zA-
Z0-9_\.-]+(\([0-9\.]*\))?] )


// An action can be executed on a transition, or on entry or
exit
action : / [[moreCode]]+

entryOrExitAction : [=type:entry|exit] [[guardCode]]? /
[[moreCode]]+
```

```
guardCode :[ [**code] ]


// A do activity is long-lasting and can be interrupted
activity : do [[moreCode]]+


guard : [ [[constraint]] ]

// The master of this part of the Umple grammar is available
at
// https://github.com/umple/umple/blob/master/cruise.umple/src/trace/
umple_traces.grammar

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source
license found at:
// https://umple.org/license

// Trace capabilities of the MOTL sublanguage of Umple.
// See user manual page: ModelOrientedTracingLanguage(MOTL)

// ======= TRACER
// ======= grammar rules - The tool to output the trace: Tracers
tracerDirective : tracer [tracerType] [[tracerOptions]] ;

tracerOptions- : [[logConfig]]* [[traceMessageHeader]]?
[tracerArgument]?

logConfig :
  (root = [rootLevel]
  )
    |
  (monitorInterval = [monitorInterval]
  )
    |
  ( [logLevel]
  ( , [logLevel]
  )* = [logAppender]
  ( , [logAppender]
  )*
  )
```

```
traceMessageHeader : [=switch:on|off] : [option] ( , [option]
)*


// ======= TRACE
// ======= grammar rules

trace : [[traceDirective]] | [[traceCase]]


traceDirective : trace [[Prefix]]? [[traceEntity]] [[Postfix]] ;


traceEntity- : [traceEntity]
  ((
  )?
  ()
  )?
  ( , [traceEntity]
  ((
  )?
  ()
  )?
  )*


Prefix : [[PrefixOption]] ( , [[PrefixOption]] )*

PrefixOption- : [=option:set|get|in|out|entry|exit|cardinality|
add|remove]


Postfix- :
  ( [[traceCondition]]
    |[[traceFor]]
    |[[tracePeriod]]
    |[[traceDuring]]
    |[[traceLevel]]
    |[[traceRecord]]
    |[[executeClause]]
    |[[logLevel]]
    |[[traceCaseActivation]]
  )*
```

```
traceCondition : [=conditionType:where|until|after|giving]? [
[[constraintToken]] ]

traceFor- : for [traceFor]

tracePeriod- : period [tracePeriod]

traceDuring- : during [traceDuration]

traceLevel- : level [traceLevel]

traceRecord- : record [[recordEntity]]

recordEntity- :
  ( only
  )?
  ( " [**recordString] " | [traceRecord]
  )
  ( , [traceRecord]
  )*

logLevel- : logLevel [[logLevelOption]]
  ( , [[logLevelOption]]
  )*

logLevelOption- : [=logLevel:trace
    |debug
    |info
    |warn
    |error
    |fatal
    |all
    |finest
    |finer
    |fine
    |config
    |warning
    |severe]

executeClause- : execute { [**traceExecute] }


traceCase : [[traceCaseDef]] | [[traceCaseActivation]] |
[[traceCaseDeactivation]]
```

```
traceCaseDef- : tracecase [tracecase_name] { [[traceDirective]]*
}

traceCaseActivation- : activate [tracecase_act_name]
  (onAllObjects | onThisThreadOnly
  )? ;

traceCaseDeactivation- : deactivate [tracecase_deact_name]
onThisObject [[deActivateFor]]? ;

deActivateFor- : for [deactivate_for]



//--------- Old rules (left as reference) ---------
//++Tracer
//traceType : tracer [tracerType] ( [=onoff:on|off] :
[[tracerOptions]] ( , [[tracerOptions]] )* )* (
[=verbisty:verbose] )? ( [tracerArgument] )* [[log4jConfig]]*
([[tracerOptions]])* ;
//traceType : tracer [tracerType] ( , [tracerType] )*
[[log4jConfig]]* ([[tracerOptions]])* ;
//traceType : tracer [=tracerType:Console|File|String|Visual]
( , [=tracerType:Console|File|String|Visual] )* ( [=onoff:on|
off] : [[tracerOptions]] ( , [[tracerOptions]] )* )* (
[=verbisty:verbose] )? ( [tracerArgument] )* ;
//tracerOptions- : [[tracerOn]]? [=option:Time|time|Thread|
thread|File|file|Line|line|Trace|trace|Trigger|trigger|
TraceFile|tracefile|TraceLine|traceline|TriggerLine|
triggerline|TriggerFile|triggerfile|Class|class|Object|object|
Operation|operation|Name|name|Value|value] [=option:Time|time|
Thread|thread|File|file|Line|line|Trace|trace|Trigger|trigger|
TraceFile|tracefile|TraceLine|traceline|TriggerLine|
triggerline|TriggerFile|triggerfile|Class|class|Object|object|
Operation|operation|Name|name|Value|value]*
//tracerOn : on:|off:
//log4jConfig : (root = [rootLevel]) | (monitorInterval =
[monitorInterval]) | (OPEN_ROUND_BRACKET) [log4jLevel] ( ,
[log4jLevel] )* (CLOSE_ROUND_BRACKET) = (OPEN_ROUND_BRACKET)
[log4jAppender] ( , [log4jAppender] )* (CLOSE_ROUND_BRACKET)
//-----------------------------------------------
// The master of this part of the Umple grammar is available
at
```

```
// https://github.com/umple/umple/blob/master/cruise.umple/src/layout/umple_layout.grammar

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source license found at:
// https://umple.org/license

// Layout elements are injected into classes as mixins by diagramming tools such
// as UmpleOnline. Programmers can tweak the layout textually.
position- : [[associationPosition]] | [[elementPosition]]

elementPosition : position [x] [y] [width] [height] ;

associationPosition : position.association [name] [[coordinate]] [[coordinate]] ;

coordinate : [x] , [y]

displayColor : ( displayColor | displayColour ) [**colorValue] ;

// The master of this part of the Umple grammar is available at
// https://github.com/umple/umple/blob/master/cruise.umple/src/filter/umple_filter.grammar
//
// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source license found at:
// https://umple.org/license

// This feature allows drawing diagrams of part of a model

filter : filter
  ( [filterName]
  )? {
  ([[filterStatement]]
  )* }


filterStatement- : [[filterCombinedValue]] | [[filterNamespace]] | [[filterValue]] | [[hops]]
```

```
hops- : hops {([[super]] | [[sub]] | [[association]])*}

filterValue : include ([classname]) ( , [classname] )*;

super : super [superNum];

sub : sub [subNum];

association : association [associationNum];

filterNamespace : namespace [Namespace] (, [Namespace])* ;

filterCombinedValue : includeFilter ([filterName]) ( ,
[filterName] )*;

// The master of this part of the Umple grammar is available
at
// https://github.com/umple/umple/blob/master/cruise.umple/src/mixset/
umple_mixsets.grammar

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source
license found at:
// https://umple.org/license

//Mixsets allow creation of mixins composed from multiple
locations plus some constraints regarding usage of these
mixins.
// See user manual page BasicMixsets

mixsetDefinition : mixset [mixsetName]
  ( [[mixsetInnerContent]]
    | [[requirement]]
    | [[mixsetInlineDefinition]]
  )
```

```
mixsetInnerContent- : { [[extraCode]] | [[reqImplementation]] }


mixsetInlineDefinition- :
  ( [entityType] [entityName]
  ( [[mixsetInnerContent]]
    | [entityOneElement]
  )
    | [oneElement]
  )


// Specialize a mixset to be a feature.
mixsetIsFeature : isFeature


// require statement allows adding dependencies between
mixsets.

requireStatement : require ( [=subfeature] )? [[requireBody]]


requireBody- : [
  (
  ([[requireLinkingOptNot]]
  )?
  ((
  )* [[requireTerminal]] [[requireList]]
  )]


requireList- :
  ([[requireLinkingOp]]
  ((
  )* [[requireTerminal]]
  ()
  )*
  )*


requireLinkingOp :
  ([[requireLinkingOptNot]]
    | [=and:&|&&|and|,]
    | [!or:
```

```
  (|][|]?|or|;
  )]
    | [=xor:xor|XOR]
  )
```

```
requireLinkingOptNot- : ([=opt:opt] | [=not:not] )
```

```
requireTerminal : [~targetMixsetName] | [[multiplicityTerminal]]
```

```
multiplicityTerminal- : [[multiplicity]] of { [~targetMixsetName]
  (, [~targetMixsetName]
  )* }
```

Grammar Notation

In Umple we use our own extended EBNF syntax for the grammar, with our own parsing tool. The reason for this is that we wanted several advanced features in the grammar and parser.

Samples of syntax in this user manual are generated directly from the input grammar files used to parse Umple code.

Our syntax offers a very simple mechanism to define a new language, as well as extend an existing one. We will be using examples to help explain our syntax.

Example of a simple rule

Let's start with a simple assignment rule (not part of Umple, just an example):

```
assignment : [name] = [value] ;
```

Above, the rule name is "assignment". An assignment is comprised of a non-terminal called "name", then the equals symbol ("="), a non-terminal "value" and finally a semi-colon symbol (";").

A **non-terminal** by default as shown above is a sequence of **characters** that is a **non-whitespace** that is delimited by the next **symbol** (based on the specified grammar). In the above case, the non-terminal "name" will be defined by the characters leading up to either a space, newline, or an equals ("=").

Here are a few examples that satisfy the assignment rule above:

- key = "one";
- wasSet=true;
- numberOfItems =7;

Rules that refer to other rules

Let us now consider nesting sub-rules within a rule. Sub-rules are words between [[and]]. Again, the following is illustrative, and is not Umple itself

```
directive- : [[facadeStatement]] | [[useStatement]]

facadeStatement- : [=facade] ;
```

```
useStatement : use [=type:file|url] [location] ;
```

Above, we have three rules, "directive", "facadeStatement", and "useStatement". A "directive" is either a "facadeStatement" or a "useStatement" (the "or" expression is defined by the vertical bar "|"). As indicated earlier, to refer to a rule within another rule, we use double square brackets ("[[" and "]]").

By default, rule names are added to the tokenization string (the result of parsing the input, shown in blue below). But, some rules act more like placeholders to help modularize the grammar (and to promote reuse). To exclude a rule name (and just the name, the rule itself will still be evaluated and tokenized as required), simply add a minus ("-") at the end of its name.

Above, we see that the rule names "directive", and "facadeStatement" are not added to the tokenization string because of the trailing minus signs..

For example, the text "facade;" is tokenized as follows:
[facade:facade]

Without the ability to exclude rule names, that same text would be tokenized with the following additional (and unnecessary) text:
[directive][facadeStatement][facade:facade]

Terminal symbols and constants

Symbols (i.e.terminals), such as "=" and ";" are used in the analysis phase of the parsing (to decide which parsing rule to invoke), but they are not added to the resulting tokenziation string for later processing.

If we want to tokenize symbols, we can create a constant using the notation

```
[=name]
```

In the earlier example we see that a "facadeStatement" is represented by the sequence of characters "facade" (i.e. a constant).

To support *lists* of potential matches we use a similar notation

```
[=name:list|separated|by|bars].
```

In the earlier example, we see that the "type" non-terminal can be the constant string sequence "file" or "url".

Hence, here are a few examples that satisfy the earlier example:

- facade;
- use file Parser.ump;
- use url http://cruise.site.uottawa.ca/Parser.ump;

Optionality and repeating

Parentheses can be used to group several elements in the grammar into a single element for the purposes of the following special treatment.

An asterisk * means that zero or more of the preceding elements may occur.

A plus sign + means that one or more of the preceding elements must occur.

A question mark ? means that the preceding element may occur.

Avoiding consumption of whitespace

Normally when parsing, all whitespace (spaces, carriage returns, tabs, etc.) around tokens are ignored, and the token output by the parser does not contain them. However, if a # symbol is found after the rule (on the left hand side) then all whitespace is preserved. This is useful for cases where that space is actually useful, such as in Umple's templates.

Comments and arbitrary input

The grammar syntax supports a simple mechanism for non-terminals that can include whitespace (e.g. comments). Text in [**] such as [**arbitraryStuff] is not parsed. However if the rule name is followed by a colon, such as [**templateTextContent:<<(=|#|/[**] then a pattern for different types of brackets that can be internally parsed can be specified. So the above says ignore everything except things in <<= <<# <</* , which will be processed.

Let us consider the rules to define inline and multi-line comments.

```
inlineComment- : // [*inlineComment]

multilineComment- : /* [**multilineComment] */
```

The [*name] (e.g. [*inlineComment]) non-terminal will match everything until a newline character. The [**name] (e.g. [**multilineComment]) non-

terminal will match everything (including newlines) until the next character sequence is matched. In the case above, a "multilineComment" will match everything between "/*" and "*/".

Here are a few examples that satisfy the assignment rule above:

- // remove all references to "x" once complete
- /* This class will help calculate
  your overdue library fees */

Special matching cases

By default, [name] matches identifiers that can include underscore and certain other symbols. To match alphanumeric identifiers only, then use

```
[~name]
```

```
To match based on a regular expression (such as a sequence of
one or more digits in this case):
```

```
[!bound:\d+]
```

To match one or more identifiers, but with some being optionally omitted use this notation:

```
[attr_qualifier,attr_type,attr_name>2,1,0]
```

This states that there will be one, two, or three identifiers. The priority of inputs is attr_name,attr_type and attr_qualifier.

Parsing a language of your own

The parser that reads grammars in this notation can parse other languages too. An Umple program brings it in with `use lib:UmpleParser.ump;`, then reads a grammar file and parses a text file into a tree of tokens:

```
RuleBasedParser parser = new RuleBasedParser();
parser.addGrammarFile("greetings.grammar");
ParseResult result = parser.parse(new File("greetings.txt"));
```

When the grammar cannot parse the text, `result.getWasSuccess()` is false and the result holds error 1500 for the line where parsing stopped; otherwise `parser.getRootToken()` is the root of the tokens. The UmpleParser README has a complete example.

API Summary

The following table describes the API generated from various Umple features. This is designed as a quick reference. To find the exact API for any system that would be generated in Java, use UmpleOnline and request to generate JavaDoc output.

Summary of the API generated by Umple from attributes, associations, state ma

API GENERATED IN ALL CASES

void delete()	removes object, correctly handling casc

any methods in the code are emitted as-is; they are made public if this is not

API GENERATED FROM ATTRIBUTES

T is the type of the attribute (String if omitted); note that Umple builtin types su
z is the attribute name
Note that the API for array attributes is shown as a column in the associations t
Umple concept
Umple notation

Querying value

T getZ()	return the value
boolean isZ()	return true if value is true
boolean equals(Object)	is equal to another?

Mutating

boolean setZ(T)	mutates the attribute
boolean resetZ()	restores original default

Other

T getDefaultZ()	returns original default
int hashCode()	Likely unique value

constructor args
initial value

API GENERATED FROM ASSOCIATIONS

X is the name of current class
W is the name of the class at the other end of the association; used in method
r is a role name used when referring to W (optional, except in reflexive associat
Umple concept
Umple notation

Querying link(s)

W getW()	return the W
W getW(index)	pick a specific linked W
List<W> getWs()	ges immutable list of links
boolean hasWs()	is cardinality > 0?
int indexOfW(W)	look up specific W -1 if none
int numberOfWs()	cardinality

Mutating

boolean setW(W)	add a link to existing W
W addW(args)	construct a new W and adds link
boolean addW(W)	add a link to existing W
boolean setWs(W…)	add a set of links
boolean removeW(W)	remove link to W if possible

Other

static int minimumNumberOfWs()	multiplicity lower bound
static int maximumNumberOfWs()	multiplicity upper bound if > 1
static int requiredNumberOfWs()	fixed multiplicity if > 1
boolean isNumberOfWsValid()	false if breaching constraints
constructor args	
initial value	

API GENGERATED FROM STATE MACHINES

sm is the name of the state machine

Umple concept

enum getSm()	current state for statemachine sm
String getSmFullName()	current state for statemachine sm
enum getSmS2()	state of substate s2
boolean e()	event method e (for all events)
boolean setSm(enum)	set state directly
Substates allowed	
initial value	first state listed

API GENERATED FROM SINGLETON

static X getInstance()

Umple messages

General information about Umple errors and warnings

The subsequent pages in the user manual describe each of the errors or warnings that can be emitted by the compiler. These can appear in UmpleOnline, Eclipse and the command-line compiler.

An **error** is a situation where the compiler can't generate code because the Umple code doesn't make sense; i.e. there is something critical missing, or there is an inconsistency.

A **warning** is a situation where code can be generated, but the resulting system may not to have the intended semantics (it may not behave as intended) because of an inconsistency, redundancy or assumption that may not be correct.

For developers of Umple itself: The file defining the messages in English is called en.error and is located in cruise.umple/src. Errors have severity 1 or 2, and warnings have severity 3, 4 and 5. When adding a message, a user manual page should be added describing it.
All messages should be detected in the parsing phase by calling setFailedPosition.
It is intended in the future that internationalized versions of Umple will be created; for example fr.error would be the French version. In the near future, it is intended to enable messages from Java or other compilers to be passed through as part of the Umple compilation process; these would be the same as the native language compiler, except that line numbers would be changed to point to the correct line of Umple code.

W001 Singleton With Non-Lazy Attribute

Umple semantic warning reported when a singleton class has an attribute that is not marked lazy

A singleton class can't have any arguments in its constructor. In general in Umple, unless an attribute is specified as 'lazy', then an argument for the attribute is added to the constructor. In the case of singletons, this is not done. This warning is to let programmer know this. To avoid the warning, add the keyword 'lazy' to all attributes of a singleton class. However, whether or not this is done, the generated code will behave as if it had been done.

Example

```
view plain

01. // This example generates the warning message
02. class X {
03.   singleton;
04.   Integer x;
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // The following shows how to avoid the warning
02. class X {
03.   singleton;
04.   lazy Integer x;
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W002 Singleton With Required Object

Umple semantic warning reported when a singleton class has an association with a multiplicity lower bound of 1 at the other end of the association

The constructor of a singleton can't take any arguments. Therefore, it can't have a required link to another object, since this would have to be specified in the constructor. Any associations to other classes should therefore have multiplicity with lower bound of zero.

Note: Conceptually, this warning would also have been produced if the lower bound was something greater than 1 (e.g. 5). However, Umple treats a multiplicity specified with a lower bound > 1 as if the multiplicity lower bound was 0. The programmer is supposed to add the n elements immediately after construction. The API has a validity check method to verify this has been done.

Example

view plain

```
01. // This example generates the warning message
02. class X {
03.   singleton;
04.   0..1 -- 1 Y;
05. }
06.
07. class Y {
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following shows how to avoid the warning
02. class X {
03.   singleton;
04.   0..1 -- 0..1 Y;
05. }
06.
```

```
07. class Y {
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W003 Redundant Lazy With Initialization

Umple syntactic warning reported when the lazy keyword in specified when there is also an initializer

The function of the 'lazy' keyword is to indicate that an attribute is not to appear in the constructor. However an initializer for the attribute has the same effect, so specifying both 'lazy' and an initializer is redundant, and hints that there may be a mistake. The solution is just to remove the extraneous 'lazy'.

Example

```
view plain

01. // This example generates the warning
02. class X3lazy {
03.   lazy Integer a = 1;
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // This shows how the warning can be removed.
02. class X3lazy {
03.   Integer a = 1;
04. }
05.
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E004 Invalid Multiplicity

Umple syntactic error reported when an invalid multiplicity is specified

Valid multiplicities in Umple include the following, where n and m are positive integers and where n <= m:

- 0..1 (either zero or one object)
- 1 (exactly one object)
- 0..* (any number of objects)
- * (any number of objects -- a shortcut for 0..*)
- 1..* (one or more objects)
- 0..m (up to m objects)
- 1..m (between 1 and m objects)
- n..m (between n and m objects)
- n..* (at least n objects)
- m (exactly m objects)

When this error message appears, the multiplicity doesn't fit any of the above patterns. A common error, for example, is to use the notation 'n' as found in Entity-Relationship Diagrams, instead of *. This is not valid in Umple; only integers, and * may appear.

