Umple User Manual

Pdf version
Release 1.30.2, April 23, 2021

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 is a tool for creating models of software and generating code for the software in Java, PHP, C++ and other languages. Umple's language is textual, but many kinds of diagrams can be generated, including state machines, class diagrams, feature diagrams, entity-relationship diagrams, and so on. Class diagrams can also be edited, in order to update the text. Umple can be used online, in Docker, as a command-line tool or in an IDE like Microsoft Visual Studio Code or Eclipse. Code, such as methods, in the target languages such as Java is embedded directly in the Umple code, enabling direct generation of systems from the Umple code.

This document is the pdf version of the Umple user manual. Most people will use the online html version of the manual, available at https://manual.umple.org, however some people prefer a printable document, or to have a document they can cite. This document has exactly the same content as the online version, and links will take online users of the pdf to the online manual. All examples in the manual have links that open the example code in UmpleOnline at https://try.umple.org.

Umple is an open-source project, developed by over 60 students at the University of Ottawa and elsewhere since 2007. The Umple compiler is written in itself. Umple is hosted on Github, at http://code.umple.org. 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 those 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, and user experience in software development.

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 complete set of publications can be found at http://publications.umple.org.

Reference

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

Getting Started

Umple is a technology for Model-Oriented Programming.

The Umple home page is http://www.umple.org. To download Umple for the command line or Eclipse, go to http://dl.umple.org. To use Umple online as a web app, go to http://try.umple.org

Umple allows you to do the following things:

1. **Create UML models textually**. Rather than drawing diagrams, it can often be faster to create UML class diagrams and state machines 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 UML diagram) or to edit a UML diagram (and see changes appear in the textual code)

2. **Add UML modeling constructs directly into your programs, when you are programming in Java, PHP, C++ or Ruby**. For example, when programming in Java, you can directly add a UML state machine 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.

3. **Generate high quality code from UML models**. The Umple compiler creates code in languages like Java, 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.

4. **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 contributer, 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

Hello World Examples

As is customary when introducing a new language, here are some 'hello world' examples for Umple. Load these examples into UmpleOnline by clicking on the links. Then generate Java code by clicking on the 'Generate' button. Next click on the 'download zip file' link and run 'javac' on the result or else compile it in Eclipse.

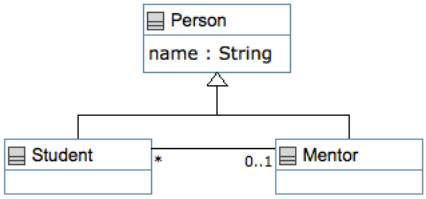
First example below: This looks identical to how a Java 'hello world' example would look, illustrating a key feature of Umple: Umple *adds* features to existing languages: Code in the original language can and does remain the same. Umple just replaces and simplifies some (or a lot) of it.

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 student can get its mentors, and the mentor can find his or her optional student.
- 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.



Example Showing a Simple Class with a Main Method

```
view plain

/*
 * Simple Hello World example for Umple.
 * Compile this with Umple and it will generate Java
 * that is essentially the same.
 *
 * You could just as readily compile this code directly
 * with javac. However, this serves as the starting point:
 * Other examples in this manual show other things you
 * can do with Umple
 */
class HelloWorld {
  public static void main(String [ ] args) {
```

```
    System.out.println("Hello World");
  }
}


```

Load the above code into UmpleOnline

Example Showing Three Classes with an Association and Attributes (Diagram is Above)

```
view plain

/*
 * Introductory example of Umple showing classes,
 * attribute, association, generalization, methods
 * and the mixin capability. Generate java 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]
 */
class Person {
  name; // Attribute, string by default
  String toString () {
    return(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) {
    Mentor m = new Mentor("Nick The Mentor");
    Student s = new Student("Tom The Student");
    s.setMentor(m);
    System.out.println("The mentor of "
      + s  + " is " +  s.getMentor());
    System.out.println("The students of "
      +  m  + " are " +  m.getStudents());
  }
}


```

Load the above code into UmpleOnline

Umple Tools

Umple files conventionally use the extension .ump. To create a model or a system you would use one or more of the following tools to view, edit and compile .ump files.

- **UmpleOnline.** 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 code as well as several other 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.
- **Eclipse.** To use Umple with Eclipse, you need to load an Umple Eclipse plugin. This can be obtained from our downloads site.

- **Command-line based compiler** You can always use Umple from the command line, typically from an ant script. Simply download the latest officially released version of the umple jar, and run it as "java -jar umple.jar *.ump".

- For the adventurous, the bleeding-edge continuous build version of the Umple command-line compiler can be obtained here.

- **Plugin for Microsoft Visual Studio Code**. This plugin allows basic editing and compilation of Umple in Visual Studio Code.

- **Plugin for Sublime Text**. Try this Umple Plugin if you use Sublime Text.

More details on downloading, installing and running Umple can be found here.

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: Re-usable 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.

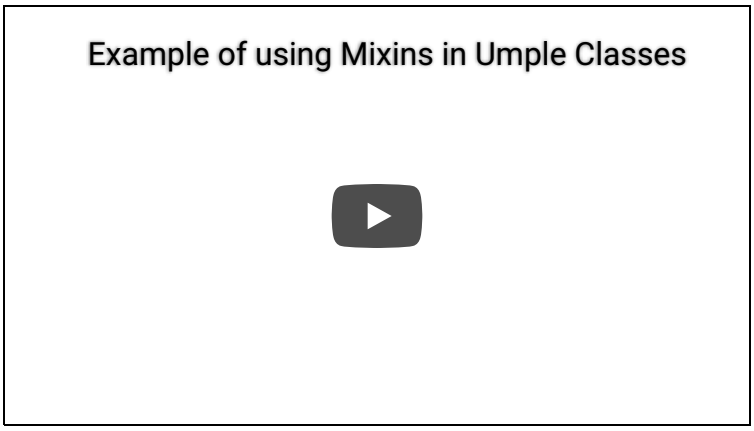
The fact that Umple allows for multiple definitions to be added to create a complete definition, also means that you can create *mixins*. A mixin is a file that has some definitions that can be added to add extra features to a system. You can therefore organize your system, in whole or in part, by *feature*. 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



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- : [[glossary]]
   | [[generate]]
   | [[distribute]]
   | [[generate_path]]
   | [[filter]]
   | [[useStatement]]
   | [[namespace]]
   | [[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 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 compliable and functional as you gradually increase your uptake of Umple.

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

2. **Fully open source with a liberal license:** If you dont like something about Umple you can fix it and contribute to the community. But there are no restrictions on what you do with Umple or its generated code since it uses an MIT-style license.

3. **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.

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

```
1.  // The Umple system is both fun
2.  // and efficient to development with
3.
4.
5.
```

Load the above code into UmpleOnline

Example Showing Both Inline And Multiline Comments

view plain

```
1.  // The Umple system is both fun and
2.  // efficient to development with
3.
4.  /*
5.    This apple can only be compared to
6.    other apples
7.  */
8.
9.
```

Load the above code into UmpleOnline

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
// Umple is both fun and efficient to develop with
```

Load the above code into UmpleOnline

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 UmpleOnline

Syntax

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

Types of Directives

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

- Use statements result in inclusion of 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 us 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- : [[glossary]]
   | [[generate]]
   | [[distribute]]
   | [[generate_path]]
   | [[filter]]
   | [[useStatement]]
   | [[namespace]]
   | [[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
   |Php
   |RTCpp
   |SimpleCpp
   |Ruby
   |Cpp
   |Json
   |StructureDiagram
   |Yuml
   |Violet
   |Umlet
   |Simulate
   |TextUml
   |Scxml
   |GvStateDiagram
   |GvClassDiagram
   |GvFeatureDiagram
   |GvClassTraitDiagram
   |GvEntityRelationshipDiagram
   |Alloy
   |NuSMV
   |NuSMVOptimizer
   |Papyrus
   |Ecore
   |Xmi
   |Xtext
   |Sql
   |Umple
   |UmpleSelf
   |USE
   |Test
   |SimpleMetrics
   |Uigu2] ( [=suboptionIndicator:-s
   |--suboption] " [**suboption] " )* ;

generate_path : generate [=language:Java
   |Php
   |RTCpp
   |SimpleCpp
   |Ruby
   |Cpp
   |Json
   |StructureDiagram
   |Yuml
```

```
   |Violet
   |Umlet
   |Simulate
   |TextUml
   |Scxml
   |GvStateDiagram
   |GvClassDiagram
   |GvClassTraitDiagram
   |GvFeatureDiagram
   |GvEntityRelationshipDiagram
   |Alloy
   |NuSMV
   |NuSMVOptimizer
   |Papyrus
   |Ecore
   |Xmi
   |Xtext
   |Sql
   |UmpleSelf
   |USE
   |test] " [**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 or program entities (i.e. classes) within your current model or to include optional blocks of code that are specified within mixsets.

A model file or mixset will only be included once, subsequent "use" commands for the same file will be ignored.

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 work in a manner similar to 'include' directives in other languages.

Example

```
view plain
use Core.ump;
```

Load the above code into UmpleOnline

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 UmpleOnline

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.
```

```
19.
20. // To redefine the namespace of an
21. // entity, use the --redefine option.
22. // B is now in namespace n.
23.
24. namespace n --redefine;
25. interface B{}
26.
27.
28. // If --redefine is not used, a warning will
29. // be issued when trying to redefine the
30. // namespace of an entity
31.
32. namespace p;
33. class C{}
34.
35.
36. // Using namepace -; will deactivate the last
37. // declared namespace. There will not be an
38. // attempt to redefine the namespace of interface
39. // B and a warning will not be issued
40.
41. namespace -;
42. interface B{}
43.
```

Load the above code into UmpleOnline

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 UmpleOnline

Example 4: Automatically redefined namespace

```
view plain
31. // Classes A is in the
32. // default namespace. Class B is in
33. // namespace m. Both are
34. // linked to each other via an association,
35. // therefore, and because they are not all
36. // in the same namespace, an import text
37. // should be generated. However, many programming
38. // languages do not support import from
39. // the default namespace so the namespace of
40. // A will become m because B is in m.
41.
42. class A{}
43. namespace m;
44. class B{*--* A;}
45. namespace -;
46.
47.
48. // The same issue occurs when an
49. // entity in a non-default namespace
50. // extends or implements an entity
51. // in the default namespace
52.
53. interface E{}
54. interface F{}
55. namespace n;
56. interface J{isA E;}
57. class K{isA F;}
58.
59.
60. // Here an import text will be generated for A in X
61. // because A is in a different namespace
62. // This feature is not yet supported for interfaces
63. // that extend interfaces in other namespaces
64. // and the import text will not be generated for K in Y
65.
66. namespace p;
67. class X{isA A;}
68. // interface Y{isA E;}
69.
```

Load the above code into UmpleOnline

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 can't 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 UmpleOnline