Example

view plain

```
01. // This example generates the error message
02. class X {
03.   1 -- 0..1..2 Y;
04. }
05.
06. class Y {
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following shows how to avoid the error
```

```
02. class X {
03.   1 -- 0..2 Y;
04. }
05.
06. class Y {
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E005 Undefined Class in Association

Umple semantic error reported when an an association is specified to a class that cannot be found

The classes at both ends of an association must exist in the system being compiled. It might be that the specified class is spelled wrongly or has been omitted from the compilation. If you are making an association to a class in existing code, then simply declare the class as external, as shown in the third example below.

Example

```
view plain

01. // This example generates the error message
02. class X {
03.   1 -- * Y;
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // The following shows how to avoid the error
02. class X {
03.   1 -- * Y;
04. }
05. class Y {}
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Solution to The Above So the Message No Longer Appears

```
view plain

01. // It is also possible to designate that
02. // the class Y is external in the following manner
```

```
03. class X {
04.   1 -- * Y;
05. }
06. external Y {}
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

```
view plain

01. // The following will also generate the error message
02. class X {}
03. class Y {}
04.
05. association {
06.   1 X -- * Z;
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E006 Missing Default

Umple syntactic error reported when no default value is specified after an attribute that is marked defaulted

The keyword 'defaulted' means that the attribute will be given a default value if the value is not set in the constructor. Omitting the default value is therefore illogical. The default value is specified after an equals sign.

Example

view plain

```
01. // This example generates the error message
02. class X {
03.   defaulted Integer a;
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following shows how to avoid the error
02. class X {
03.   defaulted Integer a = 5;
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W007 Key Already Specified

Umple semantic warning reported when a second key statement is encountered in a class

To avoid confusion, a class should normally only have one key specification. This warning is intended to direct the programmer to list the multiple attributes within the same key directive, rather specifying them separately.

The reason for this is that without this warning, code that is separated by many lines or found in a mixin file might modify the key in a way that wasn't anticipated by the developer who is not aware of the existence of multiple key statements.

Developers can ignore this warning without any consequences. The examples below will generate the same compiled code.

Example

```
view plain

// This example generates the message
class X {
  name;
  id;
  key { name }
  key { id }
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

// This example below avoids the message
class X {
  name;
  id;
  key { name, id }
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E008 Association Class Missing End

Umple semantic error reported when an association class is only defined with one end

An association class must have two classes designated as its ends. Instances of the association class represent links between instances of the associated classes, so it is illogical for an association class to have only one end.

Example

```
view plain

01. // This example generates the error message
02. class A {}
03.
04. class B {}
05.
06. associationClass C {
07.   * A;
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // The following shows how to avoid the error
02. class A {}
03.
04. class B {}
05.
06. associationClass C {
07.   * A;
08.   * B;
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E009 Reflexive Lower Bound

Umple semantic error reported when a reflexive association has a multiplicity with a lower bound greater than one

A reflexive association must have multiplicity with lower bounds of zero at both ends (e.g. *, 0.1 or or 0..2), otherwise an illogical situation would result. For example, creating a parent-child association on class Person (the example below) with a lower bound of 2 would mean that every Person in the system must have parents, *ad infinitum*.

Example

```
view plain

01. // This example generates the error message
02. class Person {
03.   * -- 2 Person parents;
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // The following shows how to avoid the error
02. class Person {
03.   * -- 0..2 Person parents;
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W010 Singleton Multiplicity Over 1

Umple semantic warning reported when a singleton class has an association with incoming multiplicity > 1

Since there can only be one instance of a singleton class, it is logically impossible for another class to have links to more than one instance of the singleton class.

Example

```
view plain

01. // This example generates the warning message
02. class X {
03.   singleton;
04. }
05.
06. class Y {
07.    0..1 -- * X;
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // The following shows how to avoid the warning
02. class X {
03.   singleton;
04. }
05.
06. class Y {
07.    0..1 -- 1 X;
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E011 Class is Subclass Of Self

Umple semantic error reported when a class is designated as a subclass of itself.

The inheritance hierarchy cannot have cycles. It must be a strict tree. It is therefore not allowed to make a class into a subclass of itself.

Example

```
view plain

01. // This example generates the error message
02. class X {
03.   isA X;
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // The following shows how to avoid the error
02. class Y {}
03.
04. class X {
05.   isA Y;
06. }
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E012 Class Indirectly Subclass of Self

Umple semantic error reported when a class is designated as a subclass of itself, through some other class.

The inheritance hierarchy cannot have cycles. It must be a strict tree. It is therefore not allowed to make a cycle or loop, in which a class is indirectly a subclass of itself.

Example

view plain

```
01. // This example generates the error message
02. class A {
03.   isA C;
04. }
05.
06. class B {
07.   isA A;
08. }
09.
10. class C {
11.   isA B;
12. }
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following shows how to avoid the error
02. class A {
03. }
04.
05. class B {
06.   isA A;
07. }
08.
09. class C {
10.   isA B;
11. }
```

12.

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E013 Non Immutable Association

Umple semantic error reported when an immutable class is defined as having an association to a non-immutable class.

By definition, an immutable class can't change state, so it can't have an association that can change state or an association to a class that can change state.

Example

view plain

```
01. // The following example generates
02. // the error message
03. class X {
04.   Integer x;
05. }
06.
07. class Y {
08.   String s;
09.   immutable;
10.   0..1 -> 0..1 X;
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // One way to solve this is to make
02. // the association point to
03. // another immutable class
04. // But note that this only works for
05. // one-way associations
06. class X {
07.   Integer x;
08.   immutable;
09. }
10.
11. class Y {
12.   immutable;
```

```
13.   String s;
14.   0..1 -> 0..1 X;
15. }
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Solution to The Above So the Message No Longer Appears

view plain

```
01. // Another solution is to not require
02. // the association to be immutable,
03. // just the attributes. But this may
04. // not match the intended semantics.
05. class X {
06.    immutable Integer x;
07. }
08.
09. class Y {
10.     immutable String s;
11.     0..1 -- 0..1 X;
12. }
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E014 Immutable Subclass of Mutable

Umple semantic error reported when an attempt is made to create an immutable subclass of a mutable superclass

Immutability means that no aspect of the state (attribute, association, state machine) can change in a class once constructed. However if a superclass has mutable elements, they are inherited as mutable. As a result it is not allowed for an immutable class to be a subclass of a mutable class.

Note that this will occur if the entire subclass is being declared as immutable and if any element of the superclass is not immutable. It is possible to have mixes of immutable elements and mutable elements in any given class. In the example below, one solution would be to declare attribute b as immutable and remove the class-level immutable status. Class X14immut would be mutable because it would inherit mutable attribute a.

Example

view plain

```
01. // This example generates message 14
02. class B {
03.   a;
04. }
05. 
06. class X14immut {
07.   isA B;
08.   immutable;
09.   b;
10. }
11. 
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W015 Immutable Class State Machine

Umple semantic warning reported when an immutable class is defined as having a state machine.

By definition, an immutable class can't change state, so it can't have a state machine. Any state machine defined is ignored.

Example

view plain

```
01. // The following example shows how to
02. // generate this warning.
03. // Removing the immutable keyword will
04. // solve the problem.
05. class X {
06.   immutable;
07.   Integer y;
08.   sm {
09.     S1 {
10.     }
11.   }
12. }
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E016 Non Immutable Superclass

Umple semantic error reported when an immutable class is defined as having a superclass that is non-immutable.

In an inheritance hierarchy, all classes must be immutable, or all non-immutable, otherwise the substitutability principle would broken.

Example

view plain

```
01. // The following example generates the error message
02. // A solution is to make class X immutable
03. class X {
04.   a;
05. }
06. class Y {
07.   immutable;
08.   isA X;
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E017 Non Immutable Bidirectionality

Umple semantic error reported when a non-directed association is declared to be immutable, or appears in an immutable class.

Allowing two-way associations to be immutable would be impossible since it would require change to one object when it is linked to the other, and that would break the concept of immutability.

Example

view plain

```
01. // The following example generates the error message
02. // Change the -- to -> to solve the problem
03. class Y {
04.   immutable;
05.   1 -- * Z;
06.   isA X;
07. }
08.
09. class Z {
10.   immutable;
11.   y;
12. }
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E018 Reflexive Immutable

Umple semantic error reported when an immutable reflexive association is declared as having 1 at the other end

This is a special case of the general rule that reflexive associations must have upper bounds greater than zero.

Example

view plain

```
01. // The following example generates the error message
02. class X {
03.     immutable;
04.     0..1 -> 1 X other;
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E019 Duplicate Association

Umple semantic error reported when a class is given two associations with the same name.

Associations between the same classes must be given different names. This error can occur when two associations have the same role name; the solution is to change one of the role names. The error can also occur when two associations are created without any role name at all. In that case the default name is generated from the associated class. The solution is to add a role name to one of the associations.

The first example below is a simple case where there are identical associations with no role name. The second example shows how to solve this.

The third example shows that error 19 can also occur with association classes. The solution to this can be found in the manual page for association classes.

Note that if warning W093 is being raised, resolve that issue first. This will resolve the error in some cases.

Example

view plain

```
01. // The following example shows how
02. // to generate error 19.
03. // The solution is to at least one
04. // role name one or both of the Y ends
05. // and at least one role name on one
06. // of both of the X ends.
07. class X {
08. }
09.
10. class Y {
11.   0..1 -> * X;
12.   0..1 -> * X;
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // Using role names rn1 and rn2 to
02. // avoid error 19
03. class X {
04. }
05.
06. class Y {
07.   1 -- * X;
08.   1 rn2 -- * X rn1;
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // Example of an association class
02. // definition that generates error 19
03. // The solution is to add a role name
04. // such as menteeAssignments between
05. // the first * and Member
06.
07. class Member {
08.   name;
09. }
10.
11. associationClass Assignment {
12.   Date dateEstablished;
13.   * Member mentor;
14.   * Member mentee;
15. }
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E020 Interface Association Not One Way

Umple semantic error reported when an association to an interface is declared to be bidirectional or pointing from the interface.

An association can be declared from a class to an interface, but if this is done, it must be one way using the '->' notation. The reason for this is that no concrete methods can be generated in an interface.

Example

view plain

```
01. // This example generates the message
02. class X20classintfarrow {
03.   1 -- * IA;
04. }
05.
06. interface IA {}
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // This example generates the message
02. class X20classintfarrow {
03.   1 <- * IA;
04. }
05.
06. interface IA {}
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E021 Invalid Reflexive Association

Umple syntactic error reported when a class has a reflexive association (an association with both ends the same) but there is neither a role name (indicating it is asymmetric), nor a 'self' keyword (indicating it is symmetric).

Code cannot be generated for asymmetric reflexive associations unless there is a role name at at least one end. This ensures the generated API has distinct words for each end.

For more information about reflexive associations, see this page.

Example

view plain

```
01. // The following example shows
02. // how to generate this error.
03. class X {
04.    0..1 -- 0..1 X;
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following example shows how
02. // to fix the error if this is an
03. // asymmetric association.
04. class X {
05.    0..1 -- 0..1 X right;
06. }
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following example shows how to fix
02. // the error if this is a symmetric association.
03. class X {
04.     0..1 self right;
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E022 Duplicate Attribute

Umple semantic error reported when a class is given two attributes with the same name.

Attributes in a class must have distinct names. The solution to this error is to rename or delete one of the attributes. This error is particularly prone to appear when a class is composed of two or more files using Umple's 'mixin' capability; the attribute should be defined in one file or the other, but not both.

Example

view plain

```
01. // The following example shows how
02. // to generate this error.
03. class X {
04.   a;
05.   a;
06. }
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E023 Attribute Association Name Clash

Umple semantic error reported when an association has the same name as an attribute or vice versa.

An association generates a variable and methods that would clash with an attribute of the same name. If a role name is used, then it must be distinct from all attributes. If no role name is used then the class name (with first character converted to lower case) at the 'other end' of the association must be distinct from all attributes.

Ths problem can sometimes be hard to notice if an association is declared 'backwards' in the 'other' class, or independently of either class. It is especially tricky if the clashing association or attribute is in a separate mixin.

Example

```
view plain

// This example generates error 23
// because an attribute and role name share a name
class X23atdupassoc1 {
  a;
  1 -- 0..1 Another a;
}

class Another {}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

```
view plain

// This generates error 23 because
// attribute b clashes with
// class name B in the association
class X23atdupassoc2 {
  b;
  1 -- 0..1 B;
}

```

```
09. class B {}
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E024 Sort Key Not of Umple Type

Umple semantic error reported when a sort key for a sorted association does not use one of the builtin Umple types

The sort key for an sorted association must have one of the simple types Integer, Short, Long, Double, Float or String. This allows sensible code to be generated for comparisons. The set of allowed types may be expanded in the future.

Example

view plain

```
01. // This example generates the error message
02. class X {
03.   1 -- * Y sorted {endDate};
04. }
05.
06. class Y {
07.    Date endDate;
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E025 Sort Key Not Found

Umple semantic error reported when a sort key for a sorted association is not an attribute of the destination class

The sort key of a sorted association must be one of the attributes of the class at the other end of the association.

Example

view plain

```
01. // This example generates the error message
02. class X {
03.    * -- 1 Y sorted {nom};
04. }
05.
06. class Y {
07.   name;
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E026 Duplicate Key Item

Umple semantic error reported when a key has a duplicate item

If an item (attribute, association, state machine) appears twice in a key, it is an error, since Umple cannot tell the preferred order of items.

Example

view plain

```
01. // The following example has error 26
02. // since the key has 'a' twice
03. class X26dupiteminkey {
04.   a;
05.   b;
06.   key {a, b, a}
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W027 Key Identifier Incorrect

Umple semantic warning reported when an identifier in a key statement was not found

The list of identifiers in a key statement can include attributes, associations (role name for example) or state machines. This message indicates that the referenced identifier could not be found. This is usually the result of a typographical error. The extraneous identifier is ignored, which is likely to result in incorrect code.

Example

view plain

```
01. // This example generates the message
02. class X27itemkeynotdef {
03.   a;
04.   key {b};
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E028 Constraint Identifier Incorrect

Umple semantic error reported when an identifier (attribute, etc.) in a constraint cannot be found

Currently, constraints can refer to attributes in the current class; in the future they will be able to refer to other model elements. This message indicates that an identifier referenced in the constraint was not found. This is often just due to a typographical error, but it might be due to the current limitations of constraints.

Example

```
view plain

01. // This example generates the message
02. class X28attrconsrnotdef {
03.   Integer a;
04.   [b > 5]
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E029 Constraint Type Mismatch

Umple semantic error reported when there is a type mismatch in a constraint

It is only possible to compare a String to a String, a Boolean to a Boolean, and a number to a number, etc. Violations result in a type mismatch error.

Example

view plain

```
01. // This example generates the message
02. class X29attrtypeconstr {
03.   a;
04.   [a > 5]
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following shows how to avoid the message
02. class X29attrtypeconstr {
03.   Integer a;
04.   [a > 5]
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W030 Redefined Namespace

Umple semantic warning reported when a class is declared in separate places to be in two namespaces

Since a class can only be in one namespace, the last namespace declaration overides earlier ones. This can sometimes be useful, e.g. when creating a mixin to change the default location of a class. However, it is normally a sign of a mistake, hence the warning.

Example

view plain

```
01. // This example generates the message
02. // Namespace b, encountered later, will
03. // be the namespace of the class
04. namespace a;
05.
06. class X30redefnamespace {
07. }
08.
09. namespace b;
10.
11. class X30redefnamespace {
12. }
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W031 Namespace Not Used

Umple semantic warning reported when a namespace is never applied to any class

If you declare a namespace and then declare another before using the first namespace, this warning is issued. The reason for the warning is that this can happen accidentally, for example, if you add a namespace declaration before a comment and then add a different one after a comment, but before the class is declared.

Example

```
view plain

01. // This example generates the message because
02. // namespace n is never used
03. namespace n;
04. namespace m;
05.
06. class X21namespacenotused {}
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E032 Reserved Role Name

Umple reports this error when a specific role name is used

In order to define a reflexive association, it is necessary to specify role names. There is a situation where it is possible to define a reflexive association with one role name. When this happens, Umple uses an implicit name for the second role name which is the plural form of the class name (e.g. if the class name is Computer, plural form would be computers). Therefore, if a user uses plural form of the class name as role name, Umple produces an error.

Example

```
view plain

01. // This example generates the error message
02.
03. class Course {
04.   * -- * Course courses;
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // The following shows how to avoid the error
02.
03. class Course {
04.   * -- * Course course_s ;
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

```
view plain
```

```
// This example generates the error message

class Course {
  * courses -- * Course;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
// The following shows how to avoid the error

class Course {
  * course_s -- * Course;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W033 Missing Superclass

Umple semantic warning issued when a referenced superclass is not found

When you declare a superclass of a class, that class must exist. If the class does not exist, then this warning is issued.

It might be the case that the class is in fact declared in some code that is defined externally. In that case, the existence of the external class should be made known to Umple using the external keyword, such as in the second example.

Example

view plain

```
01. // This example generates the warning
02. // since T is not defined
03. class S {
04.   isA T;
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // This example solves the problem
02. // by defining T as an external class
03. // Umple will not generate any code for T,
04. // but now knows it exists
05. class S {
06.   isA T;
07. }
08. external T {};
09.
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E034 Multiple Inheritance

Umple semantic error generated when multiple inheritance is encountered

Umple does not currently support multiple inheritance, in order to be consistent with code generated in Java, and also to make models simpler.

Example

view plain

```
01. // The following will generate this error
02.
03. class P1 {}
04. class P2 {}
05.
06. class Sub {
07.   isA P1, P2;
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // The following is another way of
02. // formulating the same declarations,
03. // also resulting in the error
04.
05. class P1 {}
06. class P2 {}
07.
08. class Sub {
09.   isA P1;
10.   isA P2;
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // If all but one of the parents is
02. // declared as an Interface, the problem is solved
03.
04. interface P1 {}
05. class P2 {}
06.
07. class Sub {
08.   isA P1;
09.   isA P2;
10. }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W035 Uninitialized Constant

Umple semantic warning generated when a Umple builtin data type constant is not initialized

It makes little sense to have a constant unless it is given a value. If a constant is of one of the built-in Umple data types we will set it to a default value: Integer, Double and Float will be set to zero, String will be set to an empty String, Boolean will be set to false, Date will be set to the date where the code was generated and Time will be set to midnight (00:00:00). However, the warning is issued since forgetting to initialize a constant is a common source of errors.

Example

```
view plain

01. // The following will generate this warning
02.
03. class X {
04.   const String A; //same as const String a = "";
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // The following will resolve the issue
02.
03. class X {
04.   const String A = "Something Interesting";
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W036 Unmanaged Multiplicity

Umple semantic warning generated when a directed association has a multiplicity with hard constraints

Umple generates code to manage the manipulation of the directed end (with the arrowhead) of a directed association. However there are no inverse references in such a case. In the example below, there are no references from class B to class X. As a result, the multiplicity on the undirected end is purely 'documentation'.

Specifying a hard constraint such as 1 or 2 for the upper or lower bound in such a case may be misleading and is typically incorrect. This error is raised whenever the multiplicity is other than * or 0..1

Example

view plain

```
01. // The following will generate this warnbing
02.
03. class X {
04.   1 -> * B first;
05.   1..3 -> * B second;
06.   0..2 -> * B third;
07. }
08.
09. class B {}
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following solves the problem
02. // by either making the association unndirected
03. // or making the bounds have soft constraints
04.
05. namespace W036UnmanagedMultiplicity2;
06.
07. class X {
08.   0..1 -> * B first;
```

```
09.   * -> * B second;
10.   0..2 -- * B third;
11. }
12.
13. class B {}
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E037 Uninitialized Constant Object

Umple semantic error generated when an Object constant is not initialized

It makes little sense to have a constant unless it is given a value. Since there is no obvious default value for arbitrary data types, it is an error to declare an uninitialized constant unless it is of one of the builtin Umple data types. (For builtin datatypes this is not a problem since a number can be initialized as zero by default, and a string as empty by default).

Example

```
view plain

01. // This example generates error 37
02. class X {
03.   const Y A;
04. }
05. 
06. class Y {
07.   Integer a;
08. }
09. 
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // This example resolves error 37
02. 
03. class X {
04.   const Y A=new Y(3);
05. }
06. 
07. class Y {
08.   Integer a;
09. }
10. 
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E038 Autogenerated Attribute Name Conflict

Umple semantic error issued when a name conflict occurs between an attribute and an automatically generated variable in the same class

Some attribute generate two variables, one representing the attribute itself, and one representing associated information. In particular an autounique attribute will generate a static variable prefixed with 'next' and an immutable attribute will generate a variable prefixed with 'canSet'. This error message occurs when conflicts with these occurs.

Example

view plain

```
01. //The error is generated because of
02. //the name conflict between attr
03. //and the attribute canSetAttr,
04. //which is automatically generated
05. //for an immutable attribute
06.
07. //The same occurs for the attr2
08. //and nextAttr2
09. class A {
10.   immutable attr;
11.   canSetAttr;
12.
13.   autounique attr2;
14.   nextAttr2;
15. }
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The error can be fixed by
02. //modifying the names of the
03. //attributes
04. class A {
05.   immutable attr;
```

```
06.   otherAttr;
07.
08.   autounique attr2;
09.   otherAttr2;
10. }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W039 Missing Interface

Umple semantic warning issued when a referenced interface is not found

If an interface extends a second interface, that second interface must exist in the system. It can also be declared as external to the class, if it has been defined externally to the code.

Example

view plain

```
01. //Since B is not defined,
02. //the warning will be generated
03. interface A {
04.   isA B;
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The problem can be solved by
02. //defining B
03. interface A {
04.   isA B;
05. }
06.
07. interface B {}
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E040 Singleton Has Subclasses

Umple semantic error generated when a singleton class has subclasses

Singleton is a software pattern to allow only one instance of a class. A singleton class has a private constructor and cannot be inherited.

Example

view plain

```
01. // In this example a singleton class
02. // has subclasses and it generates an error
03. class Airplane
04. {
05.   singleton;
06. }
07. class F16
08. {
09.     isA Airplane;
10. }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //I this example the singleton class
02. // does not have a subclass and is correct
03. class Airplane
04. {
05.
06. }
07. class TheAirplane
08. {
09.   singleton;
10. }
11. class F16
12. {
13.     isA Airplane;
14. }
```

15.

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E041 Non Void Method Declared as Queued

Umple semantic error generated when a class contains a non-void method declared as queued

The queued keyword cannot be used on methods with a non-void return type. Declare the method as 'queued void' or 'void'.

Example

view plain

```
01. //Causes the error due to the return type of
02. //method1 being Integer
03. class A
04. {
05.   queued Integer method1() { /* implementation */ }
06. }
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //No error is raised, as both methods
02. //have a void return type
03. class A
04. {
05.   queued method1() { /* implementation */ }
06.   queued void method2() { /* implementation */ }
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W042 Namespace Cannot Be Default

Umple warning reported when an entity cannot be in the default namespace

If you declare an entity in the default namespace and then declare another entity that extends, implements, or has an association with the first entity, this warning is issued. The reason for the warning is that if two related entities are in different namespaces, import will be generated for those entities, and many programming laguages do not support import from the default namespace. Therefore, the namespace for entities in the default namespace will be changed.

Example

view plain

```
01. // Classes A, B and C are in the
02. // default namespace. Class D is in
03. // namespace m. All these classes are
04. // linked to each other via associations,
05. // therefore, and because they are not all
06. // in the same namespace, an import text
07. // will be generated. However, many programming
08. // languages do not support import from
09. // the default namespace so the namespace of
10. // C will become m because it is linked to
11. // D, and the same goes for B because C is
12. // now in namespace m, and then A is placed
13. // in m because B is now in m.
14.
15. class A{}
16. class B{*--*A;}
17. class C{*--* B;}
18. namespace m;
19. class D{*--* C;}
20. namespace -;
21.
22.
23. // The same issue occurs when an
24. // entity in a non-default namespace
25. // extends or implements an entity
26. // in the default namespace
27.
28. interface E{}
29. interface F{isA E;}
```

```
30. interface J{isA F;}
31. namespace n;
32. interface K{isA J;}
33.
34.
35. // Here an import text will be generated for D in X
36. // because D is in a different namespace
37. // This feature is not yet supported for interfaces
38. // and the import text will not be generated for K in Y
39.
40. namespace p;
41. class X{isA D;}
42. // interface Y{isA K;}
43.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W044 Attribute Duplicated in Superclass

Umple semantic warning reported when an attribute in a class is a duplicate of an attribute in a superclass and their types differ.

The purpose of this warning is to alert the developer that they have to resolve the issue by changing the name of the duplicated attribute in either the superclass or subclass, or else remove one of them entirely, as you should not define an attribute twice with a different type.

Example

```
view plain

// This example generates the warning message
class A {
  Integer attr;
}

class B {
  isA A;
  attr;
}


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

//This example solves the problem
class A {
  Integer attr;
}

class B {
  isA A;
  differentAttr;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W045 Initialized Value in Key

Umple semantic warning reported when an attribute in the key is given an initial value

If an attribute is given an initial value and is part of a key, then there is potential for keys of all instances of a class to be the same, meaning that they would be treated as equal and would have the same hash value. It is unlikely that a developer intends for this to be the case, so a warning is thrown to inform them of this behaviour.

Example

view plain

```
01. // This example generates the message
02. class X {
03.   Integer z = 1;
04.   key { z }
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following shows how to avoid the message
02. class X {
03.   Integer z;
04.   key { z  }
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W046 Attribute Has Template Type

Umple syntactic warning reported when an attribute has a template type, characterized by brackets "<...>".

Template types do not follow Umple modelling conventions. Umple encourages users to take full advantage of associations in their modelling, which in almost all contexts replace the need for template types. Multivalued attributes can also be used.

Example

```
view plain
01. // This example generates the message
02.
03. class A {
04.   depend java.util.List;
05.   List<OtherClassA> otherClassGroup;
06. }
07. class OtherClassA {}
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain
01. // The following shows how to avoid
02. // the message using a multivalued attribute
03. class B {
04.   OtherClassB[] otherClassGroup;
05. }
06. class OtherClassB {}
07.
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following shows how to avoid
02. // the message using an association
03. class C {
04.   0..1 -- * OtherClassC;
05. }
06. class OtherClassC {}
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W047 Empty Key Statement

Umple semantic warning reported when a key statement has no elements inside of it

An empty key statement has no meaning, but Umple will detect the key statement and generate methods associated with having a key. This might lead the developer to think that they have defined a key, when in fact the generated methods cannot differentiate between instances of the class in question. The warning is shown to notify the developer of the potential mistake.

Example

```
view plain

01. // This example generates the message
02. class A {
03.   id;
04.   key { }
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // The following shows how to avoid the message.
02. class A {
03.   id;
04.   key { id }
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E048 Attribute Generates Duplicate Method

Umple semantic error issued when an attribute generates a duplicate method

An attribute may autogenerate multiple methods, which can lead to duplicates if the methods have already been generated. Changing the name of the attribute causing the duplicate can fix the error.

Example

```
view plain

01. //The error will be issued, attr
02. //and Attr both generating setAttr
03. //and getAttr causing a name conflict
04. class A {
05.   attr;
06.   Attr;
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. //The following example is a
02. //way to fix the error
03. class A {
04.   attr;
05.   attr2;
06. }
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W049 Duplicated Method Name

Umple semantic warning reported when two methods have the same names and return types

You cannot have two methods that have the same names and return types. The warning is shown to notify the developer of the potential mistake.

Example

```
view plain

01. // This example generates the message
02. class A{
03.   String test1(){return("hello world");}
04.   String test1(){return("dlrow olleh");}
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // The following shows how to avoid the message.
02. class A{
03.   void test1() Java {}
04.   void test2() Java {}
05.
06.   void test1() Python { ''' code here ''' }
07.   void test2() Python { ''' code here ''' }
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W050 Target State Not Found

Umple semantic warning reported when the target state of a transition is not listed.

An end state (with no outgoing transitions) is generated. The purpose of the warning is to alert the developer that they may have a typographical error in the target state name.

Example

view plain

```
01. // This example generates the error message
02. class X {
03.   sm {
04.     s1 {
05.       e1 -> s2;
06.     }
07.   }
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // This example solves the problem
02. class X {
03.   sm {
04.     s1 {
05.       e1 -> s2;
06.     }
07.     s2 {}
08.   }
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E051 Event Parameters Must Match

Umple semantic error reported when events with the same name have different parameters

When parameters are specified on a state machine event, all transitions with the same event name must have identical parameters, both in name and type. The reason for this is that there is only one event method generated.

Example

```
view plain

01. // This example generates the error message
02. class X {
03.   sm {
04.     s1 {
05.       e1(String s) -> s2;
06.     }
07.     s2 {
08.       e1(Float f) -> s2;
09.     }
10.   }
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // This example solves the problem
02. class X {
03.   sm {
04.     s1 {
05.       e1(String s) -> s2;
06.     }
07.     s2 {
08.       e1(String s) -> s2;
09.     }
10.   }
11. }
```

12.

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E052 State Machine Name Clash

Umple semantic error reported when a state machine name matches the name of another element such as an association or attribute

Since, in effect, a state machine defines a special kind of attribute (whose value is enumerated as one of the states and is controlled by events) it is not allowed to have an attribute or association with the same name as a state machine.

Example

view plain

```
01. // This example generates the message
02. // because the state machine a
03. // clashes with the attribute a
04. class X52assattnamestatemachine1 {
05.   a;
06.   a {
07.     s1 {}
08.     s2 {}
09.   }
10. }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The sample generates the message
02. // because the state machine a
03. // clashes with the association role name a
04. class X52assattnamestatemachine2 {
05.   1 -- 0..1 B a;
06.   a {
07.     s1 {}
08.     s2 {}
09.   }
10. }
11. class B {}
```

12.

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E053 No Concurrency At Top Level

Umple semantic error reported when concurrent substates are placed at the top level of a state machine

A state machine must be in one top level state at all times. Concurrent substates can exist in any substate at any level, but not directly in a top level state. The second example below shows how this can be handled simply by adding a level of nesting.

If the intent is simply to create concurrent do activities, an alternative is to use Umple's active objects notation.

Example

view plain

```
01. // This example generates the message
02. class X53conctoplevel {
03.   sm {
04.     s1 {
05.       do {System.out.println(
06.         "Reached s1");}
07.     }
08.     ||
09.     s2
10.     {
11.       do {System.out.println(
12.         "Reached s1");}
13.     }
14.   }
15. }
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following shows how to avoid the message
02. class X53conctoplevel {
03.   sm {
04.     s0 {
```

```
05.       s1 {
06.         do {System.out.println(
07.           "Reached s1");}
08.       }
09.       ||
10.       s2
11.       {
12.         do {System.out.println(
13.           "Reached s1");}
14.       }
15.     }
16.   }
17. }
18.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W054 Duplicate Events

Umple warning generated when there is a duplicate unguarded event that will never be reached

In the example below, the second transition e from s1 can never be triggered because when e occurs the first transition is taken.

Example

view plain

```
01. class X {
02.   sm {
03.     s1 {
04.       e-> s2;
05.       e-> s3;
06.     }
07.     s2 {}
08.     s3 {}
09.   }
10. }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W055 Duplicate Events Within Substates

Umple warning generated when there is a duplicate unguarded event that will never be reached, in the situation where there is a superstate and a substate with the same event

In the example below, the event e from the s1 superstate and the event e in the s1a substate are ambiguous. Current semantics is that the event in the superstate takes precedence, but this semantics may be changed.

Example

view plain

```
01. class X {
02.   sm {
03.     s1 {
04.       e-> s2;
05.       s1a {
06.         e-> s3;
07.       }
08.     }
09.     s2 {}
10.     s3 {}
11.   }
12. }
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W056 No Events in Queued State Machine

Umple semantic warning issued when a queued state machine does not have events to be queued

A queued state machine must have events to be queued, or this warning will be issued.

Example

view plain

```
01. //The warning will be generated
02. //as there are no events to be
03. //queued in the state machine sm
04. class A {
05.   queued sm {
06.     s0 {
07.     }
08.     s1 {
09.     }
10.   }
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The following avoids the warning
02. //by adding an event to the state machine
03. class A {
04.   queued sm {
05.     s0 {
06.       e -> s1;
07.     }
08.     s1 {
09.     }
10.   }
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W057 No Events in Pooled State Machine

Umple semantic warning issued when a pooled state machine does not have events to be pooled

A pooled state machine must have events to be pooled, or this warning will be issued.

Example

view plain

```
01. //The warning will be generated
02. //as there are no events to be
03. //pooled in the state machine sm
04. class A {
05.   pooled sm {
06.     s0 {
07.     }
08.     s1 {
09.     }
10.   }
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The following avoids the warning
02. //by adding an event to the state machine
03. class A {
04.   pooled sm {
05.     s0 {
06.       e -> s1;
07.     }
08.     s1 {
09.     }
10.   }
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E058 Regular Queued and Pooled State Machines in Class

Umple semantic error reported when a class contains one or more queued state machine, pooled state machine and regular state machine

An Umple class cannot contain in the same class a queued state machine, a pooled state machine and/or a regular state machine.
All the state machines in a given class must be of the same type (pooled, queued, regular), or could be distributed in different classes.

Example

view plain

```
01. //The error will be generated as A
02. //contains a queued, pooled and
03. //regular state machine at once
04. class A {
05.   pooled sm1 {
06.     s0 {
07.       e -> s1;
08.     }
09.     s1 {
10.     }
11.   }
12.
13.   queued sm2 {
14.     s0 {
15.       e -> s1;
16.     }
17.     s1 {
18.     }
19.   }
20.
21.   sm3 {
22.   }
23. }
24.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The error can be fixed by having each
02. //state machine in a different class
03. class A {
04.   pooled sm1 {
05.     s0 {
06.       e -> s1;
07.     }
08.     s1 {
09.     }
10.   }
11. }
12. 
13. class B {
14.   queued sm2 {
15.     s0 {
16.       e -> s1;
17.     }
18.     s1 {
19.     }
20.   }
21. }
22. 
23. class C {
24.   sm3 {
25.   }
26. }
27. 
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E059 Queued and Pooled State Machines in Class

Umple semantic error reported when a class contains one or more queued state machine and pooled state machine

An Umple class cannot contain both a queued state machine and a pooled state machine.
The state machines must be of the same type (all pooled, or all queued) or else could be distributed in more than one class.

Example

view plain

```
01. //The error will be generated as
02. //class A contains both a queued
03. //and a pooled state machine
04. class A {
05.   pooled sm1 {
06.     s0 {
07.       e -> s1;
08.     }
09.     s1 {
10.     }
11.   }
12.
13.   queued sm2 {
14.     s0 {
15.       e -> s1;
16.     }
17.     s1 {
18.     }
19.   }
20. }
21.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The error can be fixed by having each
02. //state machine in a different class
```

```
03. class A {
04.   pooled sm1 {
05.     s0 {
06.       e -> s1;
07.     }
08.     s1 {
09.     }
10.   }
11. }
12.
13. class B {
14.   queued sm2 {
15.     s0 {
16.       e -> s1;
17.     }
18.     s1 {
19.     }
20.   }
21. }
22.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E060 Pooled and Regular State Machines in Class

Umple semantic error reported when a class contains one or more pooled state machine and regular state machine

An Umple class cannot contain both a pooled state machine and a regular state machine.
The state machines must be of the same type (e.g. both pooled), or else distributed in more than one class.

Example

view plain

```
01. //The error will be generated as
02. //class A contains both a regular
03. //and a pooled state machine
04. class A {
05.   pooled sm1 {
06.     s0 {
07.       e -> s1;
08.     }
09.     s1 {
10.     }
11.   }
12.
13.   sm2 {
14.     s0 {
15.       e -> s1;
16.     }
17.     s1 {
18.     }
19.   }
20. }
21.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The error can be fixed by having each
02. //state machine in a different class
```

```
03. class A {
04.   pooled sm1 {
05.     s0 {
06.       e -> s1;
07.     }
08.     s1 {
09.     }
10.   }
11. }
12.
13. class B {
14.   sm2 {
15.     s0 {
16.       e -> s1;
17.     }
18.     s1 {
19.     }
20.   }
21. }
22.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E061 Queued and Regular State Machines in Class

Umple semantic error reported when a class contains one or more queued state machine and regular state machine

An Umple class cannot contain both a queued state machine and a regular state machine.
The state machines must be of the same type (e.g. both queued), or else distributed in more than one class.

Example

view plain

```
01. //The error will be generated as
02. //class A contains both a regular
03. //and a queued state machine
04. class A {
05.   queued sm1 {
06.     s0 {
07.       e -> s1;
08.     }
09.     s1 {
10.     }
11.   }
12.
13.   sm2 {
14.     s0 {
15.       e -> s1;
16.     }
17.     s1 {
18.     }
19.   }
20. }
21.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The error can be fixed by having each
```

```
02. //state machine in a different class
03. class A {
04.   queued sm1 {
05.     s0 {
06.       e -> s1;
07.     }
08.     s1 {
09.     }
10.   }
11. }
12.
13. class B {
14.   sm2 {
15.     s0 {
16.       e -> s1;
17.     }
18.     s1 {
19.     }
20.   }
21. }
22.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W062 Unspecified Event in Pooled State Machine

Umple semantic warning issued when the event "unspecified" is used in a pooled state machine

A pooled state machine cannot use the "unspecified" event in a transition. This is because the unspecified functionality is designed to handle events that occur unexpectedly, which only makes sense in a regular and pooled state machine. Pooled state machines instead just leave such events on the queue, and process events further back in the queue. In a pooled state machine, an event with the "unspecified" label is treated as a regular event and will be pooled.
To catch and immediately process unspecified events, a queued or regular state machine could be used instead.

Example

```
view plain

01. //The warning is generated for
02. //sm due to the use of the unspecified
03. //event
04. class A {
05.   pooled sm {
06.     s0 {
07.       unspecified -> s1;
08.     }
09.     s1 {
10.     }
11.   }
12. }
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. //The warning can be avoided by using
02. //a queued state machine to be able
03. //to use the unspecified event
04. class A {
05.   queued sm {
```

```
06.     s0 {
07.       unspecified -> s1;
08.     }
09.     s1 {
10.     }
11.   }
12. }
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E063 Reserved Name History State

Umple semantic error reported when a state uses a name reserved for history or deep history states

Certain state names in Umple are reserved for the implementation of history and deep history states.
An error is raised when using a reserved name such as H or HStar as a state name.

Example

view plain

```
01. //The error occurs as state machine
02. //sm uses the reserved name H as a
03. //state name
04. class A {
05.   sm {
06.     H {
07.     }
08.   }
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The error can be avoided by
02. //using a non reserved state
03. //name
04. class A {
05.   sm {
06.     s1 {
07.     }
08.   }
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E064 Non Existent History State

Umple semantic error reported when a history state is declared in a non existent state

A transition to the history of a state must be declared on an existing state for Umple to be able to complete the transition to its stored history state. The error will be raised if the state is not found in the state machine.

Example

view plain

```
01. //The following generates the error,
02. //OtherState not existing within sm
03. class A {
04.   sm {
05.     s1 {
06.       e -> OtherState.H;
07.     }
08.   }
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The error can be resolved by
02. //declaring OtherState in sm
03. class A {
04.   sm {
05.     s1 {
06.       e -> OtherState.H;
07.     }
08.     OtherState {
09.       OtherStateA {
10.       }
11.       /* ... other substates */
12.     }
13.   }
14. }
```

15.

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E065 No Substates History State

Umple semantic error reported when a history state is declared at a state with no substates

A transition to a history state can only be declared on a state with substates, as a state with no substates has no history to be managed.

Example

view plain

```
01. //The error is generated because
02. //s2 does not contain a substate
03. //and therefore no history to
04. //be managed
05. class A {
06.   sm {
07.     s1 {
08.       e -> s2.H;
09.     }
10.     s2 {
11.
12.     }
13.   }
14. }
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The following does not generate
02. //the error as s2 contains substates
03. //and can keep the last substate in
04. //memory
05. class A {
06.   sm {
07.     s1 {
08.       e -> s2.H;
09.     }
10.     s2 {
```

```
11.       s2a {
12.       }
13.       s2b {
14.       }
15.       /* ... more substates */
16.     }
17.   }
18. }
19.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W066 Transition Multiple Possible Destinations

Umple semantic warning issued when a transition has multiple possible destinations

If a transition's destination state name has multiple occurrences in the state machine, this warning will be issued, as it is possible the developer means to reference a specific substate of the same name. The dot notation should be used to clarify which substate is intended to be the destination state.
The transition destination is by default the highest level state in the state machine. In the case of multiple states at the same level using the same state name, Umple currently assumes the state that has been defined first is being referenced.

Example

view plain

```
01. //The following generates the warning
02. //as s1 could designate both s1 and s2.s1
03. //By default, it is s1
04. class A {
05.   sm {
06.     s1 {
07.       e -> s1;
08.     }
09.
10.     s2 {
11.       s1 {
12.       }
13.     }
14.   }
15. }
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The warning does not occur
02. //by removing the ambiguity
```

```
03. class A {
04.   sm {
05.     s1 {
06.       e -> s2.s1;
07.     }
08.
09.     s2 {
10.       s1 {
11.       }
12.     }
13.   }
14. }
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W067 Non Reachable State

Umple semantic warning issued when a state is non-reachable

A state that is never the destination state of a transition in the state machine or is not the initial state of the state machine is considered non-reachable by the Umple compiler. The generated methods will not assign this state to the machine.
The warning can be avoided by removing the non-reachable state or by adding an appropriate transition to the state machine.