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

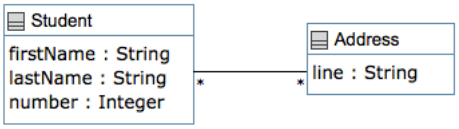
A class definition defines an object-oriented class available for use as a type in the system you are developing. Objects (chunks of data) are created as *instances* of the class. A class describes the structure of that data in terms of attributes (simple data like strings and numbers), associations (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 UmpleOnline

YouTube Video with Additional Explanation

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]]
   | [[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

```
interface RegisterCapable
{
  depend school.util.*;

  boolean registerForCourse(Course aCourse);
}

class Person {
  name;
}


class CorporateClient {
  isA RegisterCapable;
  boolean registerForCourse(Course aCourse) {
    // write code here
  }
  0..1 <- * Person employees;
}

class IndividualStudent {
  isA Person, RegisterCapable;
  boolean registerForCourse(Course aCourse) {
    // write code here
  }
}


class Course
{
  name;
  description;
  * -- * Person registrants;
}

```

Load the above code into UmpleOnline

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]]
   | [[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 (i.e. a Supervisor could be descrbied 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 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 themselves contains other items that can be in classes. These allow

for conditional compilation and are activated by use statements elsewhere.[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]]
  | [[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]]
| [[traitDefinition]]
| [[trace]]
| [[position]]
| [[displayColor]]
| [[abstract]]
| [[keyDefinition]]
| [[softwarePattern]]
| [[depend]]
| [[symmetricReflexiveAssociation]]
| [[attribute]]
| [[stateMachine]]
| [[inlineAssociation]]
| [[concreteMethodDeclaration]]
| [[abstractMethodDeclaration]]
| [[constantDeclaration]]
| [[invariant]]
| ;
| [[exception]]
| [[extraCode]]

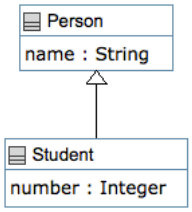
isA clause

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

// A superclass-subclass relationship defined using the isA keyword
class Person
{
  name;
}

class Student
{
  isA Person;
  Integer number;
}

```

Load the above code into UmpleOnline

The Same Model as Above but Using Nesting Instead of IsA

```
view plain

// A superclass-subclass relationship defined
// by embedding the subclass in the superclass
class Person
{
  name;

  class Student
  {
    Integer number;
```

```
  }
}

```

Load the above code into UmpleOnline

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

```
namespace sports.core;

class Equipment
{
  name;
  Integer weight;
}

namespace sports.baseball;

class Baseball
{
  depend sports.core.*;

  public boolean isRequired(Equipment eq)
  {
    return eq.getName().equals("bat");
  }
}
```

Load the above code into UmpleOnline

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,
```

```
  // so the settable keyword can be omitted.
  settable Date d;

  // internal: No getter and setter methods created.
  // Only for private use in internal methods. However
  // in this case it is initalized using =  The code
  // following = is in the 'base' language, here Java.
  internal Time t2 = new Time(
    System.currentTimeMillis());

  // unique: Every new object created must have a
  // unique new value assigned.
  // You can get the value. You can set it as well,
  // but it has to be unique.
  unique u;

  // autounique: Every new object created will have a
  // new integer assigned.
  // You can get the value (an integer) but not set it.
  autounique x;

  // The value is initialized as shown in the constructor.
  // There is no constructor argument generated.
  String q = "chicken";

  // defaulted: Set in the constructor to the default,
  // and can be reset to the default any time by
  // calling a reset method (here resetP()).
  // Can also be set to any other value using setP().
  // The default can be queried by calling getDefaultP().
  defaulted String p = "robot";

  // Similar to the above, except this shows that if
  // no type is given, then the default type is String.
  defaulted r = "";
}
```

Load the above code into UmpleOnline

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

```
class Demo
{
  Integer i;
  Integer i2 = 5;

  String str;  // the default if no type is specified
  str2;
  str3 = "";

  Float flt;

  Double dbl;
  Double dbl2 = 8.0;

  Boolean bln;
  Boolean bln2 = false;

  Date dte; // In Java. uses the java.sql.Date class
  Date dte2 = "2018-09-25";

  Time tme; // In Java. uses the java.sql.Time class
  Time tme2 = "14:56:50";
}
```

Load the above code into UmpleOnline

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

view plain

```
class DemoNonUmpleType
{
  // If you use a non-umple type that is specific to a base
  // programming language, then code will be generated
  // consistent with that type, but generation of code
  // independent of base language is no longer possible
  int demoJavaInt;
}
```

Load the above code into UmpleOnline

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;
23.   country;
24. }
25.
```

Load the above code into UmpleOnline

Alternative to the Above that Uses an Association

view plain

```
01. // Contrast this example with the previous one
02. // Here we use a directed association instead of
03. // 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 address;
}

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

Load the above code into UmpleOnline

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 UmpleOnline

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.
```

Load the above code into UmpleOnline

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 UmpleOnline

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.
```

Load the above code into UmpleOnline

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.
```

Load the above code into UmpleOnline

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

```
// Every player has a ranking.
// Rankings can be modified,
// but two different players
// can not have the same ranking.

class Player
{
  unique Integer ranking;
}
```

Load the above code into UmpleOnline

Example 2

view plain

```
// Unique registration numbers (Integer)
// are automatically assigned to every
// new registration created

class Registration
{
    autounique registrationNumber;
}
```

Load the above code into UmpleOnline

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

// In this example, the "Status" enumeration
// is defined for the "Student" class

class Student {
  enum Status { FullTime, PartTime }
  Status status;
}
```

Load the above code into UmpleOnline

Example of Enumeration Shared by Two Different Classes

```
view plain

// In this example, the "Status" enumeration
// is shared by the "GradStudent"
// and "UndergradStudent" class

enum Status { FullTime, PartTime }

class GradStudent {
  Status status;
}

class UndergradStudent {
  Status status;
}
```

Load the above code into UmpleOnline

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

```
view plain
```

```
// In this example, the "Status" enumeration
// is shared by the "GradStudent"
// and "UndergradStudent" class
// The "Semester" enumeration is only defined
// for the "UndergradStudent" class

enum Status { FullTime, PartTime }

class GradStudent {
  Status status;
}

class UndergradStudent {
  enum Semester { Spring, Summer, Fall, Winter }
  Status status;
  Semester semester;
}

```

Load the above code into UmpleOnline

Example with Enumerations Defined Within and Outside of Class Bodies

```
view plain

// In this case, the "Month" enumeration
// defined in class A will be used in
// the generated code

namespace example;

enum Month {x,y,z}

class A{
  enum Month {o,p,q}
  Month m;
  Month p;
}

```

Load the above code into UmpleOnline

Syntax

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

Association Definition

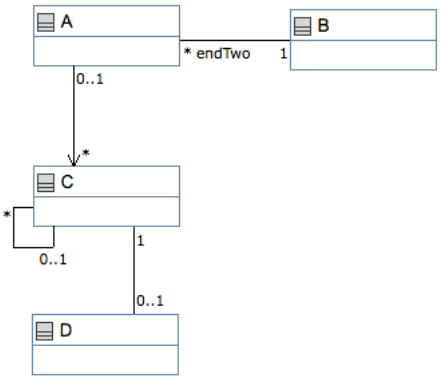
An association defines a relationship from a class to another class. More specifically, it defines which links (i.e. references or pointers) may exist at run time between instances of the classes.

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

```
// Example code illustrating various
// kinds of associations
class A {}

// Class with inline association having role name
class B {
  1 -- * A endTwo;
}

// Class with reflexive association
class C {
  0..1 -- * C;
  1 -- 0..1 D; // D is external
}

// Independently defined and directed association
association {
  0..1 A -> * C;
}

// Class with composition
class E {
  0..1 e <@>- * A a;
}

// Reference to a class defined elsewhere
external D {}

```

Load the above code into UmpleOnline

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

```
// When multiplicity is given as *, which is the
// same as 0..* there can be any number of links
// from an instance at the other end of the
// association to instances at this end
//
// The lower bound is zero and the upper bound is
// 'many'
class A
{
  // an instance of A has many B's
  1 -- * B;

  // An instance of C has many A's
  * -- 1 C;
}

class B {} class C {}
```

Load the above code into UmpleOnline

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{}

```

Load the above code into UmpleOnline

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.
//
// Here, there must be exactly two objects (lower
// and upper bound are both 2)
class F {
  0..1 -- 2 G;
}

class G{}

```

Load the above code into UmpleOnline

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

class Group
{
  // a many-to-many association
  // An item has zero or more groups and a group
  // has zero or more items
  * -- * Item item;

  // An item has an optional description
  // The association is directed, so descriptions
  // do not know which groups link to them
  1 -> 0..1 Description;
}

class Item
{}

class Description
{}

```

Load the above code into UmpleOnline

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
class Group { }
class Item {}
class Description {}

association {
  * Group -- * Item;
}

association {
  1 Group -> 0..1 Description;
}

```

Load the above code into UmpleOnline

Syntax

// Associations can be declared outside the body of classes.
// See user manual page IndependentlyDefinedAssociations
associationDefinition : association [name]? { ([[comment]] | [[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 UmpleOnline

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.

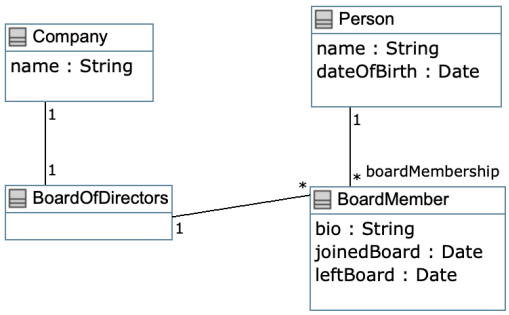
Allowing a lower bound of zero means that one object can be created without having to first create the object(s) it is to be linked to. The vast majority of associations have a lower bound of zero at at least one of the two ends.