Example

view plain

```
01. //The warning is generated for
02. //s3, as no transition leads to it
03. class A {
04.   sm {
05.     s1 {
06.       e -> s2;
07.     }
08.     s2 {
09.     }
10.     s3 {
11.     }
12.   }
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The warning can be avoided
02. //by adding a transition with s3
03. //as destination state
04. class A {
05.   sm {
06.     s1 {
07.       e -> s2;
08.     }
```

```
09.     s2 {
10.       e2 -> s3;
11.     }
12.     s3 {
13.     }
14.   }
15. }
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W068 Destination State Not Found

Umple semantic warning issued when the destination state of a transition cannot be found

When using dot notation to define the destination state of a transition in a nested state machine (e.g. sm1.s2), then the destination state must exist, otherwise the transition will be ignored. The warning is most likely caused by a typographical error. (Note that when not using dot notation, behaviour is a bit different, and a new end state will be created by default with warning 50).
The warning can be resolved by correcting the destination state name or by creating the state.

Example

view plain

```
01. //The warning is generated due
02. //to the transition destination
03. //of e -> s3.s2a not being defined in sm
04. class A {
05.   sm {
06.     s1 {
07.       e -> s3.s2a;
08.     }
09.     s2 {
10.       s2a {
11.       }
12.     }
13.   }
14. }
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The following example does
02. //not generate the warning
03. //as s2.s2a is found in sm
04. class A {
```

```
  sm {
    s1 {
      e -> s2.s2a;
    }
    s2 {
      s2a {
      }
    }
  }
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W069 Auto Transition Conflict

Umple semantic warning issued when an auto-transition has a conflict with a previously defined transition

An auto-transition in a state that is declared after another auto-transition will be ignored, and only the first auto-transition will be executed. No warning is issued if the destination state of both auto-transitions is the same.

Example

view plain

```
01. //The warning is generated due
02. //to the auto-transition to s2
03. //conflicting with the auto-
04. //transition to s3
05. class A {
06.   sm {
07.     s1 {
08.       entry /{doSomething();}
09.       -> s2;
10.
11.       do {doSomethingElse();}
12.       -> s3;
13.     }
14.     s2 {
15.     }
16.     s3 {
17.     }
18.   }
19.   void doSomething() {}
20.   void doSomethingElse() {}
21. }
22.
23.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The warning is avoided here
02. //by keeping only one auto-
03. //transition
04. class A {
05.   sm {
06.     s1 {
07.       entry /{doSomething();}
08.       do {doSomethingElse();}
09.       -> s2;
10.     }
11.     s2 {
12.     }
13.   }
14.   void doSomething() {}
15.   void doSomethingElse() {}
16. }
17.
18.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W071 Duplicate Method Different Type

Umple sematic warning reported when two methods have the same names but different types

In some programming languages like Java, you cannot have the same method name for multiple methods even if the return types are different. The warning is shown to notify the developer of the potential mistake.

Example

view plain

```
01. // This example generates the message
02. class A{
03.   String test1(){return("hello world");}
04.   int test1(){return(1);}
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following shows how to avoid the message.
02. class A{
03.   void test2(){}
04.   Integer test1(){}
05. }
06. // @@@skipcompile no return from test1
07.
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W072 Refactored Final State

Umple sematic warning when do activities, exit actions, outgoing transitions, and/or nested state machines are removed by the compiler from final states.

In Umple, final states are allowed to be empty, or they can contain entry actions.

Example

view plain

```
01. // The following shows how to generate the warning
02.
03. class InvalidFinalState {
04.   status{
05.     on{
06.       turnoff -> off;
07.       powerOff-> FINAL;
08.     }
09.
10.     off{
11.       turnOn -> on;
12.     }
13.
14.     final FINAL{
15.       entry/{entry();}
16.       do{exe();}
17.       reboot -> on;
18.       nestedSm {
19.         s1 {
20.           -> s2;
21.         }
22.         s2 {
23.
24.         }
25.       }
26.       exit/{exit();}
27.     }
28.   }
29. }
30. // @@@skipcompile no method entry written for line 15
31.
```

Load the above code into classic UmpleOnline

Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following shows how to avoid the warning
02.
03. class X {
04.   status{
05.     on{
06.       turnoff -> off;
07.       powerOff-> FINAL;
08.     }
09.     off{
10.       turnOn -> on;
11.     }
12.     final FINAL{
13.       entry/{entry();}
14.     }
15.   }
16. }
17. // @@@skipcompile no method entry written line 13
18.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following shows how to avoid the warning
02.
03. class X {
04.   status{
05.     on{
06.       turnoff -> off;
07.       powerOff-> FINAL;
08.     }
09.     off{
10.       turnOn -> on;
11.     }
```

```
12.     final FINAL{
13.
14.     }
15.   }
16. }
17.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E073 Duplicate Parallel State Machine Name

Umple semantic error reported when parallel state machines in the same composite state have the same name.

In Umple, parallel state machines within the same composite state must have different names.

Example

view plain

```
01. // The following example generates the error
02. // "s1" has two parallel state machines that are named "t1"
03.
04. class X {
05.   sm{
06.     s1{
07.       t1{
08.         goT2-> t2;
09.       }
10.       t2{ }
11.       ||
12.       t1{
13.         goT4-> t4;
14.       }
15.       t4{}
16.     }
17.   }
18. }
19.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following example shows how to avoid the error
02.
03. class X {
04.   sm{
05.     s1{
06.       t1{
```

```
      goT2-> t2;
    }
    t2{ }
    ||
    t3{
      goT4-> t4;
    }
    t4{}
  }
 }
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E074 User Defined State Cannot be Named Final

Umple semantic error reported when a user-defined state is named 'Final' keyword

In Umple, 'Final' is a reserved keyword. It is used to define the final state for the top-level state machine.

Example

```
view plain

01. // The following example generates the error
02. // "s1" has a state named "Final"
03.
04. class X {
05.   sm {
06.     s1 {
07.       Final {}
08.     }
09.   }
10. }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // The following example does not generate the error
02.
03. class X {
04.   sm {
05.     s1 {
06.       goToFinal-> Final;
07.     }
08.   }
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E075 UserDefinedStateMachineCannotBeNamedTimer

Umple error reported when a user-defined state machine is named 'timer'

In Umple, 'timer' cannot be used as a state machine name. This is because generated code for 'after' statements creates timers and there would be a naming conflict in the generated code. To fix the problem, give the state machine a slightly different name,

Example

view plain

```
01. class X {
02.   timer {
03.     s1 {
04.       after(1) -> s2;
05.     }
06.     s2 {
07.     }
08.   }
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. class X {
02.   sm_timer {
03.     s1 {
04.       after(1) -> s2;
05.     }
06.     s2 {
07.     }
08.   }
09. }
10.
```

Load the above code into classic UmpleOnline

Load the above code into UmpleOnline V2

E080 Invalid Abstract Class Association

Umple semantic error raised when an association with an abstract class forces instantiation of an object

An association with an abstract class must either be a directed association or must have a multiplicity with lower bound of zero at both ends. This is necessary to prevent situations where it would be needed to instantiate the abstract class (which is never allowed).

Example

view plain

```
01. //Class A contains an association
02. //with abstract class B, causing
03. //the error by using mandatory
04. //multiplicity
05. class A {
06.   1 -- 1 B;
07. }
08.
09. abstract class B{
10.   abstract;
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The following association does not
02. //cause the error
03. class A {
04.   0..1 -- 0..1 B;
05. }
06.
07. abstract class B{
08.   abstract;
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E081 Invalid Multivalued Attribute Assignment

Umple semantic error raised when initializing multivalued attributes with invalid values

When indicating initial values for a multivalued attribute, an error is raised if the values to assign cannot be parsed properly.
The error might be caused by a typographical mistake.

Example

```
view plain

01. //The attribute attr cannot be initialized
02. //as the values are not correct
03. class A {
04.   String[] attr = {invalid.values};
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. //attr can now be initialized
02. class A {
03.   String[] attr = {"valid", "values"};
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W082 Conflicting Method Modifiers

Umple semantic warning when declaring state-dependent methods with different method modifiers.

This warning is shown when conflicting method modifiers are provided for state-dependent methods. The example below illustrates this scenario by marking the method 'printState' as both public and private.

Example

view plain

```
01. // The following shows how to generate the warning
02.
03. class ConflictingModifiers {
04.   status{
05.     on{
06.       turnoff -> off;
07.       public void printState() {
08.           System.out.println("on");
09.       }
10.     }
11.
12.     off{
13.       private void printState() {
14.           System.out.println("off");
15.       }
16.     }
17.   }
18. }
19.
20.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E083 Invalid Multivalued Attribute Initialization

Umple syntactic error raised when initializing multivalued attributes with invalid values

When indicating initial values for a multivalued attribute, an error is raised if the values to assign cannot be parsed properly.
The error might be caused by a typographical mistake.

Example

view plain

```
01. // For derived attributes remove list indicator []
02. // For list initializer, there should be a semicolon on the end
03. class Office {
04.   Integer number;
05.   String[] emails = {"abc@umple.org", "def@umple.org"}
06.   Integer [] incomingNumber = {765432, 987654}
07. }
08.
09.
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //emails and incomingNumber can now be initialized
02. class Office {
03.   Integer number;
04.   String[] emails = {"abc@umple.org", "def@umple.org"};
05.   Integer [] incomingNumber = {765432, 987654};
06. }
07.
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W089 Association Rolename Matching Classname

Umple semantic warning issued when assigning a role to an association that matches the class name

The role name of an association is used to explain how a class participates in a relationship. A role name that matches the class name may be unnecessary, since it provides little clarification as to the relationship between the classes.
This warning can be avoided by removing or changing the role name.

Note that in certain cases this could raise syntactic errors such as E019.

Example

```
view plain

01. //This example generates the warning message
02. class Person {
03.   name;
04. }
05.
06. class Intern {
07.   isA Person;
08.   * -- 0..2 Employee employee;
09. }
10.
11. class Employee {
12.   isA Person;
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. //This example shows how to avoid the warning
02. class Person {
03.   name;
04. }
05.
06. class Intern {
```

```
07.   isA Person;
08.   * -- 0..2 Employee supervisor;
09.  }
10.
11.  class Employee {
12.   isA Person;
13.  }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E090 Attribute Name Not Found Constraint

Umple semantic error raised when a class does not contain an attribute of a given name specified in a constraint

Model constraints can be applied to Umple classes to ensure they respect certain properties, as defined in their constraints. This error is raised if a class does not contain an attribute with a name specified in the constraint. The attribute might have been defined in a separate file previously, but no longer exists, causing the error.

Example

```
view plain

01. //Class A does not contain
02. //the appropriate attribute,
03. //the constraint causes the error
04. class A {
05.   [model: A has attribute named attr]
06. }
07. // @@@skipcompile
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. //The class A now contains an
02. //attribute with the appropriate name
03. class A {
04.   attr;
05.   [model: A has attribute named attr]
06. }
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E091 Attribute Of Type Not Found Constraint

Umple semantic error raised when a class does not contain the required attribute of a given type

Model constraints can be applied to Umple classes to ensure they respect certain properties, as defined in their constraints. This error is raised if a class does not contain an attribute of a specified type in the constraint.

Example

view plain

```
01. //The error is raised, A does
02. //not contain an attribute of
03. //type B
04. class A {
05.   [model: A has attribute of B]
06. }
07.
08. class B {
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //A now contains an
02. //attribute of type B
03. class A {
04.   B attr;
05.   [model: A has attribute of B]
06. }
07.
08. class B {
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E092 Class Is Not Subclass Constraint

Umple semantic error raised when a class is not the subclass of a given class as defined by a constraint

Model constraints can be applied to Umple classes to ensure they respect certain properties, as defined in their constraints. This error is raised if a class is not the subclass of a given class as expected from the constraint.

Example

view plain

```
01. //The error occurs as A
02. //does not respect the constraint
03. //Alternative syntax is indicated
04. //in comments
05.
06. class A {
07.   //[model: A child of B]
08.   //[model: A inherits from B]
09.   [model: A isA B]
10. }
11.
12. class B {
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //A respects the constraint
02. //and avoids the error
03.
04. class A {
05.   isA B;
06.   [model: A isA B]
07. }
08.
09. class B {
10. }
```

```
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E093 Class Is Not Superclass Constraint

Umple semantic error raised when a class is not the superclass of a given class as defined by a constraint

Model constraints can be applied to Umple classes to ensure they respect certain properties, as defined in their constraints. This error is raised if a class is not the superclass of a given class as expected from the constraint.

Example

view plain

```
01. //The error is produced, the
02. //constraint is not respected because
03. //A is not a superclass of B
04. //Alternative syntax in comment
05. class A {
06.   //[model: A parent of B]
07.   [model: A superclass B]
08. }
09.
10. class B {
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The constraint is now respected
02. //and the error does not occur
03. class A {
04.   [model: A superclass B]
05. }
06.
07. class B {
08.   isA A;
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E094 No Association Found Constraint

Umple semantic error raised when a class does not contain an association as defined by a constraint

Model constraints can be applied to Umple classes to ensure they respect certain properties, as defined in their constraints. This error is raised if an association required by a constraint is not found in the class. The multiplicity can optionally be specified.

Example

```
view plain

01. //The class A generates the error
02. //by not containing the required
03. //association
04. class A {
05.   [model: A -- B]
06. }
07.
08. class B {
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. //The error is not generated
02. //A contains an association
03. //fulfilling the requirements
04. //of the constraint
05. class A {
06.   0..1 -- *B;
07.   [model: A -- B]
08. }
09.
10. class B {
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E095 Duplicate Enumerations

Umple semantic error reported when duplicate enumerations are detected

In Umple, enumerations must have unique names.

Example

```
view plain

01. // This example causes the error
02. enum Month {x,y,z}
03. enum Month {o,p,q}
04.
05. class A{
06.   Month m;
07.   Month p;
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

```
view plain

01. // This example causes the error
02.
03. class A{
04.   enum Month {x,y,z}
05.   enum Month {o,p,q}
06. }
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // This example does not cause the error.
```

```
02. // The enumeration within the class
03. // is prioritized over the enumeration
04. // defined outside of the class
05.
06. enum Month {x,y,z}
07.
08. class A{
09.   enum Month {o,p,q}
10.   Month m;
11.   Month p;
12. }
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E096 Enumeration Naming Conflict

Umple semantic error reported when an enumeration defined in a class has the same name as the class, or when a model level enumeration has the same name as a class, interface, or trait.

In Umple, class enumerations must not have the same name as the class they are defined in. Model level enumerations must have unique names.

Example

```
view plain

01. // This example generates the error
02. // since the enumeration "X" will be added
03. // to the class "X" because of attribute "t".
04.
05. enum X{ Red, Blue, Green }
06.
07. class X{
08.   X t;
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

```
view plain

01. // This generates the error
02. // because the enumeration "X"
03. // has the same name
04. // as the class it's defined in
05.
06. class X{
07.   enum X { Red, Blue, Green }
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // This example generates the error
02. // since enumeration "Y"
03. // conflicts with interface "Y"
04.
05. interface Y {
06.   name;
07. }
08.
09. class X {
10.  isA Y;
11. }
12.
13. enum Y { Red, Blue, Green }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // This example generates the error
02. // since enumeration "Y" conflicts with trait "Y"
03.
04. class X {
05.  isA Y;
06. }
07.
08. trait Y {
09.   name;
10. }
11.
12. enum Y { Red, Blue, Green }
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // This example does not generate the error
02. enum Z { Orange, Yellow }
03.
04. class X {
05.  enum Y { Red, Blue, Green }
06.  Z attr;
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E097 Enumeration and State Machine Conflict

Umple semantic error reported when an enumeration, and a state machine within the same class have the same name.

In Umple, enumerations and state machines within the same class cannot have the same name.

Example

view plain

```
01. // This example generates the error
02. // because enumeration "Y" has the
03. // same name as state machine "Y"
04.
05. class X {
06.  enum Y { Red, Blue, Green }
07.  Y {
08.   s1 {
09.     goToS2 -> s2;
10.   }
11.   s2 { }
12.  }
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // This example generates the error
02. // because enumeration "Y" will be
03. // added to class "X" since the "attr"
04. // parameter of "showY" is "Y".
05.
06. enum Y { Red, Blue, Green }
07.
08. class X {
09.   showY(Y attr) {
10.     System.out.println(attr); }
11.   Y {
```

```
    s1 {}
   }
 }
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
// This example does not generate the error

class X {
 enum Y { Red, Blue, Green }
 Z {
  s1 {
    goToS2 -> s2;
  }
  s2 { }
 }
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

WE1xx Identifier Invalid

Umple syntactic error or warning reported when an identifier does not have the correct syntax

In Umple, identifiers used for classes, association role names, types, attributes, state machines, states, events and other elements must be alphanumeric. In addition, class and interface names should start with a capital letter (or _), and other elements should start with a lower case letter. The exact error message you received will tell you which identifier has the problem. The examples below will generate these messages.

Example

view plain

```
01. // Class with non-alphanumeric name (100)
02. class @ {}
03.
04.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // Warnings from attribute starting with
02. // upper case letter (131)
03. class X {
04.   Attrib;
05. }
06.
07. class Y {}
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // Attribute name with special characters (130)
02. class X {
03.   na$me;
04. }
05.
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // Attribute name with special characters
02. // that looks like an association (132)
03. // If this is supposed to be an association,
04. // put spaces before and after the -- and *
05. class X {
06.   1--*Y;
07. }
08.
09. class Y {}
10.
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // Attribute type with special characters
02. // that looks like an association (140)
03. // If this is supposed to be an association,
04. // put spaces before and after the --
05. class X {
06.   1--* Y;
07. }
08.
09. class Y {}
10.
```

```
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // State machine name with special
02. // characters (150)
03. class X {
04.   ed%4 {
05.     s1 {}
06.   }
07. }
08.
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // State  name with special characters (152)
02. class X {
03.   ed4 {
04.     s*1 {}
05.   }
06. }
07.
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W102 Enumeration Causes Method Parameter Ambiguity

Umple semantic warning reported when an enumeration makes a method parameter's type ambiguous. This occurs when there is another class that could also be the parameter's type.

Example

view plain

```
01. // This example generates the warning
02. // because it is ambiguous if "param"
03. // is an object of class "Y" or
04. // the enumeration "Y".
05.
06. class X {
07.   enum Y { Red, Blue, Green }
08.   showY(Y param) {
09.     System.out.println(param); }
10. }
11.
12. class Y { }
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // This example does not generate the warning
02.
03. class X {
04.   enum Y { Red, Blue, Green }
05.   showY(Y param) {
06.     System.out.println(param); }
07. }
08.
09. class Z { }
10.
```

Load the above code into classic UmpleOnline

Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // This example does not generate the warning
02.
03. enum Z { Red, Blue, Green }
04.
05. class X {
06.   showY(Y param) {
07.     System.out.println(param); }
08. }
09.
10. class Y { }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W103 Enumeration Causes Event Parameter Ambiguity

Umple semantic warning reported when an enumeration makes an event parameter's type ambiguous. This occurs when there is another class that could also be the parameter's type.

Example

view plain

```
01. // This example generates the warning
02. // because "goToS2" has "param" which could be
03. // either an object of class "Y"
04. // or the enumeration "Y"
05.
06. class X {
07.   enum Y { Red, Blue, Green }
08.   sm {
09.     s1 {
10.       goToS2(Y param) /{
11.         System.out.println(param); } -> s2;
12.     }
13.     s2 {}
14.   }
15. }
16.
17. class Y { }
18.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // This example does not generate the warning
02.
03. class X {
04.   enum Y { Red, Blue, Green }
05.   sm {
06.     s1 {
07.       goToS2(Y param) /
```

```
08.         { System.out.println(param); } -> s2;
09.     }
10.     s2 {}
11.   }
12. }
13.
14. class Z { }
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E104 Enumeration in Bi-directional Association

Umple semantic error reported when an enumeration is used in a bi-directional association.

In Umple, enumerations cannot be used in bi-directional associations.

Example

```
view plain
01. // This example generates the error
02.
03. class X {
04.   enum Y { Red, Blue, Green }
05.   0..1 -- * Y;
06. }
07.
08. class Y{ }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain
01. // This example does not generate the error
02.
03. class X {
04.   enum Y { Red, Blue, Green }
05.   0..1 -- * Z;
06. }
07.
08. class Z { }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E105 Enumeration in Composition

Umple semantic error reported when an enumeration is used in a composition.

In Umple, enumerations cannot be used in compositions.

Example

```
view plain
01. // This example generates the error
02.
03. class X {
04.   enum Y { Red, Blue, Green }
05.   0..1 <@>- * Y;
06. }
07.
08. class Y {
09.   name;
10. }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain
01. // This example does not generate the error
02.
03. class X {
04.   enum Y { Red, Blue, Green }
05.   0..1 <@>- * Z;
06. }
07.
08. class Z {
09.   name;
10. }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W106 Enumeration in Uni-directional Association

Umple semantic warning reported when an enumeration is used in a uni-directional association.

Example

```
view plain
01. // This example generates the warning
02.
03. class X {
04.   enum Y{ Red, Blue, Green }
05.   0..1 -> * Y;
06. }
07.
08. class Y { }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

```
view plain
01. // This example generates the warning
02.
03. class X {
04.   enum Y{ Red, Blue, Green }
05.   0..1 <- * Y;
06. }
07.
08. class Y { }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain
01. // This example does not generate the warning
```

```
02.
03.  class X {
04.    enum Y { Red, Blue, Green }
05.    0..1 -> * Z;
06.  }
07.
08.  class Z { }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W107 Names Differ Only By Case

Umple semantic warning reported when two or more classes or interfaces, attributes of a class, or mixsets have names that differ only in case

Names such as *Ping* and *PinG*, or *apple* and *apPle*, are distinct identifiers to Umple and to languages such as Java, but they are easily confused by people and can lead to defects (see CWE-178). They also cause concrete problems in generated code. Each class and interface becomes a file named after it, so when two of them are generated into the same directory on a case-insensitive file system, such as the defaults on Windows and macOS, one file overwrites the other. Attribute names are capitalized to form method names, so *apple* and *apPle* produce *setApple* and *setApPle*, which PHP treats as the same method. Mixsets named, for example, *M1* and *m1* are separate mixsets, and a use statement that names one of them does not include the other.

Class and interface names are compared together. The attributes of a class are compared with each other, including those it inherits from its superclasses and those it gets from traits; the same letters in unrelated classes are fine. Mixsets are compared wherever they are declared, including inside method bodies. The message lists all the names that clash and points at the first of them that duplicates an earlier one. A clash with an inherited attribute is reported once, at the subclass attribute that introduces it. When the names differ only in the case of their first letter, such as *attr* and *Attr*, the generated methods are identical and error E048 is reported as well when Java or PHP code is generated. To avoid the message, rename the elements so their names differ by more than their case.

The first example below is not compiled when the manual is built, since its generated code shows exactly these problems: on some file systems and in PHP it does not compile.

Example

view plain

```
01. // This example generates the message three times:
02. // for the classes, the attributes and the mixsets
03. use Logging, logging;
04.
05. mixset logging class Account {
06.   logLevel;
07. }
```

```
08.
09.  mixset Logging class Account {
10.    logFile;
11.  }
12.
13.  class Customer {
14.    firstName;
15.    firstname;
16.  }
17.
18.  class CUSTOMER {}
19.
20.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01.  // The following shows how to avoid the message:
02.  // names differ by more than their case
03.  use Logging;
04.
05.  mixset Logging class Account {
06.    logLevel;
07.    logFile;
08.  }
09.
10.  class Customer {
11.    firstName;
12.    preferredName;
13.  }
14.
15.  class CorporateCustomer {}
16.
17.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E112 Duplicate Constants in Interface

Umple semantic error reported when an interface is given two or more constants with the same name.

Constants in an attribute must have distinct names. The solution to this error is to rename or delete one of the constants.

Example

view plain

```
01. // The following example shows how to generate
02. // this error.
03. interface X {
04.   const A="dog";
05.   const A="cat";
06. }
07.
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W141 Value Type Mismatch

Umple syntactic warning reported when an attempt is made to initiate an attribute with a value that does not match the type of the attribute

Numbers must be initialized with numbers, strings with strings in double quotes, and Booleans with true or false. For Dates the value format is a string "yyyy-mm-dd"; for Times the format is "hh:mm:ss".

Note that for an initialization value with no type specified will result in the attribute having its type inferred from the value. This is demonstrated in the third example.

Example

view plain

```
01. // The following will generate
02. // this warning on every line of the class
03. class X {
04.   Date d = "20130708";
05.   Integer i = "abc";
06.   String s = 123;
07.   Boolean b = 1;
08.   Time t = "120000";
09. }
10.
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following shows how to solve the above problem
02. class X {
03.   Date d = "2013-07-08";
04.   Integer i = 7;
05.   String s = "123";
06.   Boolean b = true;
07.   Time t = "12:00:00";
08. }
```

```
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following shows how to solve
02. // the above problem not even showing the type at all
03. class X {
04.   d = "2013-07-08";
05.   i = 7;
06.   s = "123";
07.   b = true;
08.   t = "12:00:00";
09.   m = 2.3;
10. }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W142 Type is access specifier

Umple warning when a type looks like it was meant as a visibility specifier

Umple attributes do not support visibility specifiers such as public, private and protected. An attribute is private by default, and programmers are supposed to use the corresponding generated get method to read it, or the set method to modify it.

This warning appears when one of the visibility keywords precedes an attribute.

Example

```
view plain

01. // This case raises the warning
02. // because it is probably an error
03. // Umple will consider "x" to have type "public"
04. class X {
05.   public x;
06. }
07.
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // This is probably what was intended,
02. // attributes have private access by default
03. // but the programmer would call
04. // public String getX() to read it
05. // and public boolean setX(String) to set it
06. class X {
07.   x;
08. }
09.
10.
```

Load the above code into classic UmpleOnline

Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // Although not recommended, the programmer
02. // can also bypass Umple and directly specify
03. // both the visibility directive and the type.
04. // Umple will not see that as an attribute
05. // and will inject the line of code into the
06. // output directly.
07. class X {
08.   public String x;
09. }
10.
11.
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E180 Duplicate Association Name Class Hierarchy

Umple semantic error reported when a subclass has an association that is not a specialization, but where the association has the same name as the superclass association

A subclass can not have an association with the same role name as an association of its superclass, unless it is a valid specialization. A valid specialization would be to the same class, but with a refined multiplicity (see the second example)

Example

view plain

```
01. class Person {
02.    * -> * Person friends;
03. }
04.
05. class Student {
06.    isA Person;
07.    * -> * Dog friends;
08. }
09.
10. class Dog {
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // This example shows a valid specialization
02. // of the friends association in which the
03. // multiplicity 0..3 is refined to 0..2, so a
04. // person in general may have 0 to 3 dogs but
05. // students are limited to 2.
06. // This conforms to the Liskov substitution
07. // principle: The data invariant on Person
08. // which limits them to 3 dogs
09. // is still true in the subclass Student.
10.
```

```
11. class Person {
12.    * -> 0..3 Dog friends;
13. }
14.
15. class Student {
16.    isA Person;
17.    * -> 0..2 Dog friends;
18. }
19.
20. class Dog {
21. }
22.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E181 Association Specialization Invalid Multiplicity

Umple error raised when an association specialization has an invalid multiplicity

When performing an Association Specialization, the multiplicities are more specific than that of the previously defined association. If the multiplicities on either side of the specialized association are less strict/specific, the association is considered invalid and this error is raised.

If you intend to perform specialization, this error can be avoided by narrowing the multiplicity to be more specific than that of the superclass association.
If you did not intend to perform specialization, you may have reused a rolename in the association. See E019.

Example

```
view plain
01. // This example generates the error
02.
03. class Vehicle {}
04. class Wheel {}
05. class Bicycle {isA Vehicle;}
06.
07. // Parent multiplicity of 0..1 to 0..2
08. association { 0..1 Vehicle vehicleA --
     0..2 Wheel vehicleWheel;}
09.
10. // Broader child multiplicity of 0..1 to 0..3
11. association { 0..1 Bicycle vehicleA --
     0..3 Wheel vehicleWheel;}
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to the above so the message no longer appears

```
view plain
01. // This example shows how to resolve the error
02.
```

```
03. class Vehicle {}
04. class Wheel {}
05. class Bicycle {isA Vehicle;}
06.
07. // Parent multiplicity of 0..1 to 0..2
08. association { 0..1 Vehicle vehicleA --
     0..2 Wheel vehicleWheel;}
09.
10. // Narrower child multiplicity of 0..1 to 0..1
11. association { 0..1 Bicycle vehicleA --
     0..1 Wheel vehicleWheel;}
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E200 Trait Identifier Invalid

Umple syntactic error related to identifiers of traits

In Umple, identifiers used for traits must be alphanumeric. It means that must start with an alpha character, or _ .

Example

view plain

```
01. //This example shows an error
02. // in the name of trait
03. trait T@ {
04.     //traits elements
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. //This example shows an error
02. // in the name of trait
03. trait 1T {
04.     //traits elements
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //This example shows a valid name for the trait
02. trait Color {
03.     //traits elements
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //This example shows a valid name of traits
02. trait Equality {
03.     //traits elements
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W201 Trait Name Syntax

Umple syntactic warning related to identifiers of traits

In Umple, identifiers used for traits, classes and interfaces should start with a capital letter (or "_"), and other elements should start with a lower case letter.

Example

view plain

```
01. // This example shows a warning
02. // in the name of traits because
03. // they start with lower case.
04. trait color {
05.     //traits elements
06. }
07.
08. trait equality{
09.     //traits elements
10. }
11. // @@@skipcompile no code
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //This example shows a valid name for the trait
02. trait Color {
03.     //traits elements
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //This example shows a valid name of traits
02. trait Equality {
03.     //traits elements
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E202 Trait not defined

Umple semantic error resulting from a trait not being defined

In Umple, when traits are used inside of classes or traits they have to be defined. The Umple compiler does not allow the use of traits that are not defined in the system.

Example

```
view plain

01. //In  this example, trait "T"
02. // uses trait "T1" which is not available.
03. interface I{
04.     //elements
05. }
06.
07. class A {
08.     isA I;
09.     isA T;
10. }
11.
12. trait T{
13.   isA T1;
14. }
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. //In  this example, the error
02. // has been resolved just by
03. // defining trait T1.
04. interface I{
05.     //elements
06. }
07.
08. class A {
09.     isA I;
10.     isA T;
```

```
11. }
12.
13. trait T{
14.   isA T1;
15. }
16.
17. trait T1{
18.     //elements
19. }
20.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E203 Not having an unique identifier

Umple semantic error related to not using traits which do not have unique identifiers

In Umple, all elements which are capable of being reused should have unique identifiers. Traits, as reusable elements in Umple also have to follow the same rule. Identifiers of traits should be unique in the system. The conflict happens among classes, interfaces, and traits. Therefore, these three elements always should have unique identifiers.

Example

view plain

```
01. // In  this example, Identifier of
02. // trait "I" is not unique.
03. class A{
04.     isA I;
05. }
06.
07. class I {
08.   //elements
09. }
10.
11. trait I{
12.   //elements
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //In  this example, Identifier of
02. // trait "T" is unique.
03. class A{
04.     isA T;
05. }
06.
07. class I {
08.   //elements
```

```
09. }
10.
11. trait T{
12.   //elements
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E204 Self Use of Traits

Umple semantic error related to self use of traits

In Umple, traits cannot be used in an explicit or implicit cyclic way. It means that a trait cannot use itself and it also cannot be used in a cyclic use. This error happens when a trait extends itself.

Example

view plain

```
01. // In this example, there is an explicit
02. // use of a trait inside of itself.
03. class A{
04.     isA T;
05. }
06.
07. trait T{
08.   isA T;
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E205 Cycle in Traits

Umple semantic error related to a cycle of trait use

In Umple, traits cannot be used in an explicit or implicit cyclic way. This means that a trait cannot use itself and it also cannot be used in a cyclic hierarchy. This error happens when a hierarchy is created completely based on traits. Moreover, this issue generally arises because of invalid design or error in typing a trait name.

Example

view plain

```
01. // In this example, there is a cycle
02. // created in a hierarchy.
03. class A{
04.     isA T;
05. }
06. 
07. trait T{
08.   isA T1;
09. }
10. trait T1{
11.     isA T;
12. }
13. 
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E206 Satisfaction of Bound Type

Umple semantic error related to binding a type to a template parameter

If clients bind types to the template parameters of used traits and the types exist but they do not satisfy the constraints of the template parameters, this error is raised. The constraint is related to the implementation of interfaces.

Example

view plain

```
01. // In this example, there is an error
02. // because class C1 does not implement
03. // interface I. Any bound value to template
04. // parameter TP must implement interface I.
05. trait T1<TP isA I>{
06.   /*implementation*/
07. }
08. interface I{/*implementation*/}
09. class C1{/*implementation*/}
10. class C{
11.   isA T1< TP = C1 >;
12.   /*implementation*/
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E208 Required Methods Not Available

Umple semantic error related to not having required methods of traits in classes

Traits need to have required methods in order to provide the functionality they have designed to provide. These required methods can provide special functionality or be a way of accessing states (attributes) in host classes. This error is raised when a required method is not available.

Example

view plain

```
01. // In this example, the required method
02. // "FinalResult()" in Trait "T" is
03. // not satisfied by class "A".
04. class A{
05.     isA T;
06. }
07.
08. trait T{
09.     Integer FinalResult();
10.     void calculate(){
11.         //method FinalResult() will be used here;
12.     }
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // In this example, the require method
02. // "FinalResult()" in Trait "T" is
03. // satisfied by class "A".
04. class A{
05.     isA T;
06.     internal Integer x;
07.     Integer FinalResult(){
08.         return x/100;
09.     }
```

```
10. }
11.
12. trait T{
13.     Integer FinalResult();
14.     void calculate(){
15.         //method FinalResult() will be used here;
16.     }
17. }
18.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E210 Conflict in Methods

Umple semantic error related to having two methods with the same signature but different implementation

Traits can be used in several hierarchies without any priority. It means that classes can be composed of several traits and a trait can be composed of other traits. This brings cases in which two methods with the same signature come from either different traits or a different hierarchy path. If those methods come from two different traits, it is obvious that there is a conflict because those have been implemented for different purposes. On the other hand, if two methods come from two different paths but the source of those methods is the same trait, then there is no conflict. However, there is a case in which the source is the same but one of the methods is overridden by traits somewhere in the path. In this case, there is a conflict because now, the functionality is not unique. The Umple compiler detects these cases and raises this error.

Example

view plain

```
01. // In this example, there is a conflict
02. // in class "A" on the method "test()".
03. class A{
04.     isA T1;
05.     isA T2;
06. }
07. trait T1 {
08.     void test(){
09.     //special implementation by T1
10.     }
11. }
12. trait T2 {
13.     void test(){
14.     //special implementation by T2
15.     }
16. }
17.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // In this example, there is
02. // a conflict in trait "T"
03. // on the method "test()".
04. class A{
05.     isA T;
06. }
07. trait T{
08.     isA T1;
09.     isA T2;
10. }
11. trait T1{
12.     void test(){
13.     //special implementation by T1
14.     }
15. }
16. trait T2{
17.     void test(){
18.     //special implementation by T2
19.     }
20. }
21.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // In this example, there is a
02. // no conflict in class "A".
03. class A{
04.     isA T1;
05.     isA T2;
06. }
07. trait T1 {
08.     isA M;
09. }
10. trait T2 {
11.     isA M;
12. }
13. trait M{
```

```
14.     void test(){
15.     //special implementation by M
16.     }
17. }
18.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // In this example, there is a conflict
02. // in class "A" because method "test"
03. // is overridden in trait "T2".
04. class A{
05.     isA T1;
06.     isA T2;
07. }
08. trait T1 {
09.     isA M;
10. }
11. trait T2 {
12.     isA M;
13.     void test(){
14.     //special implementation by T2
15.     }
16. }
17. trait M{
18.     void test(){
19.     //special implementation by M
20.     }
21. }
22.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E211 Two or More Modifications

Umple semantic error related to two or more modifications of provided methods of traits

When traits are used inside classes or traits, it is possible to add or remove provided methods. This feature is used to resolve conflicts and when we do not need some provided methods or just need one of them. Logically, it is not correct to add a method twice, or add and then remove a method. These problems are detected by the Umple compiler.

Example

view plain

```
01. // In this example, there are
02. // two modifications ("add") for
03. // the method "show()" when trait "T2"
04. // is used inside of trait "T1".
05. class A{
06.     isA T1;
07. }
08. trait T1{
09.     isA T2 <+show(),+show()>;
10. }
11. trait T2{
12.     void show(){
13.     //implementation
14.     }
15. }
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // In this example, there are
02. // two modifications ("add" and "remove")
03. // for the method "show()" when trait "T2"
04. // is used inside of trait "T1".
05. class A{
06.     isA T1;
```

```
07. }
08. trait T1{
09.    isA T2 <-show(),+show()>;
10. }
11. trait T2{
12.    void show(){
13.    //implementation
14.    }
15. }
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E212 Methods Not Available

Umple semantic error related to modification of unavailable methods in traits

When traits are used inside classes or traits, it is possible to add or remove provided methods, and to change their visibility and names. This feature is used to resolve conflicts when we do not need some provided methods, just need one of them, need different visibilty, or need a different name. Logically, it is not correct to do these operations on methods which are not available. These problems are detected by the Umple compiler.

Example

view plain

```
01. // In this example, there is an error
02. // because trait "T1" tries to remove
03. // method "show()" from trait "T2"
04. // while it is not available
05. // in trait "T2".
06. class A{
07.    isA T1;
08. }
09. trait T1{
10.    isA T2 <-show()>;
11. }
12. trait T2{
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // In this example, there is an error
02. // because trait "T1" tries to add
03. // just method "show()" from trait "T2"
04. // to itself. while it is not
05. // available in trait "T2".
06. class A{
```

```
07.     isA T1;
08. }
09. trait T1{
10.     isA T2 <+show()>;
11. }
12. trait T2{
13.
14. }
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // In this example, there is an error
02. // in class "A" because it tries to
03. // change visibility of method "test()"
04. // which is not available in trait T.
05. // Consider that the method "test()"
06. // defined in trait "T" is a
07. // required method.
08. class A{
09.     isA T<test() as private>;
10. }
11. trait T{
12.     void test();
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E213 Bidirectional Association in Traits

Umple semantic error related to having bidirectional association with interfaces

Traits can make associations with interfaces, classes, and template parameters. If one end of the association is a template parameter, the binding type must be checked to make sure it is compatible with the type of the association. For example, if a trait has a bidirectional association with a template parameter, the binding value cannot be an interface and it must be a class. Breaking this constraint is reported to modelers using this error code.

Example

view plain

```
01. // In this example, there is an error
02. // because a bidirectional association
03. // is created with interface I,
04. // which is not valid.
05. interface I {
06.   /*implementation*/
07. }
08. trait T <RelatedClass> {
09.   0..1 -- * RelatedClass;
10.   /*implementation*/
11. }
12. class C{
13.  isA T<RelatedClass=I>;
14.  /*implementation*/
15. }
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E214 Two Identical Parameter Names

Umple semantic error related to having two identical template parameter names

When defining traits, it is possible to define template parameters for traits. The names of these parameters should be unique in order to let the compiler perform the correct binding for them. Therefore, when there are two or more template parameters of the same name, the Umple compiler detects it as an error.

Example

view plain

```
01. // In this example, there is an error
02. // because trait T has two template
03. // parameters with the same name.
04. trait T<TP,TP>{
05.   TP name;
06. }
07. class C1{
08.   isA T;
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E215 Template Parameter Not Available

Umple semantic error related to binding a template parameter of a trait which is not available

When using traits, we can bind types to template parameters. In the process of binding, we can just refer to the name of the parameters which are available. The Umple compiler detects cases in which there are template parameters that are not defined in a trait.

Example

view plain

```
01. // In this example, there is an error
02. // because class A cannot bind
03. // type String to template parameter Y,
04. // which is not defined. for trait T.
05. class A{
06.   isA T< X= Integer, Y = String>;
07. }
08. trait T<X>{
09.     X variable;
10. }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E216 Twice Binding of a Parameter

Umple semantic error related to binding a template parameter twice

When using traits, we can bind values for template parameters. In the process of binding, we can bind two values for a template parameter. This brings a case in which there is no clear binding. The Umple compiler detects this case and prevents the system from being compiled.

Example

view plain

```
01. // In this example, there is an error
02. // because there are two bindings for
03. // template parameter "X" in class "A".
04. class A{
05.     isA T< X = B , X = C >;
06. }
07. Class B{
08.     //elements
09. }
10. Class C{
11.     //elements
12. }
13. trait T<X,Z>{
14.     //elements
15. }
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E217 Conflict in Types of Attributes

Umple semantic error related to having an attribute from a trait with different types

We can define template parameters for traits and use them with different bindings in several hierarchy paths. If types of some attributes are based on template parameters, there is a case in which those are bound with different values. In a case that a trait is going to be used in a diamond form of hierarchy, this can result in a conflict. Note that if the types are the same, then there is no conflict.