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
// Core class of the example
// A company is a corporation
class Company {
  name;
```

```
  // A company must have a BoardOfDirectors
  // that must be created at the same time
  // as the Company itself
  1 -- 1 BoardOfDirectors;
}

// BoardOfDirectors is in a 1-1 relationship
// with Company meaning that theoretically it
// could just be merged into the Company class
// However to facilitate separation of concerns
// it is sometimes best to keep such classes
// separate
class BoardOfDirectors {
  // A BoardMember is just a member of one
  // board (but a person can be on multiple
  // boards using multiple BoardMembers)
  1 -- * BoardMember;
}

// 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");

```

```
53.     BoardOfDirectors b = c.getBoardOfDirectors();
54.
55.     b.addBoardMember("Largest Shareholder", p1);
56.     b.addBoardMember("Founder", p2);
57.
58.     // Output the results to prove that this works
59.     System.out.println("Key company info: "+c);
60.     System.out.println("Board members: "+
61.       b.getBoardMembers());
62.
63.   }
64. }
65.
```

Load the above code into UmpleOnline

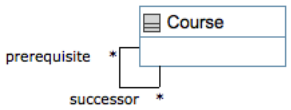
Reflexive Associations

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

```
// Example asymmetric association with role names
// on both ends
class Course {
  * successor -- * Course prerequisite;
}
```

Load the above code into UmpleOnline

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 UmpleOnline

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 UmpleOnline

Symmetric association

```
view plain

// Example symmetric association. Note the use of
// the keyword self
class Course {
  * self  isMutuallyExclusiveWith;
}
```

Load the above code into UmpleOnline

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:

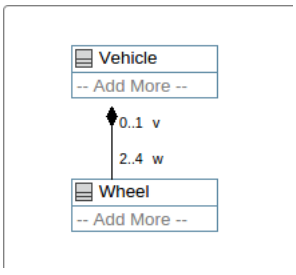
See **Example 2**

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
class Vehicle {}

class Wheel {}

association {
    0..1 Vehicle v -- 2..4 Wheel w;
}

```

Load the above code into UmpleOnline

Example 2

```
view plain
class Vehicle {}

class Wheel {}

// vehicle composed with wheels
association {
    0..1 Vehicle v <@>- 2..4 Wheel w;
}

```

Load the above code into UmpleOnline

Example 3

```
view plain
class Vehicle {
   0..1 v <@>- 2..4 Wheel w;
}

class Wheel {}

```

Load the above code into UmpleOnline

Example 4

```
view plain
class Vehicle {}

class Wheel {
  2..4 w -<@> 0..1 Vehicle v;
}

```

Load the above code into UmpleOnline

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

```
// Here we define our base classes
class Vehicle {}
class Wheel {}

// Here we define our subclasses
class Bicycle { isA Vehicle; }
class Unicycle { isA Vehicle; }

// This is the "parent association" so to speak.
// It defines the Vehicle to Wheel relationship
// with as much abstraction as possible.
association {
  0..1 Vehicle vehicle -- 0..* Wheel wheel;
}

// Here, we'd like to extend the functionality of
// the previous association to work with
// subclasses of Vehicle, while utilizing as much
// of the existing code as possible.
// Specializations allow us to do this, since:
// -> Bicycle and Unicycle extend Vehicle;
// -> Wheel is part of the Wheel hierarchy;
// -> The bounds on the left are not less specific
//    than the left bound of the parent
//    association;
// -> The bounds on the right are not less
//    specific than the right bound of the parent
//    association;
// -> Finally, the role names are the same.
```

```
30.  association
31.    { 0..1 Bicycle vehicle -- 0..2 Wheel wheel; }
32.  association
33.    { 0..1 Unicycle vehicle -- 0..1 Wheel wheel; }
34.
35.
36.
```

Load the above code into UmpleOnline

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
```

```
// thus many-to-many.
//
// There is, however, data to record about each
// ticket. This can be  recorded as an
// association class
//
// Note the following doesn't currently render
// in UmpleOnline using Correct UML
// association class notation.
// There are plans to fix this.
associationClass Ticket
{
  Integer ticketNumber;
  Double price = 0.0;

  * Person attendee;
  * Seminar;
}

class Person
{
  name;
}

class Seminar
{
  Date when;
  address;
}

```

Load the above code into UmpleOnline

Example 2

```
view plain

// The following shows how we might model sales of
// Ticket for Seminars to Persons without using an
// association class. Note, however, that in this
// model, it would be possible to sell more than
// one ticket to a Person for the same Seminar.
// Using an association class would hence be
// better than this.
class Ticket
{
  Integer ticketNumber;
  Double price = 0.0;

  * -- 1 Person attendee;
  * -- 1 Seminar;
}

class Person
{
  name;
}

```

```
class Seminar
{
  Date when;
  address;
}

```

Load the above code into UmpleOnline

Example 3

```
view plain

// 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 UmpleOnline

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 UmpleOnline

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 UmpleOnline

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]]
| [[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

```
// This example demonstrates two cases of sorted
// associations. The main program adds items out
// of order. But when they are printed the output
// will be sorted
class Academy {
  1 -- * Course;
  1 -- * Student registrants sorted {id};
}

class Student {
  Integer id;
  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);
  }
  public String toString() {
    String result="Students:\n";
    for (Student s: getRegistrants()) {
      result +=s + "\n";
    }
    result +="Courses:\n";
    for (Course c: getCourses()) {
      result +=c + "\n";
    }

    return result;
  }
}

class Student {
  public String toString() {
    String result="Student=" + id + "["
      + name + "\n";
    for (Registration r: getRegistrations()) {
      result +="  In: " + r + "\n";
    }
    return result;
  }
}

class Course {
```

```
  public String toString() {
    String result ="Course=" + code + "\n";
    for (Registration r: getRegistrations()) {
      result +="  Has: " + r + "\n";
    }
    return result;
  }
}
```

Load the above code into UmpleOnline

Example of sorted associations serialisation in Java

view plain

```
// 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());
    }

    Academy ac2 = null;

    // Deserialization
    try
    {
      FileInputStream file = new FileInputStream(filename);
      ObjectInputStream in = new ObjectInputStream(file);
      ac2 = (Academy)in.readObject();
      in.close();
      file.close();
    }
    catch(Exception ex)
    {
      System.out.println(ex.getMessage());
    }

    // Adding new elements and comparing
    try
    {
    Student m = ac2.addRegistrant(8,"Mary");
    Student f = ac2.addRegistrant(3,"Francois");
    }
    catch(Exception ex)
    {
      System.out.println(ex.getMessage());
    }

    System.out.println("After Serialization");
    System.out.println(ac2);

  }

  public String toString() {
    String result="Students:\n";
    for (Student s: getRegistrants()) {
      result +=s + "\n";
    }
    return result;
  }
}

class Student {
  public String toString() {
    String result="Student=" + id + "["
      + name +"]"+ "\n";
    return result;
  }
}
```

Load the above code into UmpleOnline

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

```
// The following demonstrates two simple
// constraints limiting the range of age
// The test code demonstrates that this works
// as expected.
class Client {
  const Integer minAge =18;
  Integer age;
  [age >= minAge]
  [age <= 120]

  public static void main(String [ ] args) {
    Client c = new Client(40);
    for (int i: new int[]
      {-1,10,17,18,19,50,119,120,122,1000})
    {
      System.out.println(
        "Trying to initialize age to "+i);
      System.out.println(c.setAge(i)
        ? "  Success"
        : "  FAILURE");
    }
  }
}
```

Load the above code into UmpleOnline

Example with more operators

view plain

```
// Example with a few more operators
class X {
  Integer i;
  Integer j;
```

```
  [i > i]
  [i < 5 && i > 0]
  [! (i == 10)]
}

```

Load the above code into UmpleOnline

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.   }
.0. }
.1.
.2.
```

Load the above code into UmpleOnline

Syntax

```
//queuedMethodDeclaration- : queued [[methodDeclarator]] [[methodThrowsExceptions]]? [[methodBody]]
methodBody- : ( [[codeLangs]] { ( [[precondition]]
   | [[postcondition]]
   | [[assertion]]|[[testCase]] )* [**code] } )+

// 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]] ]
```

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.

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
.0.   [model: isA X]
.1. }
.2.
.3. class Z {
.4.  a;
.5.  1 -- * X;
.6.  // Verifies that there is an association a
.7.  [model: has a]
.8. }
.9.
```

Load the above code into UmpleOnline

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
01. class Airline
02. {
03.   singleton;
04. }
05.
```

Load the above code into UmpleOnline

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
// A simple example of an immutable class
class Point2D
{
  immutable;
  Float x;
  Float y;
}
```

Load the above code into UmpleOnline

Immutable class with reference to another class

```
view plain
// An example of one immutable class making reference to another

class Path
{
  immutable;
  1 -> * Point2D pathElements;
}

class Point2D
{
  immutable;
  Float x;
  Float y;
}
```

```
.5.
```

Load the above code into UmpleOnline

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 UmpleOnline

Syntax

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])? ;

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

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

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

```
// This system has many delegation methods so that information in one class can
// be obtained in a neighbouring class easily. The delegation is created using
// derived attributes, which generate get methods for use by other code.
class Airline {
  code;
  name;
  1 -- * RegularFlight;
}

// A regular flight flies every day and has a flight number and other fixed data.
class RegularFlight {
  Integer number;
  Time depTime;
  * outgoing -- 1 Airport origin;
  * incoming-- 1 Airport destination;
  fullNumber = {getAirline().getCode()+getNumber()}
}

// A specific flight actually has passengers
class SpecificFlight {
  * -- 1 RegularFlight;
  Date depDate;

  // Four delegation attributes to save multiple methods in this
  // class from having to traverse the association
  flightNumber = {getRegularFlight().getFullNumber()}
  Airport origin = {getRegularFlight().getOrigin()}
  Airport destination = {getRegularFlight().getDestination()}
  Time plannedDepTime = {getRegularFlight().getDepTime()}
```

```
}

// A passenger on a flight
class Booking {
  lazy seat;
  * -- 1 SpecificFlight;
  * -- 1 Passenger;

  // Second-level delegation - data comes from RegularFlight
  flightNumber = {getSpecificFlight().getFlightNumber()}
  Airport destination = {getSpecificFlight().getDestination()}

  // Simple delegation
  passengerName = {getPassenger().getName()}
  Time plannedDepTime = {getSpecificFlight().getPlannedDepTime()}

  //Simple method using the data gathered by delegation
  public String toString() {
    return getFlightNumber()+" "+getPlannedDepTime()+" "+
      getDestination().getName()+" "+getPassengerName()+" "+getSeat();
  }
}

class Passenger {
  lazy Integer freqFlyerNumber;
  name;
}

class Airport {
  code;
  name;
 }


// Mixin with main program to test the above
class Airline {
  public static void main(String [ ] args) {
    Airport ott = new Airport("YOW","Ottawa");
    Airport tor = new Airport("YYZ","Toronto Pearson");
    Airline a = new Airline("AO","Air Ottawa");
    RegularFlight r = a.addRegularFlight(100,java.sql.Time.valueOf("13:12:00"),ott,tor)
    SpecificFlight f = r.addSpecificFlight(java.sql.Date.valueOf("2017-03-15"));
    Passenger p = new Passenger("John");
    Booking b = new Booking(f,p);
    System.out.println(b);
  }
}
```