Example

view plain

```
01. // In this example, there is a
02. // conflict because in trait "T"
03. // there will be two attributes
04. // with the same name "data" but
05. // with different types
06. // which are "B" and "C".
07. class A{
08.     isA T;
09. }
10. class B{
11.     //elements
12. }
13. class C{
14.     //elements
15. }
16. trait T{
17.     isA T1;
18.     isA T2;
19. }
20. trait T1{
21.     isA Share<Type = B>;
22. }
23. trait T2{
24.     isA Share<Type = C>;
25. }
26. trait Share<Type> {
27.     Type data;
28. }
29.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W218 Conflict in Attributes

Umple semantic error related to having an attribute with the same name in a trait and a class

When classes uses traits, all attributes in traits are flattened in classes. Therefore, if there are attributes with the same name, they might create a conflict. The Umple compiler considers this as a warning, because it is the developers' responsibility to decide about the nature of the conflict. Sometimes, developers indicate exactly the same attributes and therefore there is no need to consider the conflict as an error. However, generally, if there are some attributes in host classes which traits need to use them, traits define them as required methods (needed accessors). In this case, host classes need to have accessors for those attributes. In the case of this warning, the Umple compiler just removes one of those attributes and proceeds.

Example

view plain

```
01. // In this example, there is a warning
02. // because in class "A" there will be
03. // two attributes with the name called "name".
04.
05. class A{
06.    isA T;
07.    name;
08. }
09. trait T{
10.    name;
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. // In this example, there is a
02. // warning because in trait "T"
03. // there will be two attributes
04. // with the name called "name".
```

```
05. class A{
06.    isA T;
07. }
08. trait T{
09.    isA T1;
10.    name;
11. }
12. trait T1{
13.    name;
14. }
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E219 No Binding for Parameters

Umple semantic error related to having two same template parameter names

When defining traits, it is possible to define template parameters for traits. It means when traits with template parameters are going to be used by classes or traits, they must bind values for the parameters. If there is no binding for a parameter, the Umple compiler raises an error and shows which parameter does not have a value.

Example

```
view plain

01. // In this example, there is an error
02. // because class "A" doesn't bind
03. // a value to one of the template
04. // parameters of trait "T" named "Y".
05. class A{
06.     isA T<X = B>;
07. }
08. class B{
09.     //elements
10. }
11. trait T<X,Y>{
12.     //elements
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

```
view plain

01. // In this example, there is an error
02. // because class "A" doesn't bind
03. // a value to template parameters
04. // of trait "T".
05. class A{
06.     isA T;
07. }
08. class B{
```

```
09.     //elements
10.  }
11.  trait T<X,Y>{
12.     //elements
13.  }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E220 Two Methods With the Same Name

Umple semantic error related to changing name of a provided method in traits to one which already exists

When changing the name of provided methods in traits, it is necessary to be sure that there are no other methods with the same names that come from other traits. If the changed name already exists inside of a class, in this case there is no problem becase methods of classes have priority over ones coming from traits. In this case, the method from traits will be disregarded. Otherwise, there is a conflict on the name of methods. The Umple compiler detects this case and raises the error.

Example

```
view plain

01. // In this example, there is an error
02. // because in class "A" there will be
03. // two methods with the name "ShowInConsole".
04. class A{
05.     isA T<show() as ShowInConsole>;
06. }
07. trait T{
08.     isA T1;
09.     void show(){/*T*/}
10. }
11. trait T1{
12.     void ShowInConsole(){/*T1*/}
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

```
view plain

01. // In this example, there is an error
02. // because in class "A" there will be
03. // two methods with the name "ShowInConsole".
04. class A{
05.     isA T<show() as ShowInConsole>;
06.     isA T1;
```

```
}
trait T{
    void show(){/*T*/}
}
trait T1{
    void ShowInConsole(){/*T1*/}
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
// In this example, there is no error
// because in class "A"
// method "ShowInConslole" has high priority.
class A{
    isA T<show() as ShowInConsole>;
    isA T1;
    void ShowInConsole(){
    //Implementation
    }
}
trait T{
    void show(){
    //Implementation
    }
}
trait T1{
    void ShowInScreen(){
    //Implementation
    }
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E221 Availability of Bound Type

Umple semantic error related to binding types to template parameters

When using a trait with a template parameter, a type must be bound to that template parameter. If the bound type is not available in the system, the Umple compiler raises this error.

Example

view plain

```
01. // In this example, there is an error
02. // because class C binds type C1 to the
03. // template parameter of trait T,
04. // while type C is not available
05. // in the system.
06. trait T<TP>{
07. /*implementation*/
08. }
09. class C{
10.   isA T<TP=C1>;
11.   /*implementation*/
12. }
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E222 Satisfaction of Required Interfaces

Umple semantic error related to satisfaction of required interfaces of traits

When a class uses traits, it needs to implement the required interfaces of those traits, otherwise, the Umple compiler detects missing interfaces and raises this error code. If a trait uses other traits with required interfaces, those required interfaces are added to the set of required interfaces of the trait and final clients are required to implement all of those required interfaces as well.

Example

```
view plain

// In this example, there is an error
// because class C must implement
// required interface of trait T,
// which is interface I.
trait T{
  /*implementation*/
  isA I;
}
interface I{
  /*implementation*/
}
class C{
  isA T;
  /*implementation*/
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E223 Availability of Types

Umple semantic error related to availability of types used to constraint template parameters

If traits define template parameters and put constraints on them, the classes and interfaces involved in the constraints must exist in the system under design, otherwise, this error is raised.

Example

view plain

```
01. // In this example, there is an error
02. // because type I used to constraint
03. // template parameters TP in trait T
04. // does not exist.
05. trait T<TP isA I>{
06.   /*implementation*/
07. }
08. class C1{
09.   /*implementation*/
10. }
11. class C1{
12.   isA T< TP = C1 >;
13.   /*implementation*/
14. }
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E224 Multiple Inheritance Constraint

Umple semantic error related to putting multiple inheritance constraint on template parameters

If a multiple inheritance is applied as a constraint to a template parameter, it is detected and this error is raised. Umple does not support multiple inheritance.

Example

view plain

```
01. // In this example, there is an error
02. // because trait T puts multiple
03. // inheritance constraint on its
04. // template parameter TP.
05. trait T<TP isA C1&C2>{
06.   /*implementation*/
07. }
08. class C1{
09.   /*implementation*/
10. }
11. class C2{
12.   /*implementation*/
13. }
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E225 Satisfaction of Bound Type

Umple semantic error related to binding a type to a template parameter

If clients bind types to the template parameters of used traits and the types exist but they do not satisfy the constraints of the template parameters, this error is raised. The constraint is related to inheritance.

Example

```
view plain

01. // In this example, there is an error
02. // because class C1 is not subclass
03. // of class C2.
04. // Any bound value to template parameter
05. // TP must be a subclass of class C2.
06. trait T<TP isA C2>{
07.   /*implementation*/
08. }
09. class C{
10.   isA T< TP = C1 >;
11.   /*implementation*/
12. }
13. class C1{/*implementation*/}
14. class C2{/*implementation*/}
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W228 Different Initial States

Umple semantic error related to composing state machines or regions

If two state machines or regions are being composed and do not the same initial states.

Example

view plain

```
01. // In this example, there is an warning
02. // because two traits T1 and T2 have
03. // state machines with the same name but
04. // with different initial states.
05. trait T1{
06.   sm{
07.     s1{ e1 ->  s2;}
08.     s2{ e3 ->  s3;}
09.     s3{ e2 ->  s2;}
10.   }
11. }
12. trait T2{
13.   sm{
14.     t1{ t1 ->  t2;}
15.     t2{ t3 ->  s3;}
16.     s3{ t2 ->  t2;}
17.   }
18. }
19. class C1{
20.   isA T1,T2;
21. }
22.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E229 Two Times Renaming

Umple semantic error related to state machine operators

When a state is to be renamed, it should not be renamed more than one time. If this happens, the Umple compiler detects that and raises this error.

Example

view plain

```
01. // In this example, there is an error
02. // because class C tries to
03. // rename states s0 of trait T two times.
04. trait T {
05.   sm{
06.     s0{
07.       e1-> s1;
08.       s11{ e12-> s12; }
09.       s12{ e11-> s11; }
10.     }
11.     s1{ e0-> s1;  }
12.   }
13. }
14. class C {
15.   isA T<sm.s0 as state0, sm.s0 as state01>;
16. }
17.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E230 Availability of State Machines

Umple semantic error related to state machine operators

When a state machine or state with a given name is specified by the renaming operator, it must be available in the trait being operated on, either directly in the trait or another trait used by the trait. Otherwise, the Umple compiler raises this error.

Example

view plain

```
01. // In this example, there is an error
02. // because state machine sm3 is
03. // not available in trait T.
04. trait T {
05.   sm1{
06.     s0 {e1-> s1;}
07.     s1 {e0-> s0;}
08.   }
09.   sm2{
10.     s0 {e1-> s1;}
11.     s1 {e0-> s0;}
12.   }
13. }
14. class C3 {
15.  isA T<sm3 as mach1>;
16. }
17.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E231 Availability of Events

Umple semantic error related to state machine operators

When an event name is used with a specific name for its state machine, the state machine and event must be available in the trait, otherwise, the Umple compiler raises this error.

Example

view plain

```
01. // In this example, there is an error
02. // because event e2() is not available
03. // in state machine sm.
04. trait T {
05.   sm{
06.     s0{
07.       e1-> s1;
08.       s11{ e12-> s12; }
09.       s12{ e11-> s11; }
10.     }
11.     s1{ e0-> s1;  }
12.   }
13. }
14. class C {
15.   isA T<sm.e2() as event2>;
16. }
17.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E232 Availability of Events

Umple semantic error related to state machine operators

When an event name is used with a specific name for its state machine, the state machine and event must be available in the trait. The same restriction is applied when the symbol * is used for the name of state machines. If the event is not available in at least one of the state machines existing in the trait, then the Umple compiler raises this error code.

Example

view plain

```
01. // In this example, there is an error
02. // because event e2() is not available
03. // in state machines of trait T.
04. trait T {
05.   sm{
06.     s0{
07.       e1-> s1;
08.       s11{ e12-> s12; }
09.       s12{ e11-> s11; }
10.     }
11.     s1{ e0-> s1;  }
12.   }
13. }
14. class C {
15.   isA T<*.e2() as event2>;
16. }
17.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E233 Removing Initial State

Umple semantic error related to state machine operators

The removing operator cannot be applied to the initial states of a state machine or a composite state. If the modeler applies it in such cases, the Umple compiler raises this error.

Example

view plain

```
01. // In this example, there is an error
02. // because the initial state of a
03. // state machine cannot be removed.
04. trait T {
05.   sm{
06.     s0{
07.       e1-> s1;
08.       s11{ e12-> s12; }
09.       s12{ e11-> s11; }
10.     }
11.     s1{ e0-> s1;  }
12.   }
13. }
14. class C {
15.   isA T<-sm.s0>;
16. }
17.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E234 Compositing Non-Deterministic Transitions

Umple semantic error related to state machine composition algorithm

Umple does not support non-deterministic state machines and so composed state machines must be deterministic as well. Our composition algorithm automatically detects transitions that cause the composed state to be non-deterministic and raises this error code.

The example below shows how a transition in a base state could be cause to have non-determinism. As seen, both state machines in class C1 and trait T1 have state s1 and they need to be composed. The state s1 in trait T1 has transition e1[x>0] and the transition's destination is s2. The base state s1 has transition e1 without a guard and its destination state is state s3. This transition does not exist in the state s1 coming from the trait, so it can be added to the composed state s1. However, this causes a situation in which the composite state machine can be in two states s2 and s3 simultaneously. Therefore, the composition is not allowed.

Example

view plain

```
01. // In this example, there is an error
02. // because the composed state machine
03. // will be nondeterministic.
04. trait T1{
05.   sm{
06.     s1{ e1[x>0] -> s2;  }
07.     s2{ e2 -> s1; }
08.   }
09. }
10. class C1{
11.   isA T1;
12.   sm{
13.     s1{
14.       e1 -> s3;
15.       e3 -> s3;
16.     }
17.     s3{ }
18.   }
19. }
20.
```

Load the above code into classic UmpleOnline

Load the above code into UmpleOnline V2

E235 SuperCall Conflict

Umple semantic error related to state machine composition algorithm

When clients use more than one trait, using the keyword superCall can cause a conflict. This happens because an order of execution is required.

As seen in the example below, class C1 uses two traits T1 and T2 and there is matching state s1 and matching transition e1 needed to be composed. Since the keyword has been used in the base transition e1, it indicates that the actions of matching transitions are to be executed, but there is more than one action to be executed. Therefore, the Umple compiler prevents these models from being composed.

Example

view plain

```
01. // In this example, there is an error
02. // because the keyword superCall can be
03. // pointed out to two used traits.
04. trait T1{
05.   sm{
06.     s1{ e1 -> /{action1();} s1; }
07.   }
08. }
09. trait T2{
10.   sm{
11.     s1{ e1 -> /{action2();} s2;}
12.     s2{}
13.   }
14. }
15. class C1{
16.   isA T1,T2;
17.   sm{
18.     s1{ e1 -> /{superCall; action3();} s3; }
19.     s3{}
20.   }
21. }
22.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E236 Two State Entries

Umple semantic error related to state machine composition algorithm

A client can use more than one trait (and obtain more than one state machine) and so it is possible to have more than two states whose entry (or exit) actions (and do activities) need to be composed with the base state's entry action. In this case, the composition algorithm needs to consider an order among them. Since we do not consider any order when traits are used by clients, this should be respected in the composition of actions. Therefore, the Umple compiler detects this case and raises this error. Note that this conflict happens if more than one state coming from used traits have an entry action.

The example below shows a case in which composition of entry actions is considered as a conflict. The base state s1 in class C1 has no entry action and so it can accept entry actions coming from state machines in T1 and T2. However, there are two entries coming from used traits which cause a conflict. Note that if class C1 did not have state machine sm, it would still be a conflict for composition. However, if state s1 in class C1 had an entry then this would not be a conflict because those entries coming from traits would be disregarded.

Example

view plain

```
01. // In this example, there is an error
02. // because of having different ways
03. // to compose entries of two states
04. // S1 coming from traits T1 and T2.
05. trait T1{
06.   sm{
07.     s1{ entry /{action1();} }
08.   }
09. }
10. trait T2{
11.   sm{
12.     s1{ entry /{action2();} }
13.   }
14. }
15. class C1{
16.   isA T1, T2;
17.   sm{
18.     s1{}
```

```
19.   }
20. }
21.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E237 Composing Two State Entries

Umple semantic error related to state machine composition algorithm

A client can use more than one trait (and obtain more than one state machine) and so it is possible to have more than two states whose entry (or exit) actions (and do activities) need to be composed with the base state's entry action. In this case, the composition algorithm needs to consider an order among them. Since we do not consider any order when traits are used by clients, this should be respected in the composition of actions. Therefore, the Umple compiler detects this case and raises this error. Note that this conflict happens if more than one state coming from used traits have an entry action.

The example below shows a case in which composition of entry actions is considered as a conflict. The base state s1 in class C1 has no entry action and so it can accept entry actions coming from state machines in T1 and T2. However, there are two entries coming from used traits which cause a conflict. Note that if class C1 did not have state machine sm, it would still be a conflict for composition. However, if state s1 in class C1 had an entry then this would not be a conflict because those entries coming from traits would be disregarded.

Example

view plain

```
01. // In this example, there is an error
02. // because class C rename the region
03. // name r1 to r2 which is already
04. // available in the state machine sm.
05. trait T{
06.   sm {
07.     s1{
08.       r1{ e1-> r11; }
09.       r11{}
10.       ||
11.       r2{ e2-> r21; }
12.       r21{}
13.     }
14.   }
15. }
16. class C{
17.   isA T<sm.s1.r1 as r2>;
18. }
```

19.

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W301 Tracing Entity Not Found

Umple semantic warning issued when tracing a non-existent model entity

When tracing an entity in a class, a warning will be issued if the entity is not found and the trace directive will be ignored.
The name of the entity might be misspelled.

Example

```
view plain

01. //The trace directive causes the
02. //warning to be issued as the attribute
03. //id is not found
04. class A {
05.   Integer identifier;
06.
07.   trace id;
08. }
09. // @@@skipphpcompile See issue 596 (PHP tracing causes issues
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. //The warning is resolved
02. //by correcting the name of the
03. //attribute to be traced
04. @SuppressWarnings("deprecation")
05. class A {
06.   Integer identifier;
07.
08.   trace identifier;
09. }
10. // @@@skipphpcompile See issue 596 (PHP tracing causes issues
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W302 Tracer Not Recognized

Umple semantic warning issued when a tracer is specified but not recognized

A certain set of MOTL tracers can be specified for use in Umple. Indicating a tracer that is not supported will cause the tracer directive to be ignored, and the Console tracer will be used.

Example

view plain

```
01. //The tracer used is not recognized,
02. //the Console tracer will be used
03. tracer OtherTracer;
04.
05. @SuppressWarnings("deprecation")
06. class A {
07.   Integer id;
08.
09.   trace id;
10. }
11. // @@@skipphpcompile See issue 596 (PHP tracing causes issues
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //Using a supported tracer avoids
02. //the warning
03. tracer File;
04.
05. @SuppressWarnings("deprecation")
06. class A {
07.   Integer id;
08.
09.   trace id;
10. }
11. // @@@skipphpcompile See issue 596 (PHP tracing causes issues
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W901 Deprecated Keyword Constant

Umple warning reported when deprecated keyword "constant" is used.

The keyword "constant" is still supported in Umple interfaces due to legacy reasons. However, there is no guarantee that this keyword will be supported in the future. The keyword "const" should be used instead.

Example

```
view plain

01. // This causes the warning
02. interface Y {
03.   constant A=5;
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. // This does not cause the warning
02. interface Y {
03.   const A=5;
04. }
05.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E3500 Invalid Template Name

Umple semantic error raised when a template name is not valid

In Umple, a template name must be alphanumeric and start with a letter or the symbol '_'.

Example

```
view plain

01. //The name of the template is
02. //not valid and generates
03. //an error
04. class A {
05.   !template <<!output!>>
06. }
07.
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. //The name of the template
02. //is valid in the following
03. class A {
04.   template <<!output!>>
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E3501 Template Method Cannot Be Main

Umple semantic error raised when a template emit method uses the method name "main"

A template emit method cannot be called "main", as it would cause a conflict with the main method.

Example

view plain

```
01. //The emit method name
02. //cannot be main, the error
03. //is produced
04. class A {
05.   template <<!output!>>
06.   emit main()(template);
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //Using another method name
02. //solves the error
03. class A {
04.   template <<!output!>>
05.   emit method()(template);
06. }
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E3502 Template Name Cannot Be Resolved

Umple semantic error raised when a template name cannot be found

A template used in an emit method must exist within the context of the emit directive.

Example

```
view plain

01. //The template name cannot be
02. //found as it does not exist
03. class A {
04.   emit method()(template);
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. //The error is resolved by
02. //declaring the template
03. class A {
04.   template <<!output!>>
05.   emit method()(template);
06. }
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E3503 Template Reference Refers To Itself

Umple semantic error raised when a template reference refers to itself

A template reference cannot refer to itself. The logic of the template might be flawed.

Example

view plain

```
01. //Template otherTemp refers to
02. //itself causing the error
03. class A {
04.   temp <<! output !>>
05.   otherTemp <<! <<@otherTemp>> !>>
06. }
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //No template refers to itself,
02. //the error does not occur
03. class A {
04.   temp <<! output !>>
05.   otherTemp <<! <<@temp>> !>>
06. }
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E3504 Template Reference Cannot Be Resolved

Umple semantic error raised when a template reference cannot be resolved

When referencing a template, an error is thrown if the template cannot be found. The template name might be misspelled.

Example

view plain

```
01. //The template otherTemplate
02. //is not found, generating the
03. //error
04. class A {
05.   temp <<! <<@otherTemplate>> !>>
06.   emit method()(temp);
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //otherTemplate exists and can
02. //be referenced
03. class A {
04.   otherTemplate <<! output !>>
05.   temp <<! <<@otherTemplate>> !>>
06.   emit method()(temp);
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E3505 Template Reference Cycle

Umple semantic error raised when a template reference causes a cycle

A template reference cannot be used in a cyclically referring way, creating an indirect reference to itself. The issue might come from a flaw in the templates' design.

Example

view plain

```
01. //The error arises as temp refers
02. //to temp3, referring to temp2, which
03. //causes a cycle due to temp2 referring
04. //to temp
05. class A {
06.   temp <<! <<@temp3>> !>>
07.   temp2 <<! <<@temp>> !>>
08.   temp3 <<! <<@temp2>> !>>
09.
10.   emit method()(temp);
11. }
12.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //There is no longer a cycle,
02. //the error is resolved
03. class A {
04.   temp <<! <<@temp3>> !>>
05.   temp2 <<! <<@temp>> !>>
06.   temp3 <<! output !>>
07.
08.   emit method()(temp);
09. }
10.
```

Load the above code into classic UmpleOnline

Load the above code into UmpleOnline V2

W3506 Duplicate Template Name

Umple semantic warning reported when a class has a duplicate template name

A class should have unique template names for each distinct template. If duplicate templates are found, the last definition of the template is kept. The warning might be caused by a template of the same name being defined in a separate file.

Example

```
view plain

01. //Class A contains two templates
02. //of the same name, causing
03. //the warning
04. class A {
05.   temp <<! output !>>
06.   temp <<! otherOutput !>>
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. //The warning is resolved by
02. //giving the second template
03. //a distinct name
04. class A {
05.   temp <<! output !>>
06.   secondTemp <<! otherOutput !>>
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E3507 Duplicate Emit Method Name

Umple semantic error raised when a class has duplicate emit method names

Each emit method name must be unique in a class.
Attributing a different name to the duplicate method will solve the error.

Example

```
view plain

01. //Class A contains two emit
02. //methods of the same name
03. class A {
04.   temp <<! output !>>
05.   otherTemp <<! output !>>
06.
07.   emit aMethod()(temp);
08.   emit aMethod()(otherTemp);
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. //Changing the name of the
02. //second emit method resolves
03. //the issue
04. class A {
05.   temp <<! output !>>
06.   otherTemp <<! output !>>
07.
08.   emit aMethod()(temp);
09.   emit aMethod2()(otherTemp);
10. }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E3607 Missing Class in Association or Port Name Cannot Be Resolved

This error can occur in two situations: a semantic error raised when a port name cannot be resolved, or a syntax error caused by an invalid association

A port used in a class must be defined before usage by port bindings and active methods. This is shown in the first example below, with a resolution in the second example.

This error might also be raised if you intended to create an association from the current class to another, but omitted the destination class. This is shown in the third example. The solution would be to add another class and then give the name of that class as the destination of the association after the asterisk.

Example

view plain

```
01. //inB has not been defined
02. //in class A, causing the error
03. class A {
04.   public in Integer inA;
05.   public out Integer outA;
06.
07.   inB -> outA;
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //Both inA and outA are
02. //defined; the error does
03. //not occur
04. class A {
05.   public in Integer inA;
06.   public out Integer outA;
07.
```

```
08.   inA -> outA;
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Example where this an association with a missing Class that should have followed the *

view plain

```
01. //Destination class name is missing,
02. //causing the error
03. class X {
04.   1 -> *;
05. }
06.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E4500 Tag Not Closed Correctly

Umple semantic error raised when a FIXML tag is not closed properly

In FIXML, a start tag must be closed properly with its respective end tag. Elements must also be correctly nested within each other.

Example

view plain

```
01. //The tags are not properly
02. //nested, causing E4500
03. <FIXML>
04.   <tag>
05.     <othertag>
06.       </tag>
07.   </othertag>
08.  </FIXML>
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //Each tag is now correctly
02. //closed
03. <FIXML>
04.   <tag>
05.     <othertag>
06.     </othertag>
07.   </tag>
08.  </FIXML>
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W7002 Distributed Association To Non-Distributed Class

Umple semantic warning reported when a distributable class navigates to a class that is not distributable

A class marked *distributable* may have its objects in another virtual machine, so an association from that class is a remote reference. This warning is reported when such a class has an association that is navigable toward a class that is not distributable. The distributable class would then refer to an ordinary object that is not set up for remote access. An arrow pointing at the ordinary class, or a bidirectional association written in the distributable class, both cause the warning.

To avoid the warning, mark the other class *distributable* as well, as shown in the second example. If the first class does not need to be remote, remove *distributable* from it instead.

Example

view plain

```
01. // This example generates the warning message
02. class Supplier
03. {
04.   distributable;
05.   0..1 -> * Product;
06. }
07.
08. class Product
09. {
10. }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The following shows how to avoid the warning
02. class Supplier
03. {
04.   distributable;
```

```
  0..1 -> * Product;
}

class Product
{
  distributable;
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W1002-3 Unexpected Embedded Code

Umple warning reported when embedded code is found in a native language like Java, PHP or C++, yet 'strictness' has been set indicating this should not occur.

If strictness is set to 'modelOnly', this means that the developer intends to only include classes, associations, state machines and other modeling elements, but not embedded code in another programming language, not even method bodies.

If strictness is set to 'noExtraCode', it is the same as the above, except that method bodies are allowed.

This warning is issued when strictness has been set to one of the above, yet Umple has encountered sytax that interprets as either a method body or some other code in a class that it cannot parse, and is assuming it is native code to be passed to the native compiler.

This warning can often happen when the programmer/modeler wrote Umple code with a syntax error, so Umple thinks it is not Umple, but some other language. Here are some things to try in order to resolve this warning:

- Add spaces before and after Umple syntactic elements such as the -> or -- of an association.
- Check that you have terminated statements with semicolons where needed.
- Ensure your parentheses and quotes match everywhere. An extra or missing { or } can often cause this warning.
- Look at the examples of sytax in the user manual pages for such items as associations and state machines, and compare them to what you have provided.
- Check the grammar for the elements you are trying to specify in the user manual grammar page.

Example

view plain

```
01. // The following example shows how to
02. // generate this warning.
03. strictness modelOnly;
04. class X {
05.   public static void main() {
06.     System.out.println("Hello World");
```

```
  }
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E1004-5 Messages not as Expected

Umple errors reported when an expected message is not found or a disallowed message is found.

The strictness clause with 'expected' can be used to indicate that a certain message is expected. An error occurs when the expected warning or error does not actually occur.

Similarly, the strictness clause with 'disallowed' can be used to indicate that a certain message is not expected. An error occurs when the expected warning or error does occur.

W1006 State Machine Not Parsed

Umple semantic warning reported when a state machine could not be fully parsed and is treated as 'extra code'.

In Umple, elements of a class not recognized as valid Umple are assumed to be elements of the target programming language that are embedded in the Umple. However, this warning is raised when the Umple compiler has reason to believe that the developer might have been trying to specify a state machine, because the segment of code starts with something like sm {.

Since that sequence is not found in target languages, and since it is easy to make a mistake specifying states, substates, guards, or events, this message is generated. If you encounter this message and indeed intended to specify a state machine, look carefully at the state machine code.

The following are some common mistakes and things to check if you get this warning when writing a state machine

- Make sure the curly brackets match for each state machine, state, substate and action block.
- Make sure there are semicolons after transitions. In the example below, a semicolon is needed at the end of line 9 to declare the transition.
- In guards, make sure the expression is a valid condition (Boolean expression). For example, in the example below, the v = 5 should be v == 5.
- Make sure the order of items in transitions is valid. For example, in the example below the arrow should be after the guard, not before it.
- Make sure an element of the syntax is not missing. For example, in the example below, after the entry keyword there should be a slash / prior to the block of code to execute on entering the state. All such actions (exit actions and transition actions) also use the slash symbol, since this is the notation inherited from UML.

If you are still stuck when writing details of a state machine, comment out segments until you can narrow down the problem.

Example

view plain

```
01. // This example generates the warning
02. // on line 9 there is a missing semicolon
```

```
// on line 17 the = should be ==
// on line 24 the -> should be after the guard
// on line 30 there should be an / after entry
class X {
  sm1 {
    s1 {
      a -> sb
    }
    s2{}
  }

  Integer v = 0;
  sm2 {
    s3 {
      e1 [v = 5] -> s4;
    }
    s4 {}
  }

  sm3 {
    s5{
      e2 -> [v==1] s6;
    }
    s6{}
  }

  sm4 {
    s6 {
      entry {dosomething();}
      e3 -> s7;
    }
    s7 {}
  }
}
class X {
  sm {
    a -> b
    b -> c
  }
}


```

Load the above code into classic UmpleOnline

Load the above code into UmpleOnline V2

W1007 Element Not Parsed

Umple semantic warning reported when Umple finds an un-parsable element in a class, which is likely a malformed attribute, association, method or state machine declaration.

Umple emits such elements 'as is' (or what we call 'extra code') in its generated code, in case they are some part of the base language that Umple cannot understand.

Example

view plain

```
01. // The following will generate two such warnings
02. class X {
03.   blah blah blah;
04.   1 - * X;
05. }
06.
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W1008 Method Not Parsed

Umple semantic warning reported when a method could not be fully parsed and is treated as 'extra code'. The method will still be emitted and can be called, but Umple cannot analyse it.

In Umple, elements of a class not recognized as valid Umple are assumed to be elements of the target programming language that are embedded in the Umple. However, this warning is raised when the Umple compiler has reason to believe that the developer might have been trying to specify a method, but omitted some aspect.

Note that currently, Java main methods generate this warning. It can be safely ignored in that context, and the warning will eventually be removed.

Example

view plain

```
01. // This example generates the warning
02. // because there is a missing return type
03. class X {
04.   m() {}
05. }
06.
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W1009 Getter Setter Method Name Conflict

Umple semantic warning reported when a class redefines a getter or setter method on a declared attribute

Umple generates a setter and getter for most attributes, and will issue a warning if these methods are redefined. The attempt to redefine the method will be ignored. Developers can either rename the redefined method or else use a before or after statement if they want to add functionality to the getter or setter method.

Example

view plain

```
01. //The getAttr method is already
02. //generated by the declaration of
03. //the attribute attr and generates
04. //the warning
05. class A {
06.   attr;
07.   getAttr() {/* Implementation */};
08. }
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //Modifying the name of the
02. //method can fix the warning
03. class A {
04.   attr;
05.   returnAttr() Java {/* Implementation */};
06.   returnAttr() Python {''' Implementation '''};
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W1010 Constraint Syntax Could Not Be Processed

Umple semantic warning issued when the syntax of a constraint cannot be processed

When a constraint-like syntax is found but it cannot be processed by Umple, it is treated as extra code and left as-is. The constraint might not be constructed properly.
The issue could come from unmatched parentheses or from an incorrect usage of operators.

Example

view plain

```
01. //The warning is raised twice in
02. //the following as both constraints
03. //are not properly constructed
04. class A {
05.   Integer a;
06.   Integer b;
07.   Integer c;
08.
09.   [(a > b)) && a < c]
10.   [c < a | b > a]
11. }
12.
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //The warning is not generated
02. //by fixing the constraint syntax
03. class A {
04.   Integer a;
05.   Integer b;
06.   Integer c;
07.
08.   [(a > b) && a < c]
```

```
09.     [c < a || b > a]
10.   }
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W1011 Invalid Association Class Key

Umple semantic warning reported when an Association Class key is missing a required member.

For Association Classes with explicit keys, each class that participates in the relationship should be declared as a member of that key.

Example

view plain

```
01. // This example generates the warning
02. // because one participating class is
03. // missing from the key
04. class Passenger {}
05.
06. class Flight {}
07.
08. associationClass Booking {
09.   Integer number;
10.   * Passenger;
11.   * Flight;
12.   key {number, flight}
13. }
14.
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // This example shows how to remove
02. // the warning by completing the key
03. class Passenger {}
04.
05. class Flight {}
06.
07. associationClass Booking {
08.   Integer number;
09.   * Passenger;
10.   * Flight;
```

```
  key {number, flight, passenger}
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W1012 Method Not Found Injection

Umple semantic warning issued when an injection is called on a method that cannot be found

Injecting code before or after a method must be done on an existing method. The injection will otherwise be ignored.

Example

view plain

```
01. //getAttr does not exist in the class
02. //which generates the warning
03. class A {
04.   before getAttr {
05.     doSomething();
06.   }
07.
08.   doSomething(){/* Implementation */}
09. }
10. // @@@skippythoncompile
11.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //By declaring the attribute attr
02. //the getAttr method can now be used
03. class A {
04.   attr;
05.
06.   before getAttr {
07.     doSomething();
08.   }
09.
10.   doSomething() {/* Implementation */}
11. }
12. // @@@skippythoncompile - Contains Java Code
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W1013 Parameter Specification Does Not Apply

Umple semantic warning issued when parameters are specified on a code injection that does not permit it

Parameter specification on code injection does not apply to certain generated methods.
The code injection will be applied to all generated methods.

Example

```
view plain

01. //The before statement refers to
02. //the getAttr method specifying no
03. //parameters, causing the warning
04. class A {
05.   attr;
06.
07.   before getAttr() {
08.   }
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

01. //The method is referred to without
02. //parameters in the following, avoiding
03. //the warning
04. class A {
05.   attr;
06.
07.   before getAttr {
08.   }
09. }
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W1014 Excluded Method Not Found Injection

Umple semantic warning issued when an excluded method is not found in a code injection

When excluding methods in a code injection with pattern matching, a warning will be raised if a method is not found. The exclusion will be ignored.
Verifying which method was meant to be excluded will generally resolve the issue.

Example

```
view plain

//The warning is generated as the code
//injection excludes getAttr3, which
//is not a method in class A
//The exclusion is ignored
class A {
  attr;
  attr2;

  before getAttr*, !getAttr3 {
  }
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

//Fixing the name of the
//method removes the warning
class A {
  attr;
  attr2;

  before getAttr*, !getAttr2 {
  }
}

```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E15xx Parsing Error

Umple error messages reported when the Umple parser can't interpret the code.

These messages occur in several contexts:

- There is a typographical error in one of the directives specified outside a class declaration. Umple does not recognize the keyword.
- The syntax (i.e. arguments) to one of the directives such as 'generate' or 'use' is not properly typed.
- There is a missing semicolon at the end of a directive, or a missing space separating elements of the syntax.
- The overall structure of an umple element does not conform to the Umple grammar. For example specifying a class without a name, or without a body in curly brackets.
- Some text cannot be parsed at all, such as a stray <<! that opens no template. Error 1500 then shows the rest of the line where parsing stopped.

W1016 Unmatched or Extra Brackets

Umple semantic warning issued when a extra bracket is added

When either a extra curly or extra round bracket is found. A warning will be raised.
Warning will show line number and removing the extra bracket will resolve the issue.

Example

```
view plain

01. //The warning is generated as the code
02. // has a extra bracket on top (Could be open or close)
03. {
04. class A {
05.   attr;
06.   attr2;
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

```
view plain

01. //The warning is generated as the code
02. // has a extra bracket on top (Could be open or close)
03. (
04. class A {
05.   attr;
06.   attr2;
07. }
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. //Remove the extra bracket
02. //Removes the warning
03. class A {
04.   attr;
05.   attr2;
06. }
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W1510 Use File Missing

Umple warning reported when a use statement refers to a file that cannot be loaded.

The argument to a use statement must be a valid .ump file found in the same directory as the file with the use statement, or in its parent directory, or in a subdirectory named lib. It can also be a builtin Umple library file (prefixed by lib:) or an https URL pointing to an umple file on the Internet.

Check the spelling of the filename, and check file permissions.

E1511 Inline Mixset Is Not Supported

Error reporting that most inline mixsets must have curly brackets around their content.

An inline mixset is one in a class, interface or trait and where the mixset name is not followed by block denoted by curly brackets. Most of the time to specify a mixset in a class you just specify the keyword mixset, followed by the name of the mixset, followed by the block of items to be included (e.g. attributes, methods, state machines, associations).

However Umple allows inline mixsets for certain entities in Umple. Inline mixsets allow for nested declarations of classes (to create subclasses), traits, and interfaces. However, empty code classes, interfaces, and traits are not allowed for inline mixsets.

If you encounter this error, you can easily add a pair of curly brackets around the code that a mixset introduces. The code bellow shows how to use inline mixsets.

Example

view plain

```
01. /*
02. Example to show how to define inline mixsets.
03. */
04.
05. // The mixset M1 has been defined as inline mixset. this because
06. mixset M1 class A {
07.   isA B;
08.   isA C;
09.   x;
10. }
11.
12. // M1 also is inline mixset because it has been followed by a trait
13. // The usuall definition of M1 is:
14. /*
15. mixset M1
16. {
17.   trait B
18.   {
19.     y;
20.   }
21. }
22. */
```

```
23. mixset M1 trait B
24. {
25.   y;
26. }
27.
28.
29. // M2 is an inline mixset that defines an interface.
30. mixset M1 interface C { void method(); }
31.
32.
33. class D {
34.   // M3 adds an attribute for class D. It's an inline mixset since th
35.   // However, Umple parser understands this implicit definition. Th
36.   mixset M3 {attr1;}
37.
38.   }
39.
40.
41. //The line below will cause error since the interface E is empty.
42. //Mixset M4 interface E {}
43.
44. use M1; use M2; use M3;
45.
46.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W1512 Code Labels Inside Method Are Not Unique

Warning reporting that code labels inside methods should have unique names.

Code Labels allow aspects to inject code in specific locations inside method bodies. However, the code injection is only applied to the first encountered label. Therefore, a code label should not be reused multiple times (inside a method).

If you encounter this warning, you can easily rename code labels with unique names for each label inside the method body.

Example

view plain

```
01. class CodeLabelExample
02. {
03.   void methodToDoSomething()
04.   {
05.     //code
06.     Label1: ;
07.     int x=0;
08.     //code
09.     Label1: ;
10.     x++;
11.   }
12.
13.   // aspect injection in a code label.
14.   after  Label1:methodToDoSomething
15.   {
16.    int y =0;
17.   }
18. }
19. // @@@skipcompile - injects code in impossible place
20.
21.
22.
23.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W1513 Mixset Used With No Declaration

Warning reporting that a mixset use statement tries to activate a mixset that has not been declared.

If you encounter this warning, you can easily comment out the use statement that causes this warning, or you can declare the mixset with an empty body.

Example

view plain

```
01. // There are two options to get rid of this warning:
02. //option1: comment out the use statement below:
03. use M1;
04. 
05. // option2: remove the comment below to declare M1 with empty
06. //mixset M1 { }
07. 
08. 
09. 
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W1514 Require Statement Not Satisfied

Warning reporting that the use statements in the model do not satisfy a require statement.

A require statement constrains which mixsets may be used together. This warning is raised when the use statements violate such a constraint: for example, a required mixset has no use statement, or two mixsets that exclude each other are both used. A require statement at the top level of a file always applies; a require statement inside a mixset applies only when that mixset is used. The same check sets the Configuration Status shown in a feature diagram to Valid or Not Valid.

To get rid of this warning, change the use statements so that they satisfy the require statement, or change the require statement itself.

Example

view plain

```
01. // DesignA and DesignB exclude each other, so using both of then
02. // does not satisfy the feature model and raises the warning.
03. mixset DesignA {
04.   require [not DesignB];
05.   class X { name; }
06. }
07.
08. mixset DesignB {
09.   require [not DesignA];
10.   class X { age; }
11. }
12.
13. // To get rid of the warning, use only one of the two designs.
14. use DesignA, DesignB;
15.
16.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W20xx Error in Embedded Code

Umple warning issued when a base language compiler encounters a non-umple error

Under development: Umple can have code in any programming language embedded in method bodies or other constructs. This warning is issued when the compiler for the embedded language reports an error. This can occur if the -c option is supplied to the Umple compiler, telling it to invoke the base language compiler after the Umple is generated.

E9210 Python Unsupported

Python generation does not support a feature the class uses yet

A few Umple features are generated for Java but not yet for Python. When a class uses one of them, the Python generator reports this error, which names the feature, and does not write a Python file for the class. Other classes are still generated, and generation for other languages is not affected. The Python page lists the features concerned.

To generate Python, leave the feature out of the classes you generate as Python, or put it in a mixset that is only used for Java.

Example

view plain

```
01. // Tracing is not available for Python yet, so generating
02. // Python from this model reports E9210 and writes no Lamp.py
03. generate Python;
04. class Lamp {
05.   Integer level;
06.   trace level;
07. }
08.
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // Without the trace directive the class is generated
02. generate Python;
03. class Lamp {
04.   Integer level;
05. }
06.
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E9211 Python Untranslatable Code

Untagged code cannot be translated to Python

Code written without a language tag in an action, an entry, exit or do activity, a code injection, a derived attribute or an initial value is Java. For Python, Umple translates such code when it keeps to the subset of Java listed on the Python page. Code outside that subset, for example a call to the Java library or to a method the model does not define, is reported with this error, which names the code and shows how that kind of code is written in Python. The class is then not generated.

The error is also reported for a derived attribute written only for other languages, which would have no value in Python, and for an interface constant whose value is not a literal, which Umple does not translate to Python.

Give the code a Python version: add a Python block beside the Java one, or move the code into a method with a Python body and call that method. Method bodies and extra code are never translated: an untagged method body is taken to be Python already.