Load the above code into UmpleOnline

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
class Sport {
  name;
  description;
  code;
  key { code }
}

class League {
  name;
  id;
  geographicalArea;
  * -- 1 Sport;
  Date seasonStart;
  Date seasonEnd;
  key { id }
}

class Team {
  name;
  * -- 1 League;
}

class Player {
  name;
  Integer id;
  key { id }
 }

class PlayerOnTeam {
  Integer year;
  * -- 1 Player;
  * -- 1 Team;
  key { year, player, team }
}
```

Load the above code into UmpleOnline

Syntax

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

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

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

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

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.

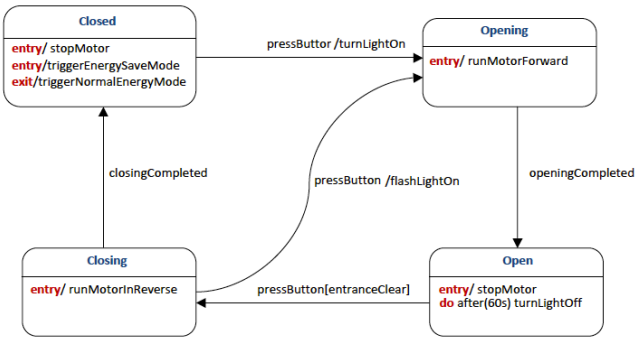
Within the block, you list the names of the various states. Each state is followed by a block, again indicated using '{' to start and '} to end. This block defines the details of a state.

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
```

```
// by various events.
//
// Open, Closing, Closed, Opening and HalfOpen are
// the possible values, or states, of status.
//
// buttonOrObstacle, reachBottom and reachTop are
// events. These become generated methods that can
// be called to cause a state change.
//
// To make the state diagram appear in
// UmpleOnline, select 'Options' then choose
// 'GraphViz State Diagram'
class GarageDoor
{
  status {
    Open { buttonOrObstacle -> Closing;  }
    Closing {
       buttonOrObstacle -> Opening;
       reachBottom -> Closed;
    }
    Closed { buttonOrObstacle -> Opening; }
    Opening {
       buttonOrObstacle -> HalfOpen;
       reachTop -> Open;
    }
    HalfOpen { buttonOrObstacle -> Opening; }
  }
}

```

Load the above code into UmpleOnline

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 {
  lazy 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;
  }
}

boolean runMotorInReverse() {
   System.out.println(
     "Running motor in reverse");
  return true;
}

boolean flashLightOn() {
   System.out.println("Flashing light on");
  return true;
}

boolean turnLightOn() {
   System.out.println("Turning light on");
  return true;
}

 boolean turnLightOff() {
   System.out.println(
     "Turning light off");
  return true;
}

boolean runMotorForward() {
   System.out.println(
     "Running motor forwards");
  return true;
}

boolean triggerEnergySaveMode() {
   System.out.println(
     "Triggering Energy Saving Mode");
  return true;
}

boolean stopMotor() {
   System.out.println("Stopping motor");
  return true;
}

boolean triggerNormalEnergyMode() {
   System.out.println(
     "Triggering Normal Energy Mode");
      return true;
}

boolean waitawhile() {
  System.out.println("Waiting");
  return true;
}

boolean test() {
       System.out.println("Testing");
```

```
31.         return true;
32.     }
33. }
34.
```

Load the above code into UmpleOnline

YouTube Video with Additional Explanation

Syntax

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

//Issue 148
inlineStateMachine : [=queued]? [=pooled]? [~name] { ([[comment]]
| [[state]]
| [[trace]]
| [[mixsetDefinition]]
| [=ll]
| [[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- : [=ll]
| [[mixsetDefinition]]
| [[entryOrExitAction]]
| [[autoTransition]]
| [[transition]]
| [[activity]]
| [[state]]
| [[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 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
namespace example;

class LightFixture
{
  bulb
  {
    On {
      entry / { doEntry(); }
      exit / { doExit(); }
      push -> /{ doTransition(); } Off;
      do { doThisContinuouslyWhileOn(); }
    }
    Off {}
  }

  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 UmpleOnline

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 UmpleOnline

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

// Illustration of nested states. Event e1 will
// take the system into state s2, and its first
// substrate s2a. Event e2 directly goes to s2b.
// Event e3 toggles between s2b and s2a.
// Event e1 when in s2, goes to s1

class A {
  sm {
    s1 {
      e1 -> s2;
      e2 -> s2b;
    }
    s2 {
      e1 -> s1;
      s2a {
        e3 -> s2b;
      }
      s2b {
        e3 -> s2a;
      }
    }
  }
}

```

Load the above code into UmpleOnline

Concrete example of nested state machines

```
view plain

// Example showing nested states.

class Course {
  code;
  description;
  1 -- * CourseSection;
}

class CourseSection
{
  // Example from Lethbridge and Laganiere: Object
  // Oriented Software Engineering: Practical
```

```
  // Software Development using UML and Java
  // McGraw Hill, 2005  www.lloseng.com

  sectionId;
  Integer classSize = 0;
  Integer minimumClassSize = 10;
  Integer maximumClassSize = 100;

  // State machine controlling status
  status
  {
    Planned {
      openRegistration -> NotEnoughStudents;
    }
    Open {
      cancel -> Cancelled;
      NotEnoughStudents {
        closeRegistration -> Cancelled;
        register
          [getClassSize() > getMinimumClassSize()]
          -> EnoughStudents;
      }
      EnoughStudents {
        closeRegistration -> Closed;
        register
          [getClassSize() > getMaximumClassSize()]
          -> Closed;
      }
    }
    Cancelled {}
    Closed {}
  }

  boolean requestToRegister(Student aStudent)
  {
    register();
    return setClassSize(getClassSize()+1);
  }
}

class Student {
  //Remainder of this class defined elsewhere
}
```

Load the above code into UmpleOnline

Concrete example of state machines without nesting

```
view plain

// Example with an opportunity to nest states, to
// avoid repeating transitions

class Course {
  code;
  description;
  1 -- * CourseSection;
```

```
}

class CourseSection
{
  // Example from Lethbridge and Laganiere: Object
  // Oriented Software Engineering: Practical
  // Software Development using UML and Java
  // McGraw Hill, 2005  www.lloseng.com

  sectionId;
  Integer classSize = 0;
  Integer minimumClassSize = 10;
  Integer maximumClassSize = 100;

  // State machine controlling status
  status
  {
    Planned {
      openRegistration -> OpenNotEnoughStudents;
    }
    OpenNotEnoughStudents {
      closeRegistration -> Cancelled;
      cancel -> Cancelled;
      register
        [getClassSize() > getMinimumClassSize()]
        -> OpenEnoughStudents;
    }
    OpenEnoughStudents {
      closeRegistration -> Closed;
      cancel -> Cancelled;
      register
        [getClassSize() > getMaximumClassSize()]
        -> Closed;
    }
    Cancelled {}
    Closed {}
  }

  boolean requestToRegister(Student aStudent)
  {
    register();
    return setClassSize(getClassSize()+1);
  }
}

class Student {
  // Remainder of class defined elsewhere
}
```

Load the above code into UmpleOnline

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

class TrackShuttler {
  sm {
    initializing {
      readyToGo -> transferringLoad;
    }
    transferringLoad {
      loaded -> shuttling;
    }
    shuttling {
      reachEnd -> transferringLoad;
      moving {
        nearEnd -> braking;
        accelerating {
          reachedMaxSpeed -> coasting;
        }
        coasting {
          tooSlow -> accelerating;
        }
        braking {
          tooSlow -> coasting;
        }
      }
      ||
      controllingLights {
        lightsOn {
          daylightDetected -> lightsOff;
        }
        lightsOff {
          darknessDetected -> lightsOn;
        }
      }
    }
  }
}

```

Load the above code into UmpleOnline

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 ll 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

// Using the keyword "Final" automatically defines the
// final state for the top-level state machine "sm"

class X {
  sm {
    s1 {
      goToS2 -> s2;
    }
    s2 {
      goToFinal -> Final;
    }
  }
}

```

Load the above code into UmpleOnline

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 { }
    }
  }
 }
}

```

Load the above code into UmpleOnline

Example of non-concurrent state machines with "final" states

```
view plain

// "sm" will be considered as complete when
// it is in either state "s3" or "s5"

class X {
 sm {
   s1 {
     s2 {
       goToS3 -> s3;
       goToS4 -> s4;
     }
     final s3 { }
   }
   s4 {
     goToS5 -> s5;
     final s5 { }
   }
 }
}

```

Load the above code into UmpleOnline

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

// In this example, the system will transition to s2 automatically
// when the do activity ends
class X {
  sm {
    s1 {
      entry / {
        System.out.println("Starting first sleep");
      }
      do {
        Thread.sleep(2000);
        System.out.println("Ending first sleep");
      }
      -> s2;
    }
    s2 {
      entry / {
        System.out.println("Starting second sleep");
      }
      do {
        Thread.sleep(2000);
        System.out.println("Ending second sleep");
      }
    }
  }
  public static void main(String [ ] args) {
    X x = new X();
  }
}


```

Load the above code into UmpleOnline

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

```
//A garage door can be modeled
//using unspecified events for
//error states

class GarageDoor {
  status {
    functionning {
      unspecified -> error;

      closed {
        pressButton -> opening;
      }
      open {
        pressButton -> closing;
      }
      closing {
        finishedClosing -> closed;
      }
      opening {
        finishedOpening -> open;
      }
    }

    error {
      entry / {
        /* attempt to solve the issue */
      }
      ready -> functionning;
    }
  }
}
```

```
32.
```

Load the above code into UmpleOnline

Another Example

```
view plain

01. //The state machine sm makes use
02. //of the unspecified event to reach
03. //specific error states to complete
04. //auto-transitions
05.
06. //If the machine is idle and receives
07. //a non-handled event, the destination
08. //state is error 1, while if the sm was
09. //in state active, the machines reaches
10. //error 2
11.
12. class AutomatedTellerMachine{
13.
14.   queued sm{
15.
16.     idle {
17.       cardInserted -> active;
18.       maintain -> maintenance;
19.       unspecified -> error1;
20.     }
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;
```

```
    }

    error2{
      ->validating;
    }

  }
  void readCard() {/*Code would be written here*/}
  void ejectCard() {/*Code would be written here*/}

}
```

Load the above code into UmpleOnline

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

// In this example, the system will transition to state b
// after a 1-second delay
class X {
  sm {
    a {
      entry / {System.out.println("entering a");}
      after(1) -> b;
    }
    b {
      entry / {System.out.println("entering b");}
    }
  }
  public static void main(String [ ] args) {
    X x = new X();
  }
}
```

Load the above code into UmpleOnline

Another Example

```
view plain

// In this example, the system will transition to state b
// when the value of the attribute count drops below 5
// An attempt to make this transition is taken every second
// The do activity slowly drops the value of count
class X
{
  Integer count = 10;
  sm
  {
    a
    {
      entry / { System.out.println("entering a"); }
      do
      {
        while (count > 0)
        {
          System.out.println("Count = " + count);
          Thread.sleep(1000);
          count --;
```

```
      }
     }
     afterEvery(1) [count<5] -> b;
    }
    b
    {
     entry / { System.out.println("entering b"); }
    }
  }

  public static void main(String [] args)
  {
   X x = new X();
  }
}

```

Load the above code into UmpleOnline

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

```
//The class Machine uses both deep history
//and regular history states.