Example

view plain

```
01. // The default value calls the Java library, which cannot be
02. // translated to Python, so Python generation reports E9211
03. generate Python;
04. class Session {
05.   String user = System.getProperty("user.name");
06. }
07.
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // The value comes from a method with a Java body and a
02. // Python body, so each language uses its own library
03. generate Python;
```

```
04. class Session {
05.   String user = currentUser();
06.   String currentUser() Java { return System.getProperty("user.na
07.     Python { import getpass; return getpass.getuser() }
08. }
09.
10.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

W9212 Python No Python Code

Code has versions only for other languages

A method, action, entry, exit or do activity, or code injection has code only for languages other than Python (for example a Java body), and no untagged code. The Python generator has no code to use for it, so it leaves it out of the generated Python and this warning names it.

Add a Python version, or untagged code, if the Python classes need that behaviour.

Example

view plain

```
01. // describe has a Java body only, so the generated Python
02. // leaves it out and W9212 names it
03. generate Python;
04. class Book {
05.   title;
06.   String describe() Java { return "Book " + getTitle(); }
07. }
08.
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // With a Python body the method is generated for Python too
02. generate Python;
03. class Book {
04.   title;
05.   String describe() Java { return "Book " + getTitle(); }
06.     Python { return "Book " + self.getTitle() }
07. }
08.
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E9213 Python Module Name Clash

A class or namespace is named like a standard Python module

Each class becomes a Python module named after its namespace and class, and Python resolves the first part of that name before anything else. That part must not be the name of a standard library module (such as datetime, json, os, queue or email). A class without a namespace named like one would hide the standard module from the whole program, generated code included, or could not be imported when the standard module is built into the interpreter or already loaded. A namespace named like one could not be imported: Python finds the standard module before the generated directory. The Python generator reports the class or interface and does not generate it.

Rename the class, or use a namespace whose first part is not a standard module name (`app.sys` is fine).

Example

view plain

```
01. // A class without a namespace named like a Python standard
02. // module that generated code imports hides that module: E9213
03. generate Python;
04. class datetime {
05.   Integer day;
06. }
07.
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // In a namespace the class gets its own package, so the
02. // standard module stays reachable
03. generate Python;
04. namespace planner;
05. class datetime {
06.   Integer day;
07. }
```

```
08.
09.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E9214 Python Self Parameter

A method or event has a parameter named self

Generated Python methods name the object they are called on `self`, so a parameter cannot also be named `self`. The Python generator reports this error at the method (for an event, at its class) and does not write a Python file for the class. Rename the parameter.

Example

view plain

```
01. // Python reserves self for the object a method is called on
02. generate Python;
03. class Shape {
04.   Double scale(Double self) Python { return self * 2 }
05. }
06.
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // Any other parameter name works
02. generate Python;
03. class Shape {
04.   Double scale(Double factor) Python { return factor * 2 }
05. }
06.
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E9215 Python Reserved Name

A name that Python reserves or that generated code uses

The Python generator reports this error, and does not generate the class, for three kinds of name:

- A class or interface named like a Python built-in that generated code calls, such as `len`, `id`, `set` or `tuple`: it would hide the built-in in its module and in every method that imports it.
- A class, interface, method, event, state, constant, enumeration or enumeration value whose generated name is a Python keyword, such as `lambda`, `async`, `break` or `None`. Enumeration values are written with a capital first letter, so `none` becomes `None`.
- A parameter named `__class__`, the name generated methods use for their own class.

Rename the element.

Example

```
view plain

// len is a Python built-in that generated code calls,
// so a class named len would hide it: E9215
generate Python;
namespace shapes;
class len {
  Double value;
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

// Another name avoids the clash
generate Python;
namespace shapes;
class Length {
  Double value;
}
```

```
07.
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E9216 Python Duplicate Parameter

Two parameters get the same Python name

The Python generator renames a parameter named like a Python keyword: `in` becomes `input`, and other keywords get a trailing underscore (`lambda` becomes `lambda_`). When two parameters of a method then have the same name, it reports this error at the method (for an event, at its class) and does not write a Python file for the class. Rename one of the parameters.

Example

view plain

```
01. // A parameter named in becomes input in Python (in is a
02. // Python keyword), which the second parameter already uses
03. generate Python;
04. class Copier {
05.   String copy(String in, String input) Python { return input }
06. }
07.
08.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // Distinct Python names for the parameters
02. generate Python;
03. class Copier {
04.   String copy(String source, String input) Python { return input }
05. }
06.
07.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E9217 Python Package Name

A namespace is not a valid Python package name

A class in a namespace is generated as a module of the package named by the namespace, so each part of the namespace must be made of ASCII letters, digits and underscores, must not start with a digit, and must not be a Python keyword. Namespaces such as `releases.2024` or those taken from URLs cannot be imported, so their classes are reported and not generated. Use a namespace made of such names. A namespace whose first part is a standard Python module is reported with E9213.

Example

```
view plain

// Python packages are identifiers, so a namespace segment
// such as 2024 cannot be imported: E9217
generate Python;
namespace releases.2024;
class Release {
  name;
}
// @@@skipcompile -
 Java packages cannot start with a digit either


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

```
view plain

// A namespace made of identifiers is a valid Python package
generate Python;
namespace releases.y2024;
class Release {
  name;
}


```

Load the above code into classic UmpleOnline

Load the above code into UmpleOnline V2

E9218 Python Path Clash

Two namespaces or classes would share a Python package or module

Each namespace becomes a Python package folder and each class a module file. Python matches the case of module names even where file names ignore case, as on Windows and macOS. There, namespaces such as Shop and shop would share one folder, and classes such as `Item` and `item` in one namespace would share one file, so one of them could not be imported. Python also imports a module rather than a namespace of the same name, so a class Shop and a namespace Shop in the same place cannot both be used. The Python generator reports the class or interface generated later and does not generate it.

Spell each namespace the same way everywhere, or give one of the classes or namespaces another name.

Example

view plain

```
01. // The two namespaces differ only in case, so where file names
02. // ignore case they would share one folder: E9218
03. generate Python;
04. namespace Shop;
05. class Item {
06.   name;
07. }
08.
09. namespace shop;
10. class Order {
11.   Integer number;
12. }
13.
14.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Solution to The Above So the Message No Longer Appears

view plain

```
01. // One spelling of the namespace gives one package
```

```
02. generate Python;
03. namespace shop;
04. class Item {
05.   name;
06. }
07.
08. namespace shop;
09. class Order {
10.   Integer number;
11. }
12.
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

E9x00 Compiler Error

Umple compiler error raised when an internal crash occurs

Occasionally the Umple compiler may crash with certain code input, causing an error in the parsing phase (E9000), analysis phase (E9100) or generation phase (E9200). As soon as the Umple development team becomes aware of such a situation, we try to deal with it with a high level of priority.

In order to ensure the Umple development team is aware of the problem, please report the issue, including the code that generated the error and the stack trace that accompanied the message.

Page Being Developed

You most likely encountered an error or warning in UmpleOnline, and clicking on the message displayed this page. The web page explaining that message has not yet been created. In the meantime, you can search the manual or browse the links on the left.

W9999 Feature Under Development

Umple warning reported a feature is used that is not yet fully implemented in Umple

Features are often added to the syntax of Umple *before* they are added to the semantics and code generation. This is to ensure that adding the feature will not break existing code. If you receive this warning, then it means you are using such a feature. No code is yet generated from this feature, or else the generated code is incomplete.

RE001 Violation of Immutability

Umple runtime exception thrown when Immutability is violated in a constructor

The immutable keyword in Umple can be applied to a class, an attribute or an association to ensure that they can only be set in the constructor, and cannot be modified again subsequently. The immutability of associations means that all associated objects must be specified as unique constructor arguments. Violation of this is detected by the constructor, in which case a RuntimeException is thrown. This can be caught by a try-catch block.

In the example below, an immutable Path has immutable pathElements (instances of Point2D). The Path instances are of an immutable class too (here Point2D), and the only way to construct this association is in arguments to the constructor. But it is necessary to avoid duplicate objects in the constructor and a runtime exception is thrown when this constraint is violated. So, a runtime exception is thrown here when an attempt is made to create a Path with duplicate instances of Point2D.

Example

view plain

```
01. // This demonstrates code that will throw the
02. // immutability exception
03. class Point2D
04. {
05.   immutable;
06.   Float x;
07.   Float y;
08. }
09.
10. class Path
11. {
12.   immutable;
13.   1 -> * Point2D pathElements;
14.     public static void main(String[] argv) {
15.
16.       Point2D p2d = new Point2D(0,0);
17.
18.
19.       // This will throw an exception since the
20.       // pathElements cannot be duplicate
21.       try {
22.         Path p = new Path(p2d,p2d);
```

```
23.       }
24.       catch (RuntimeException re) {
25.         System.out.println("Exception Caught: "+re+"\n");
26.       }
27.     }
28.   }
29.
30.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

RE002 Violation of Association Multiplicity

Umple runtime exception thrown when multiplicity is violated in a constructor

Umple manages multiplicity by blocking operations that would allow too many or too few instances of a class to be linked. There are two ways in which links can be set, hence violating multiplicity: a) in a constructor, or b) after construction using an ordinary method. In a case of a constructor that would violate multiplicity, a RuntimeException is thrown. This can be caught by a try-catch block. After construction, any method that attempts to violate the multiplicity will return false.

In the first example below a student is always required to have a program. A runtime exception is thrown when an attempt is made to create a Student without specifying a required program. A method that attempt to violate multiplicity, and hence returns false, is shown at the end of the first example.

The second example shows a runtime exception that is thrown in the case of a one-to-one association. Since for such associations there must always be equal numbers of both associated classes, and objects of the two classes must be created in pairs, the constructor of either class also creates instances of the other class. An attempt to call such a constructor with a null argument will throw a runtime exception.

One to many example

view plain

```
01. // This demonstrates code that will throw the
02. // multiplicity exception
03. class Program {
04.   title;
05.   1 -- * Student;
06. }
07.
08. class Student {
09.   name;
10.
11.   public static void main(String[] argv) {
12.
13.     Student s1 = null;
14.     Program p1 = null;
15.
16.     // This will throw an exception since there must
```

```
    // be at least one program
    try {
      s1 = new Student("Abel", null);
    }
    catch (RuntimeException re) {
      System.out.println("Exception Caught: "+re+"\n");
    }

    // After we create a program we can repeat the
    // attempt.
    p1 = new Program("Computer Science");
    s1 = new Student("Abel", p1);

    // Attempting to change the program back to null
    // will not throw an exception but will result
    // in null being thrown
    if(!s1.setProgram(null)) {
      System.out.println("Could not set program "+
       "to null as would violate multiplicity \n");
    }
  }
}


```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

One to one example

view plain

```
// This demonstrates code that will throw a
// multiplicity exception in  1--1 associations
class StockSymbol {
  code;
  1 -- 1 Company;
}

class Company {
  name;

  public static void main(String[] argv) {

```

```
13.     Company c1 = null;
14.     StockSymbol s1 = null;
15.
16.     // This will throw an exception since the stock
17.     // symbol cannot be null
18.     try {
19.       c1 = new Company("Umpleco", s1);
20.     }
21.     catch (RuntimeException re) {
22.       System.out.println("Exception Caught: "+re+"\n");
23.     }
24.
25.     // This will succeed: Umple 1--1 constructors
26.     // create both objects at once
27.     c1 = new Company("Umpleco","UMPC");
28.
29.     s1 = c1.getStockSymbol();
30.     System.out.println("The company: "+
31.       c1+"\n"+
32.       "The stock symbol: "+
33.       s1+"\n");
34.   }
35. }
36.
37.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

RE003 Violation of Uniqueness

Umple runtime exception thrown when uniqueness is violated in a constructor

The unique keyword in Umple can be applied to an attribute to ensure that each instance of the class in the currently running program has a unique value for that attribute. The code generated by Umple will not allow this constraint to be violated.

If an attempt is made to call the constructor of a class with an argument that would result in a violation of the uniqueness constraint, an exception will be thrown. The text of the constraint will start with the phrase "Cannot create".

In the example below,

- s1 is first initialized with number set to 1.
- Then an attempt is made to initialize s2 with number also set to 1. This throws the exception, but it is caught using a try-catch block.
- Next s3 is successfully created with number 2, but an attempt is made to change the number to 1 using the method call setNumber(). This returns false, since it is only constructors that throw the exception.
- Finally, an attempt is made to create s4, with the number set to 1. This throws the exception without a try-catch block, thus the program terminates.

Programs should be designed with logic that avoids violation of uniqueness. It is not recommended that production code relies on catching this exception as a normal part of its logic.

Example

view plain

```
01. // This demonstrates code that will throw the
02. // uniqueness exception
03. class Student {
04.   unique Integer number;
05.   name;
06.
07.   public static void main(String[] argv) {
08.
09.     Student s1 = null;
10.     Student s2 = null;
```

```
    Student s3 = null;
    Student s4 = null;

    System.out.println("Initial\ns1= "+s1+
      "\ns2= "+s2+"\ns3= "+s3+"\n\n");

    // We set up the first student s1 with no problem
    s1 = new Student(1,"Abel");

    // For the second student we use a try catch
    // block to catch the exception
    // Ideally code would avoid creating violations
    // to start with, but the following
    // approach can be used as a safety mechanism.
    try {
      s2 = new Student(1, "Baker");
    }
    catch (RuntimeException re) {
      System.out.println("Exception Caught: "+re+"\n");
    }

    // We now create a student that doesn't
    // violate uniqueness
    s3 = new Student(2,"Charlie");

    // Uniqueness can also be detected by trying
    // to change a value
    // This does not throw the exception, since
    // the setNumber() method returns false
    boolean result = s3.setNumber(1);
    if(!result) System.out.println(
      "Could not change number as it would have"+
      " violated uniqueness\n");

    System.out.println("Final\ns1= "+s1+
      "\ns2= "+s2+"\ns3= "+s3+"\n\n");

    // The following will throw the exception
    // without being caught
    s4 = new Student(1, "Delta");

    // This statement will not be reached
    System.out.println(
      "This line should not be reached");
  }
```

```
56. }
57.
58.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Basic Testcase

Testcase can be embed within the Umple model and will be generated when using the *Test* generator. A testcase can be embedded at the class-level or the model-level.

A testcase canonsists of the following elements:

Initialization: ~identidier ~name = ~value;

- ex: ~identifier ~objectname (~Parameters);
- Person p ("name", 345);

Action: ~identifier ~name = ~value;

- | ~name = ~value ;
- Integer number = 12345;
- number = student.numberOfMentors();

Assertions: ~assertion (~assertionCode)

- assertTrue(p.setName("Jane"));
- - assertTrue (bool)
- - assertFalse (bool)
- - assertEqual (value1 , value2)
- - assertNull (value)
- - assertAttribute (attributeName) , to be presence in generated code
- - assertMethod (methodName) , to be presence in generated code

Example

view plain

```
01. class Student {
02.    id = "ID123";
03.    name = "John";
04.
05.
06.    test checkSetId {
07.    Student s ();
08.
09.    assertTrue(s1.setId("ID432"));
10.
11.    }
12. }
13.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. class Student {
02.     id;
03.     name;
04.
05.     test checkSetId {
06.     Student s ("98CS","Jane");
07.
08.     assertEqual(s.getId(),"98CS");
09.
10.     s.setName("Roger");
11.     assertFalse(s.getName() == "Jane");
12.
13.     }
14. }
15.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Automatic Test

Test can be automatically generated by compiling the Umple model with the test generator. This can be done using the following commandline

- `java -jar umple.jar -generate Test umpleModel.ump`
  When compiled this way, Umple will generate a list of files with the extension *.umpt* you can compile these abstract test files with the unit test generator targeting one of the platforms supported. This can be done using the following commandline tool:
- `java -jar umple.unit-test.jar -generate testModel.umpt -l JUnit`
  Using the automatic test generator for the following example will generate the following files:
- **testModel_ModelTest.umpt** contains test logic for associations
- **AdvisorTest.umpt**
- **StudentTest.umpt**
- **CourseTest.umpt**
- **OfficeTest.umpt**

Example

view plain

```
01. class Advisor
02. {
03.   name;
04.   id;
05.   int tel;
06. 
07.   0..1 -- * Student;
08. 
09.   1 -- * Office;
10. 
11.   2 -- * Course;
12. 
13. }
14. 
15. 
16. class Student {
17.   name;
18. 
19. }
20. 
```

```
21.
22. class Office {
23.   name;
24.   lazy address;
25.
26.
27.
28. }
29.
30.
31. class Course {
32.   code;
33.   int numberOfStudents;
34.   description;
35.
36.   Date startDate = "2020-12-26";
37.
38.   Time classTime = "12:59:59";
39.
40.
41. }
42.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Test with Mixset

Tests can be integrated with mixsets to enhance feature-based testing. This allows the user to toggle feature-specific tests using a mixset dedicated for testcases that can be switched on using the *use mixsetTest;* statement. The following example shows how to create a mixset for testing another mixset. Note the tests can also be embedded in the same mixset or can be defined in a separate one. The only issue that need to be handled is that dependency between mixsets. If your test case contains an instantiated object that uses attribute in a particular mixset then either embed the tests within the mixset to avoid errors or make sure you always switch-on the targeted feature to avoid instantiation errors. Currently, this is not caught by the compiler but will be added.

Example

view plain

```
01. class Student {
02.   id;
03.   name;
04.
05.
06. }
07.
08.
09. mixset Job {
10.   class Student {
11.   job;
12. Integer  jobRank = 1;
13.
14.     void promote () {
15.      this.setJobRank(this.jobRank + 1);
16.     }
17.
18.   }
19.
20. }
21.
22.
23. mixset JobTest {
24.
25.   class Student {
26.     test checkPromotion {
27.       Student s1 ("ID123", "John", "Director");
```

```
28.
29.       assertTrue(s1.getJobRank() == 1);
30.
31.       s1.promote();
32.
33.       assertTrue(s1.getJobRank() == 2);
34.
35.     }
36.
37.   }
38.
39.
40. }
41.
42. use Job;
43. use JobTest;
44.
45.
46.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Generic Testing

Generic testcase is an Umple testcase that is defined as *generic*. This allows you write a test template that has access to the metamodel in order to propagate the template against a number of elements those match a certain pattern. For instance, you can write a generic test that can be generated for each attribute that is string and has a certain prefix 'studentXXX' or a suffix such as 'XXXId'. You can also use regular expression to match attribute name. We can this a *fix* in the template. Currently, generic tests can be generated for: attributes, methods and associations. Attibutes can be matched using name and type, methods can be matched using name and exact order of parameters and association are matched based on the type of associations and given specfic fixes to handle association variable propagation.

Example

view plain

```
01. class Student {
02.
03.    String studentId;
04.    nameMemebershipId;
05.    address;
06.    studentAge;
07.
08.
09.   generic test checkifLogged(String attribute.regex(\w*Age)){
10.        Student st1 ( "S1425", "John", "Ottawa",20) ;
11.        String valueToCheck = st1.get<<attribute>>();
12.
13.        assertTrue(valueToCheck > 18);
14.
15.
16.        }
17.
18. }
19.
20.
21.
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
01. class Calculator{
02.
03.
04.       1--* Number;
05.
06.       0..1->* Controller;
07.
08.
09.   generic test checkifLogged(0..1->* association){
10.       Calculator c1 () ;
11.       Number n1(c1);
12.       Number n2(c1);
13.       Controller cn1();
14.       Controller cn2();
15.
16.       c1.addNumber(n1);
17.       c1.addNumber(n2);
18.
19.       c1.addController(cn1);
20.       c1.addController(cn2);
21.
22.       String numberInList = c1.numberOf<<association.toSingu
23.
24.       if(c1.numberOfNumbers() > 2)
25.       {
26.          assertTrue (numberInList > 2);
27.       }
28.
29.       else {
30.          assertTrue (numberInList < 2)
31.       }
32.
33.       }
34.
35. }
36.
37.
38.
39. class Number {
40.
41.
```

```
}


class Controller {


}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2

Another Example

view plain

```
class Calculator{

   Integer x;
   Integer y;
   String someString;

 Integer  returnInteger (Integer x) { return x+y;}

 String  returnStirng (Integer x) { return someString;}


 generic test checkifLogged(Integer method Integer){
      Calculator c1 ( 4, 5) ;
      String valueToCheck = p1.get<<method>>();
      ps1.getValue(<<method>>);
      boolean isLogged =  p1.checkIsLogged(valueToCheck);
      assertTrue(logged == "true");
      }
  generic test checkifLogged(String method Integer){
      // code omitted
    }
}
```

Load the above code into classic UmpleOnline
Load the above code into UmpleOnline V2