//When the event emergency occurs, the
//state of normalOperation is stored.
//When the issue is resolved, depending
//on the intended behaviour, the state
//machine goes back to the specific
//substate it was previously in (HStar)
//or it restarts the phase it was in (H)

class Machine {
  sm {
    initializing {readyForOperation -> normalOperation;}
    normalOperation {
      phase1 { completePhase1 -> phase2;}
      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)
```

```
```

Load the above code into UmpleOnline

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 {
        colorSelected -> idle;
      }
      changeMode -> selecting;
    }
    selecting {
      selectionCompleted -> drawing.H;
    }
  }
}

```

Load the above code into UmpleOnline

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

// The do activity of the first state machine is a thread that
// communicates with the second state machine.
// It calls the event method e with different arguments.
// The second state machine only changes state when the
// guard detects that the argument is valid
// The argument is also used in the transition action code.
class X {
  stateMachine1 {
    s1a {
      do {
        // This do activity sends events to stateMachine2
        e(5);
        Thread.sleep(1000);
        e(6);
        Thread.sleep(1000);
        e(7);
        Thread.sleep(1000);
        e(8);
      }
      -> s1b;
    }
    s1b {}
  }
  stateMachine2 {
    s2a {
      entry / {System.out.println("s2a");}
      e(int a) [a > 6] / {System.out.println("e"+a);} -> s2b;
    }
    s2b {
      entry / {System.out.println("s2b");}
      e(int a) / {System.out.println("e"+a);} -> s2a;
    }
  }
  public static void main(String [ ] args) {
    X x = new X();
  }
}


```

Load the above code into UmpleOnline

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
```

```
    test.e4(); //   next in s4

            // pooled: processed goes to s3
            // pooled: process e3 from queue
            //   goes to s4
            // pooled: process s4 from queue
    test.e2(); //   goes to s1

    test.e1();
    test.e3();
    test.e2();
    test.e4();

    test.e1();
    test.e2();
    test.e3();
    test.e4();

    test.e1();
    test.e2();
    test.e4();
    test.e3();
    test.e4();

    test.e1();
    test.e3();
    test.e2();
    test.e3();
    test.e1();
    test.e4();

    test.e2();
    test.e1();
    test.e2();

  }
}


```

Load the above code into UmpleOnline

Syntax

```
//Issue 148
inlineStateMachine : [=queued]? [=pooled]? [~name] { ( [[comment]]
   | [[state]]
   | [[trace]]
   | [[mixsetDefinition]]
   | [=ll]
   | [[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

```
// Test of pooled state machines.
// The word pooled results in a thread being
// created to process events, and 'pooled'
// semantics being employed. Events which cannot
// be handled are kept waiting until entry into a
// state where they can be handled.

class TestSM {
  String ev="";
  pooled sm{
   s1 {
     e1 /{ev="e1";} ->s2;
     e5 /{ev="e5";} ->s2;
   }
   s2 {
     e2 /{ev="e2";} ->s3;
   }
   s3 {
     e3 /{ev="e3";} ->s4;
   }
   s4 {
     e4 /{ev="e4";} ->s1;
   }
  }
  after e* {
    if(wasEventProcessed)
      System.out.println(ev);
  }

  public static  void main(String [] ags){
    TestSM test=new TestSM();
    test.e1(); // processed s2

              // Will only be processed in the
    test.e5(); // pooled case (eventually)

    test.e2(); // processed s3
```

```
    test.e3(); // processed s4
    test.e4(); // processed s1

    test.e1(); // processed s2

            // queued: ignored
            // pooled: left in queue, until we
    test.e3(); //   are next in s3

            // pooled: left in queue, until
    test.e4(); //   next in s4

            // pooled: processed goes to s3
            // pooled: process e3 from queue
            //   goes to s4
            // pooled: process s4 from queue
    test.e2(); //   goes to s1


    test.e1();
    test.e3();
    test.e2();
    test.e4();

    test.e1();
    test.e2();
    test.e3();
    test.e4();

    test.e1();
    test.e2();
    test.e4();
    test.e3();
    test.e4();

    test.e1();
    test.e3();
    test.e2();
    test.e3();
    test.e1();
    test.e4();

    test.e2();
    test.e1();
    test.e2();

  }
}
```

Load the above code into UmpleOnline

Syntax

```
//Issue 148
```

inlineStateMachine : [=queued]? [=pooled]? [~name] { ([[comment]]
| [[state]]
| [[trace]]
| [[mixsetDefinition]]
| [=ll]
| [[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

```
// Class Diagram illustrating use of top-level state machine with
// usage in separate classes.
// Note that this could also be accomplished by putting the
// state machine in a trait, and then having each class use the trait

class MotorController {
  motorStatus as deviceStatus;
}

class BrakeController {
  brakeStatus as deviceStatus;
}

// By itself, the following will not generate any code
// but it can be debugged by itself outside of any class
// and can be incorporated in multiple classes, as above
statemachine deviceStatus {
  inactive {
    activate -> booting;
  }
  booting {
    completedStartupChecks -> active;
    startupCriticalErrorDetected -> outOfOrder;
  }
  active {
    runtimeCriticalErrorDetected -> outOfOrder;
    deactivate -> shuttingDown;
  }
  shuttingDown {
    shutdownComplete -> inactive;
  }
  outOfOrder {
    repaired -> inactive;
```

```
  }
}

```

Load the above code into UmpleOnline

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
```

```
48.
49.
50.
```

Load the above code into UmpleOnline

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]] )*
```

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 UmpleOnline

Example of a method with arguments

```
view plain

generate Java;

// A class with a method that has arguments
class Person
{
  name;

  String displayName(String greeting, String property)
  {
    return(
      greeting +
      ", my " +
      property +
      " is " +
      getName());
  }
}


```

Load the above code into UmpleOnline

Example of a public method

```
view plain

generate Java;

// A class with a method that uses 'public'.
class Person
{
  name;

  public String displayName()
  {
    return("Hello, my name is " + getName());
  }
}

```

Load the above code into UmpleOnline

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

//The class Person contains abstract
//methods, which are inherited and
//implemented by its subclass Student

//Person is abstract, but Student is a
//concrete class as it has no methods
//that are unimplemented.
class Person {
  abstract void method1();
  abstract void method2();
  void method3() {
    /* Implementation */
  }
}

class Student {
  isA Person;

  void method1() {
    /* Implementation */
  }
  void method2() {
    /* Implementation */
  }
}

```

Load the above code into UmpleOnline

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() {
    /* 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() {
    return getWidth()*getHeight();
  }
}

class Circle {
  isA Shape;
  Double radius;
  const Double PI = 3.1416;

  Double area() {
    return PI*radius*radius;
  }
}

class A {
  Square s;
  Circle c;

  void Test() {
    //The method implemented in Shape is
    //used
    s.move();
    c.move();

    //The methods implemented in their
    //respective classes are used
    s.area();
    c.area();
  }
}

```

Load the above code into UmpleOnline

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

// Generating Java and Php the following will
// result in the appropriate
// method body being selected.
class LanguageSpecific {
  int m1() Java {
    // Return 1 if the model is compiled into Java
    return 1;
  }
  Php {
     // Return 0 if the model is compiled into Php
    return 0;
  }
}
```

Load the above code into UmpleOnline

Another Example

```
view plain

// A method body may work for several languages.
// If no language is specified it is taken as the
// default.
class LanguageSpecific {
  int m1()
  Java,Php {
    // Return 0 if the model is compiled into
    // Java or Php
    return 0;
  }
  {
    // Default implementation for other languages
    return 1;
  }
}
```

Load the above code into UmpleOnline

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

```
class Mario {
  Modifiers {
    Small {
      Normal {
        void onHit() {
          Die();
        }
      }

      Invulnerable {
        void onHit() {
          // took no damage
        }
      }

      void render() {
        // draw small mario
      }

      EatMushroom -> Large;
      Star -> Small.Invulnerable;
      Die -> Dead;
    }

    Large {
      Normal {
        void onHit() {
          Shrink();
        }
      }

      Invulnerable {
        void onHit() {
          // took no damage
        }
      }

      void render() {
        // draw large mario
      }

      Shrink -> Small;
      Star -> Large.Invulnerable;
```

```
    }

    Dead {
      void render() {
        // draw dying animation
      }
    }
  }
}

// @@@skipphpcompile - See #1351. State-
dependent methods are exclusive to Java generation.


```

Load the above code into UmpleOnline

Another Example

```
view plain

external Thread {};

class Philosopher {

  isA Thread;

  Fork leftFork;
  Fork rightFork;
  String fname;
  boolean isLeftHanded;

  ForksInHand {
    NoForks {
      LookingForForks {
        void eat(){
          if (isLeftHanded) {
            if (leftFork.take()) {
              System.out.println(fname + " picked up first fork (left)");
              LeftForkAcquired();
            }
          }
          else if (rightFork.take()) {
            System.out.println(fname + " picked up first fork (right)");
            RightForkAcquired();
          }
        }
        AteFood -> Idling;
      }
      Idling {
        BellyGrumble -> LookingForForks;
      }
      LeftForkAcquired -> ForkInLeft;
      RightForkAcquired -> ForkInRight;
    }
    ForkInLeft {
      void eat() {
        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.   }
```

```
  }
}

class Fork {
  depend java.util.concurrent.*;
  Semaphore fork = new Semaphore(1);

  boolean take() {
    try {
      Thread.sleep(100);
    } catch (InterruptedException ex) { }
    return fork.tryAcquire();
  }

  void putDown() {
    fork.release();
  }
}

class Main {
  public static void main(String[] args){
    String[] names = {"Plato", "Aristotle", "Cicero", "Confucius", "Eratosthenes"};
    int k = names.length;
    Philosopher[] philosophers = new Philosopher[k];
    Fork[] forks = new Fork[k];

    for (int i = 0; i < k; ++i) {
      forks[i] = new Fork();
    }

    for (int i = 0; i< k; ++ i) {
      Fork leftFork = forks[i];
      Fork rightFork = forks[(i + 1) % forks.length];
      philosophers[i] = new Philosopher(leftFork, rightFork, names[i], i == k - 1);
      philosophers[i].start();
    }
  }
}

// @@@skipphpcompile - See #1351. State-
dependent methods are exclusive to Java generation.

```

Load the above code into UmpleOnline

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 incirporating 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.

Example

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
*/
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;
}

```

Load the above code into UmpleOnline

Another Example

view plain

```
/*
   Example 2: showing a trait designed for
   comparison regarding if two objects are equal
   or not.
*/
trait TEquality{
  Boolean isEqual(Object object);
  Boolean isNotEqual(Object object){
    return isEqual(object) ? true : false;
   }
}
// @@@skipcompile Java code not compilable no class
```

.3.

Load the above code into UmpleOnline

Syntax

traitDefinition : trait [name] [[traitParameters]]? { [[traitContent]]* }

traitContent- : [[mixsetDefinition]]
| [[testCase]]
| [[requiredModelElements]]
| [[comment]]
| [[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.

Example

view plain

```
01. /*
02.   Example 1: showing how traits can be used by
03.   their clients.
04. */
05. trait T1{
06.   abstract void method1();
07.   abstract void method2();
08.   void method4(){/*implementation*/ }
09. }
10. trait T2{
11.   isA T1;
12.   void method3();
13.   void method1(){/*implementation*/ }
14.   void method2(){/*implementation*/ }
15. }
```

```
class C1{
  void method3(){/*implementation*/ }
}
class C2{
  isA C1;
  isA T2;
  void method2(){/*implementation*/ }
}

```

Load the above code into UmpleOnline

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.

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
*/
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;
}

```

Load the above code into UmpleOnline

Example showing how methods can obtain states in traits

```
view plain

/*
 Example 2: showing how states in traits can be
 obtained by required methods.
*/
trait T1{
  Integer getData();
  void setData(Integer);
  Integer processData(){
    int data = getData();
    data=data+2;
    setData(data);
  }
}
class C1{
  Integer data;
  isA T1;
}
// @@@@@@testlanguage=none Java code not compile

```

Load the above code into UmpleOnline

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.

Example 1

view plain

```
1.  /*
2.     Example 1: showing how template parameters in
3.     traits are defined and used.
4.  */
5.  trait T1< TP isA I1 > {
6.    abstract TP method2(TP data);
7.    String method3(TP data){ /*implementation*/ }
8.  }
9.  interface I1{
10.   void method1();
11. }
12. class C1{
13.   isA I1;
14.   isA T1<TP = C1>;
15.   void method1(){/*implementation*/}
16.   C1 method2(C1 data){ /*implementation*/ }
17. }
18. class C2{
19.   isA I1;
20.   isA T1< TP = C2 >;
21.   void method1(){/*implementation*/}
22.   C2 method2(C2 data){ /*implementation*/ }
23. }
24. // @@@skipcompile issue needs raising missing return statement
25.
```

Load the above code into UmpleOnline

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) {/*implementation*/}
}
trait T2<TP>{
  isA T1< TP = TP >;
  void method3(TP Data) {/*implementation*/}
}
class C1{
  isA T2< TP = String >;
  void method1(){/*implementation*/}
}

```

Load the above code into UmpleOnline

Example 3

```
view plain

/*
   Example 3: showing how template parameters in
   traits are used in code blocks.
*/
trait T1 <TP>{
  String method1();
  String method2(){
    #TP# instance = new #TP#();
    return method1() +":"+instance.process();
  }
}
class C1{
  String process(){/*implementation*/}
}
class C2{
  isA T1< TP = C1 >;
  String method1(){/*implementation*/ }
}
// @@@skipcompile issue needs raising as Java
// compile fails missing return statement

```

Load the above code into UmpleOnline

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.

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. class Dashboard{
06.   void update (Sensor sensor){ /* code */ }
07. }
08. class Sensor{
09.   isA Subject< Observer = Dashboard >;
10. }
11. trait Subject <Observer>{
12.   0..1  -> * Observer;
13.   void notifyObservers() { /* code */ }
14. }
```

```
.5.
```

Load the above code into UmpleOnline

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 UmpleOnline

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;
```

```
  Integer d;
  mixset f6 {[c > d]}
  after setC {
     System.out.println("SetC has happened from T1");
  }
  mixset f1 {1 -- 1..* C4;}
}

trait T2 {
  after setC {
    System.out.println("setC has happened from T2");
  }
}

trait T3 {
  void m2() {
    System.out.println("M2 is executing");
  }
  sm1 {
    s1 {
      e -> s2;
    }
    s2 {
    }
  }
}

trait T4 {
  isA T3;
}

class C1 {
  isA T1, I1;
  mixset f1 {1 -- * C3;}
}

mixset f5 class C2 {
  mixset f4 {1 -- 1..* C3;}
  isA C1, T2;
  void m1() {
    System.out.println("Overriding impl of m1");
  }
}

class C3 {
}

mixset f4 class C4 {
  isA T4;
  // Commented code bellow will cause an error when mixset f4 is used even when mix
  // This because inline mixsets does not support apsect.
  //mixset f5 after custom m2() {System.out.println("m2 occurred");}
  mixset f5 { after custom m2() {System.out.println("m2 occurred");} }
  after e1 {System.out.println("event e1 occurred");}
}

```

Load the above code into UmpleOnline

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.

Example

view plain

```
01. /*
02.    Example 1: showing how required interfaces
03.    in traits are defined and used.
04. */
05. interface I1{
06.   void method1();
07.   double method2();
08. }
09. interface I2 {
10.   isA I1;
11.   Boolean method3();
12. }
13. trait T1{
14.   isA I1;
```

```
  Float method3(){/*implementation*/ }
}
trait T2{
  isA I1;
  Float method4(){/*implementation*/ }
}
trait T3{
  isA T1,T2;
}
class C3{
  void method1(){/*implementation*/ }
  Double method2(){/*implementation*/ }
}
class C1{
  isA C3, I1, T1;
}
class C2 {
  isA C3, I2, T3;
  Boolean method3(){/*implementation*/ }
}
// @@@skipcompile Java code not compilable

```

Load the above code into UmpleOnline

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.

Example 1: Building a class with two state machines

```
view plain

/*
   Example 1: showing how state machines are
   defined and used in traits.
*/
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 UmpleOnline

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;}
   }
}
trait T2 {
  isA T1;
  sm2{
  s0 {e1-> s1;}
  s1 {e0-> s0;}
   }
}
class C1 {
  isA T2;
```

```
}
```

Load the above code into UmpleOnline

Example 3: Satisfying required methods

```
view plain

/*
   Example 3:showing how required methods
   of traits are satisfied by events of
   state machines.
*/
trait T1{
  Boolean m1(String input);
  Boolean m2();
  sm1{
    s1{ e1(String data) -> /{ m1(data); } s2;   }
    s2{ e2 -> /{ m2(); } s1; }
  }
}
class C1{
  isA T1;
  sm2{
    s1{ m1(String str) -> s2;}
    s2{ m2 ->  s1;}
  }
}
```

Load the above code into UmpleOnline

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 UmpleOnline

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)

class MotorController {
  isA DeviceWithStatus;
}

class BrakeController {
  isA DeviceWithStatus;
}

trait DeviceWithStatus {
  deviceStatus {
    inactive {
      activate -> booting;
    }
    booting {
      completedStartupChecks -> active;
      startupCriticalErrorDetected -> outOfOrder;
    }
    active {
      runtimeCriticalErrorDetected -> outOfOrder;
      deactivate -> shuttingDown;
    }
    shuttingDown {
      shutdownComplete -> inactive;
    }
    outOfOrder {
      repaired -> inactive;
    }
  }
}
```

Load the above code into UmpleOnline

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:

$$\text{isA TName} < \text{Operator1}, \text{Operator2}, \ldots, \text{Operatorn} >$$

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:

$$(+|-)\ \text{methodName(argumentTypes)}$$

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:

$$(+)\ \text{methodName(argumentTypes) as newName}$$

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.

Example

```
view plain

/*
   Example 1: showing how the operator
   "Removing/keeping provided methods" works.
*/
trait T1{
  abstract method1();
  void method2(){/*implementation*/}
  void method3(){/*implementation*/}
  void method4(){/*implementation*/}
  void method5(){/*implementation*/}
}
class C1{
  isA T1<-method2() , -method3()>;
  void method1() {/*implementation for C1*/}
}
class C2{
  isA T1<+method5()>;
  void method1() {/*implementation for C2*/}
}

```

Load the above code into UmpleOnline

Another Example

```
view plain

/*
   Example 2: showing how the operator
   "Renaming (Aliasing) provided methods" works.
*/
trait T1{
  abstract method1();
  void method2(){/*implementation*/}
  void method3(){/*implementation*/}
  void method4(){/*implementation*/}
  void method5(Integer data){/* implementation*/}
}
class C1{
  isA T1< method2() as function2 >;
  void method1() {/*implementation related to C1*/}
}
class C2{
  isA T1< method3() as private function3 >;
```

```
18.   void method1() {/*implementation related to C2*/}
19. }
20. class C3{
21.   isA T1< +method5(Integer) as function5 >;
22.   void method1() {/*implementation related to C3*/}
23. }
24.
```

Load the above code into UmpleOnline

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.

Example

```
view plain

/*
   Example 1: 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, sm2 as mach2>;
}
```

Load the above code into UmpleOnline

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 UmpleOnline

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>;
}

```

Load the above code into UmpleOnline

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 UmpleOnline

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. /*
```

```
   Example 5: showing how the operator
   "Removing/keeping a state machine" works.
*/
trait T1 {
  sm1{
    s0 { e0-> s0;}
  }
sm2{
    t0 { e0-> t0;}
  }
sm3{
    w0 { e0-> w0;}
  }
}
class C1 {
  isA T1<-sm1>;
}
class C2 {
  isA T1<+sm2>;
}
```

Load the above code into UmpleOnline

Another Example

```
view plain

/*
   Example 6: showing how the operator
   "Removing/keeping a state" works.
*/
trait T1 {
  sm1{
    s0 {
      e1-> s1;
      e2-> s2;
    }
    s1 {
      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>;
}
```

```
31.
```

Load the above code into UmpleOnline

Another Example

view plain

```
01. /*
02.    Example 7: showing how the operator
03.    "Removing/keeping a region" works.
04. */
05. trait T1{
06.  sm {
07.    s1{
08.      r1{ e1-> r11; }
09.      r11{}
10.      ||
11.      r2{
12.        e2-> r11;
13.        e3-> r21;
14.        e4-> r31;
15.      }
16.      r21{}
17.      ||
18.      r3{e5->r31; }
19.      r31{}
20.    }
21.  }
22. }
23. class C1{
24.   isA T1<-sm.s1.r1>;
25. }
26. class C2{
27.   isA T1<+sm.s1.r2>;
28. }
29. // @@@skipcompile issue needs raising as Java compile fails
30.
```

Load the above code into UmpleOnline

Another Example

view plain

```
01. /*
02.    Example 8: showing how the operator
03.    "Removing/keeping a transition" works.
04. */
05. trait T1{
06.   internal Boolean cond;
07.   sm {
08.     s1{
09.       e2(Integer i)-> s2;
10.       e3[!cond]-> s3;
11.       e4[cond]-> s4;
12.     }
```

```
    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.[]>;
}

```

Load the above code into UmpleOnline

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

```
1.  /*
2.     Example 9: showing how the operator
3.     "Extending a state by adding a state machine
4.     to it" works.
5.  */
6.  trait T1{
7.    sm1{
8.      m1{
9.        t2-> m2;
0.      }
1.      m2{
2.        t1-> m1;
3.      }
4.    }
5.  }
6.  class C1{
7.    isA T1<sm1 as sm.s2>;
8.    sm{
9.      s1{
0.        e2-> s2;
1.      }
2.      s2{
3.        e1-> s1;
4.      }
5.    }
6.  }
7.
```

Load the above code into UmpleOnline

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

```
1.  /*
2.     Example 1: showing how traits are combined with
3.     Umple mixins.
4.  */
5.  trait T1{
6.    void method1();
7.    void method2(){/*impl… */ }
8.  }
9.  trait T1{
10.   void method3(){/*impl… */ }
11. }
12. class C1{
13.   isA T1;
14.   void method1(){/*impl… */ }
15. }
16.
```

Load the above code into UmpleOnline

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
1.  class X
2.  {
3.    a;
4.    whenWasASet;
5.
6.    after setA {
7.      setWhenWasASet(getA() + System.currentTimeMillis());
8.    }
9.  }
0.
```

Load the above code into UmpleOnline

Example with "before" statement

```
view plain
1.  class Person {
2.    name;
3.    before setName {
4.      if (aName != null && aName.length() > 20) { return false; }
5.    }
6.  }
7.
```

Load the above code into UmpleOnline

Example with both "before" and "after" statements

```
view plain
1.  class Operation {
2.    const Boolean DEBUG=true;
```

```
  query;
  before constructor {
    if (aQuery == null)
    {
      throw new RuntimeException("Please provide a valid query");
    }
  }
  after constructor {
    if (DEBUG) { System.out.println("Created " + query); }
  }
}

```

Load the above code into UmpleOnline

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

class Student
{
  const Boolean DEBUG = true;
  firstName;
  lastName;
  cityOfBirth;
  before get*Name {
    if ( DEBUG) { System.out.println("accessing the name"); }
  }
}

```

Load the above code into UmpleOnline

Another Example

```
view plain

class Student
{
  const Boolean DEBUG = true;
  firstName;
  lastName;
  cityOfBirth;
  before get*,!getCityOfBirth {
    if ( DEBUG) { System.out.println("accessing the name"); }
  }
}


```

Load the above code into UmpleOnline

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 UmpleOnline

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

```
// This illustrates separate code injection
// for custom and generated methods
// The output will be as follows
// Sound of beep
// Warning
//  ... this is injected after the warning
// Sound of beep
// Warning
//  ... this is injected after the warning
// Sound of beep
// Warning
// ... this is injected after the warning
//
class Car {
  queued radio {
    on {
      radioToggle -> off;
    }
    off {
      radioToggle -> on;
    }
  }
  after generated radioToggle* {soundBeep();}

  void soundBeep() {
    System.out.println("Sound of beep");
    radioToggleWarning();
  }

  void radioToggleWarning() {
    System.out.println("Warning");
  }

  after custom radioToggle* {
    System.out.println(
      " ... this is injected after the warning");
  }

  public static void main(String[] args) {
    Car x = new Car();
    x.radioToggle();
    x.radioToggle();
    x.radioToggle();
  }
}


```

Load the above code into UmpleOnline

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
after {*} set* { 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;}
class Employer {isA X;}
trait X {
  doSomething() {
    System.out.println("Doing something\n");
  }
}

```

Load the above code into UmpleOnline

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 UmpleOnline

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

class AroundClass{
  static void doSomeThing()
  {
    int x; int a;
    boolean flag = false;
    Label1: x = 0;
    Label2: a = 0;
    flag = true;
  }
}

class AroundClass
{
  around custom doSomeThing()
  {
    // code before around.
    if (true)
    {
      around_proceed:
    }
    // code after around.
  }
}

```

Load the above code into UmpleOnline

Example with "top-level around injection with two labels" statement

```
view plain

class AroundClass{
  static void doSomeThing()
  {
    boolean flag = false;
    Label1: int x = 0;
    Label2:
    flag = true;
  }
}

around {AroundClass} custom Label1-Label2:doSomeThing()
{
  // code before around.
  if (true)
  {
    around_proceed:
```

```
  }
  // code after around.
}

```

Load the above code into UmpleOnline

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

```
// Initially there are two classes with no attributes
class X {}
class Z {}

// In another place, potentially a separate file.
// class X is given attribute a
// This is a simple mixin
class X {
 a;
}

// 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 {}
}


```

Load the above code into UmpleOnline

YouTube Video with Additional Explanation

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]] | [[mixsetInlineDefinition]])

mixsetInnerContent- : { [[extraCode]] }

mixsetInlineDefinition- : ([entityType] [entityName] ([[mixsetInnerContent]]
| [entityOneElement])
| [oneElement])

Filters

Documentation of the filters feature is being developed

A filter directive in Umple can be used to select only a certain part of a model 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 (seven 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.

Within a filter directive, one can specify

- *include* statements. These list the main classes to be included. There can be any number of include statements, and multiple comma-separated classes can follow each include keyword. 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 other 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. Note that including any class will always include its superclasses, so the complete picture of what is inherited is presented.

Load the example below into UmpleOnline and uncomment any of the 8 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

// Example to demonstrate filters. Uncomment one of
// the filters shown below to display a different
// diagram, or generate a different subset of the
// code

// This model describes abstract art consisting of
// various shapes connected in 3-D space, some
// of which can contain others.

```

```
// Filter 1. Show only Shape3D
// filter {include Shape3D;}

// Filter 2: Show classes connected to Model3D with
// one hop
//  filter {include Model3D; hops {association 1;}}

// Filter 3: Show classes related to qualities
// filter {namespace qualities ;}

// Filter 4: Show Shape3D, its subclasses and all
// their connections
// filter {include Shape3D; hops {sub 1;}}

// Filter 5: Including also includes parents
// filter { include Cube;}

// Filter 6: Show two classes and their neighbours
//  filter {include Surface, Model3D; hops {association 1;}}

// Filters 7: Named filters and use of the named filter
 filter 7 {include Connector; hops {association 1;}}
 filter 7a {include Cube;} // includes its parents
//  filter {includeFilter 7,7a;}

// Filters 8: Including classes that are associated
// will always show the associations
//  filter { include Colour, SurfaceQuality;}


class Point3D {
  Double x;
  Double y;
  Double z;
}

class Model3D {
  depend Qualities.*;
  depend Shapes.*;
  name;
  1 -- * Shape3D;
  1 -- * Connector;
}

namespace Qualities;

class Colour {
  Integer red;
  Integer green;
  Integer blue;
}

class SurfaceQuality {
  * -> 1 Colour;
  Float reflectance;
  Float transparency;
}
```

```
namespace Shapes;

// 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;
}

class Cube {
  isA RegularPolyhedron;
}

strictness ignore 30; // hide namespace warnings

```

Load the above code into UmpleOnline

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

```
// Simple example to demonstrate umple's template
// base string generation mechanism.
// In this approach there is an iterative call
// to the emit function

class MathExample {

  // A simple template to inject a platform
  // specific newline
  cr <<!
!>>

  // Two templates for different lines of output
  header <<!Print out a <<=times>> times table!>>
  row <<!  <<=times>> times <<=count>> is <<=times*count>>!>>

  // Specification for two methods to be generated
  // to output parts of the text
  // Method arguments are in the first parentheses
  // Templates to output are in the second set
  emit printHeader(int times)(header, cr);
  emit printRow(int times, int count)(row, cr);

  // Main program to run the above and generate
  // the output
  public static void main(String[] argv) {
    int times = 10; // default
    MathExample m = new MathExample();

    if(argv.length > 0) {
      times=Integer.parseInt(argv[0]);
    }

    // Print the header
    System.out.print(m.printHeader(times));
    // Print one row for each element
    for (int i=0; i <= times; i++) {
      System.out.print(m.printRow(times,i));
    }
  }
}
```

Load the above code into UmpleOnline

Example 2

view plain

```
// Simple example to demonstrate umple's template
// base string generation mechanism.
// In this approach iteration is embedded in the
// rows template and there is a single emitter
// function generated called result

class MathExample {

  // 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 UmpleOnline

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;
  String sender; String senderTitle;

```

```
  // Letter template
  letterTemplate <<!
Subject: Reference request for <<=applicant>>, File #<<=fileno>>

Dear <<=recipient>>,
Our company, Umple Enterprises, is hiring talented software
engineers.

We have received an application from <<=applicant>> who named you
as an individual who could provide a letter of reference. Would you
please reply to this letter, answering the following questions:
  * In what capacity do you know <<=applicant>>
  * For how long have you known <<=applicant>>
  * Describe the abilities of <<=applicant>> in software development
  * What his or her strengths and weaknesses?
  * Please provide your phone number and suitable times to call in
    case we need to follow up

Yours sincerely,
<<=sender>>
<<=senderTitle>>
!>>

  // Specification of the method to generate
  emit letterTemplate()(letterTemplate);

  // Main program to generate the letter
  public static void main(String[] argv) {
    if(argv.length < 5) {
      System.err.println("You must specify arguments for fileno, recipient, applicant, sen
    }
    // Output the entire result
    else System.out.print(new RefLetterRequest(
      argv[0], argv[1],argv[2], argv[3], argv[4]
    ).letterTemplate());
  }
}
```

Load the above code into UmpleOnline

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

```
// Umple library for generating html
// Currently it supports h1, h2 ... as well as p, and table tags.
// This will be extended to support more of html later
// It serves as an illustration of Umple's generation templates

// Class representing either a top level html node or a collection of subnodes
class HtmlNode {
  String getContent() {return "";};  // should be abstract
}

// Simple text to be output between tags - these are leaves
class HtmlTextNode {
  isA HtmlNode;
  String content;
}

// Non-leaf nodes in the html tree
class HtmlRegularNode {
  isA HtmlNode;
  const String Xmltagstart = "<";
  const String Xmltagend = ">";

  // Arguments for the constructor
  String tag; // e.g. p, h1, a
  String arguments; // e.g. href

  0..1 -> * HtmlNode subnodes; // whatever to emit between tags

  // The following template will render any html node
  rendering <<!<<=Xmltagstart>><<=getTag()>> <<=getArguments()>>
<<=Xmltagend>>
  <<#
  for(HtmlNode n : getSubnodes()) {#>><<=n.getContent()>>
  <<#}#>><<=Xmltagstart>>/<<=getTag()>><<=Xmltagend>>!>>

  emit getContent()(rendering);

  HtmlTable table() {
    HtmlTable t = new HtmlTable();
    addSubnode(t);
    return t;
  }

  HtmlTextNode text(String s) {
    HtmlTextNode t = new HtmlTextNode(s);
    addSubnode(t);
    return(t);
  }

```

```
  HtmlRegularNode taggedText(String tag, String arguments, String s) {
    HtmlRegularNode r = new HtmlRegularNode(tag, arguments);
    HtmlTextNode t = new HtmlTextNode(s);
    r.addSubnode(t);
    addSubnode(r);
    return(r);
  }
}

class HtmlTable {
  isA HtmlRegularNode;

  HtmlTable() {
    super("table","");
  }

  HtmlRow tr() {
    HtmlRow r = new HtmlRow();
    addSubnode(r);
    return r;
  }
}

class HtmlRow {
  isA HtmlRegularNode;

  HtmlRow() {
    super("tr","");
  }

  // Add a cell that contains text
  HtmlCell td(String s) {
    HtmlCell c = td();
    c.text(s);
    return(c);
  }

  // Add a cell that can contain anything
  HtmlCell td() {
    HtmlCell c = new HtmlCell();
    addSubnode(c);
    return c;
  }
}

class HtmlCell {
  isA HtmlRegularNode;

  HtmlCell() {
    super("td","");
  }
}

class HtmlGen {

  lazy HtmlNode firstNode;

  // Subtrees for the header and body
```

```
  0..1 -> * HtmlNode headerNodes;
  0..1 -> * HtmlNode bodyNodes;

  after constructor {
    firstNode = new HtmlTextNode(filehead());
  }

  filehead <<!<!DOCTYPE html PUBLIC "-
//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
transitional.dtd">!>>
  emit filehead()(filehead);

  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);
```

```
  }

  // Adds a top level p with plain text
  // Subsequent calls can add additional elements to the p
  // beyond just the text
  HtmlRegularNode p(String text) {
    return addSimpleNode("p",text);
  }


  // Adds a top level table
  HtmlTable table() {
    HtmlTable t = new HtmlTable();
    addBodyNode(t);
    return t;
  }
}
```

Load the above code into UmpleOnline

Another Example

view plain

```
// Umple program demonstrating generation
// of the html generation library for Umple

use ../HtmlGeneration.ump;

class TimesTable {
  public static void main(String[] argv) {
    int times = 10; // default
    HtmlGen h = new HtmlGen();
    HtmlRegularNode p;

    if(argv.length > 0) {
      times=Integer.parseInt(argv[0]);
    }

    // Specify the table main header
    h.h(1, "Multiplication Table: "+times);

    // Generate an explanatory paragraph
    p = h.p("The following is a ");
    p.taggedText("i","","simple");
    p.text(" multiplication table");

    // Generate another paragraph with some subnodes
    // showing a slightly different use of the API
    p = new HtmlRegularNode("p","");
    h.addNode(p);
    p.text("Nodes in ");
    p.taggedText("font", "color=\"red\"", "red");
    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 i=1; i<= 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());
62.   }
63. }
64.
65.
```

Load the above code into UmpleOnline

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 UmpleOnline

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

class X {
  activesm {
    topstate {
      thread1 {
        do {
          System.out.println(
            "Doing the activity - Normally "+
            "this would last a long time");
          Thread.sleep(1000);
          System.out.println("Done thread 1");
        }
      }
      ||
      thread2 {
        do {
          System.out.println(
            "Doing the other activity - Again "+
            "this would last a long time");
          Thread.sleep(3000);
          System.out.println("Done thread 2");
        }
      }
    }
  }
  public static void main(String[] argv) {
    new X();
  }
}


```

Load the above code into UmpleOnline

Another Example

```
view plain

// This is a more sophisticated example
class X {
 sm {
   s1 {
     do {
       System.out.println("In S1");
       Thread.sleep(1000);
       System.out.println(
         "Still in S1 after sleep");
       Thread.sleep(2000);
       System.out.println(
```

```
        "About to leave S1");
    }
    -> s2;
  }
  s2 {
   do {
     System.out.println("In S2");
     Thread.sleep(1000);
     System.out.println(
       "Still in S2 after sleep");
     Thread.sleep(2000);
     System.out.println(
       "About to leave S2");
    }
    nesteds2a {
     do {
       System.out.println("In S2a");
       Thread.sleep(1000);
       System.out.println(
         "Still in S2a after sleep");
       Thread.sleep(2000);
       System.out.println(
         "About to leave S2a");
     }
    }
    ||
    nesteds2b {
     do {
       System.out.println("In S2b");
       Thread.sleep(1000);
       System.out.println(
         "Still in S2b after sleep");
       Thread.sleep(2000);
       System.out.println(
         "About to leave S2b");
     }
    }
  }
 }
 public static void main(String[] argv) {
  System.out.println("In Main");
  X theX = new X();
  System.out.println("End of Main");
 }
}
```

Load the above code into UmpleOnline

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 'distributed' 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.
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. For 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("Customer"));
```

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("Component1"));
```

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}}
```

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
```

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();
```

More options

One can set all classes to be distributable by using the following directive:

```
distributable forced;
```

One can forbid all classes 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;
```

More examples and details about the distributable feature will be provided shortly.

Syntax

distribute :distributable [=distributeTech:RMI
|WS]? [=distributePattern:0
|1
|2
|3]? [=distributeVal:on
|off
|forced] ;

distributable- : [=distributable:distributable] [=distributeTech:RMI|WS]? ;

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

```
// Example system representing constraint-based ping-pong example
// Demonstrates the composite Structure Diagram in Umple

class Component1 {
  // Relay-Port
  public in Integer pIn1;
  public out Integer pOut1;
  pIn1 -> pOut1;


  [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;
}
```

Load the above code into UmpleOnline

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]])*

// 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 (also see here) 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. class Student
02. {
03.   Integer id;
04. 
05.   // Whenever the id attribute is changed,
06.   // emit a trace record
07.   // The record will have the format id=value
08.   trace id;
09. 
10.   // The following Java main program can be used
11.   // to demonstrate this in operation
12.   // This will output a record when id is set to 9.
13.   public static void main(String [ ] args) {
14.     Student s = new Student(6);
15.     s.setId(9);
16.   }
17. }
18. 
19. 
```

Load the above code into UmpleOnline

Another Example

view plain

```
01. class Student
02. {
03.   lazy Integer id;
04.   lazy name;
05. 
06.   // It is possible to specify multiple
07.   // UML entities to trace as follows
08.   // It is also possible to indicate interesting
```

```
19.   // information to record in addition to the
20.   // value of the element being traced
21.   trace id, name record "interesting behavior";
22.   trace id record "Even more interesting";
23.
24.   // Traces will be triggered for each call to a 'set' method
25.   // and twice for id, since there are two trace statements for id
26.   public static void main(String [ ] args) {
27.     Student s = new Student();
28.     s.setId(9);
29.     s.setName("Tim");
30.   }
31. }
32.
33.
```

Load the above code into UmpleOnline

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

// This traces setId() and setName()
class Student
{
  Integer id;
  String name;
  trace id;
  trace set name;
}


```

Load the above code into UmpleOnline

Another Example

```
view plain

// This traces  getId() method
class Student
{
  Integer id;
  trace get id;
}
// @@@skipphpcompile See issue 596 (PHP tracing causes issues)


```

Load the above code into UmpleOnline

Another Example

```
view plain

// This example traces getId() and setId()
class Student
{
  Integer id;
```

```
  trace set,get id;
}
```

Load the above code into UmpleOnline

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. class LightBulb
02. {
03.   Integer v = 0;
04.   status
05.   {
06.     On {
07.       entrv / { setV(1); }
08.       flip -> Off;
09.     }
10.     Off {
11.       entrv / { setV(2); }
12.       flip -> On;
13.     }
14.   }
15.   // trace whenever we enter/exit state On
16.   trace On;
17.   // trace whenever we exit state Off and
18.   // report value of attribute v
19.   trace exit Off record v;
20. }
21. // @@@skipphpcompile See issue 596 (PHP tracing causes issues)
22.
```

Load the above code into UmpleOnline

Another Example

view plain

```
01. class LightBulb
02. {
03.   status
04.   {
05.     On {
06.       flip -> Off;
07.     }
08.     Off {
09.       flip -> On;
```

```
    }
  }
  // trace any triggering of event flip
  trace flip;
}
// @@@skipphpcompile See issue 596 (PHP tracing causes issues)

```

Load the above code into UmpleOnline

Another Example

```
view plain

// this example will trace all states of
// State machine GarageDoor

class GarageDoor
{
    // UML state machine digram for a Garage door,
    // written in Umple
    status {
        Open { buttonOrObstacle -> Closing;  }
        Closing {
          buttonOrObstacle -> Opening;
          reachBottom -> Closed;
        }
        Closed { buttonOrObstacle -> Opening; }
        Opening {
          buttonOrObstacle -> HalfOpen;
          reachTop -> Open;
        }
        HalfOpen { buttonOrObstacle -> Opening; }
    }
    // trace whole state machine
    trace status;
}
// @@@skipphpcompile See issue 596 (PHP tracing causes issues)

```

Load the above code into UmpleOnline

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

// This example will record any addition/removal
// to supervisor assoc link

class Student {
  Integer id;
}

class Professor {
  1 -- * Student supervisor;

  trace supervisor;
}
// @@@skipphpcompile See issue 596 (PHP tracing causes issues)

```

Load the above code into UmpleOnline

Another Example

```
view plain

// This example will record any addition to
// assoc link 'supervisor'

class Student {
  Integer id;
}

class Professor {
  1 -- * Student supervisor;

  trace add supervisor;
}
// @@@skipphpcompile See issue 596 (PHP tracing causes issues)

```

Load the above code into UmpleOnline

Another Example

```
view plain
// This example will record any removal of any
// supervisor assoc link

class Student {
  Integer id;
}

class Professor {
  1 -- * Student supervisor;

  trace remove supervisor;
}
// @@@skipphpcompile See issue 596 (PHP tracing causes issues)

```

Load the above code into UmpleOnline

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. class Student
02. {
03.   name;
04.   Integer id;
05. 
06.   // trace code is injected in attribute setter
07.   // with specified precondition
08.   trace name where [name == "john"];
09. 
10.   // trace code is injected in attribute setter
11.   // with specified postcondition
12.   trace id giving [id > 9000];
13. 
14.   // this example traces attribute name whenever
15.   // its value is changed for 5 times
16.   trace name for 5;
17. }
18. 
19. class Professor {
20.   name;
21.   1 -- * Student supervisor;
22. 
23.   // trace code is injected in attribute setter
24.   // with role name cardinality condition
25.   trace name where [ supervisor cardinality > 10];
26. }
27. 
28. 
```

Load the above code into UmpleOnline

Record Clause

```
view plain
class Student
{
  name;
  Integer id;
  // trace attribute id and record provided string
  // message as part of  the trace output
  trace id record "i am tracing attribute id";

  status {
    s1 { e1 -> s2; }
    s2 {}
  }
  // trace whenever we enter/exit state "s1" and
  // record attribute name
  trace s1 record name;
}
```

Load the above code into UmpleOnline

Life Timeline

```
view plain
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 UmpleOnline

Syntax

traceCondition : [=conditionType:where|until|after|giving]? [[[constraintToken]]]

traceFor- : for [traceFor]

// A constraint is an expression optionally be surrounded in round brackets or negated
constraint- : [[constraintBody]] [[linkingOp]]? | [[constraintExpr]] [[linkingOp]]?

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

```
// this example shows how to trace
// non-generated method entry

class JavaMethod
{
  trace method();

  int method( int x ) {
    x += 5;
    return x;
  }
}
```

Load the above code into UmpleOnline

Method Exit

view plain

```
// this example shows how to trace generated
// method exit. In case of methods with a return
// statement, trace code is injected before
// the return

class JavaMethod
{
  trace exit method();

  int method( int x ) {
    x += 5;
    return x;
  }
}
```

Load the above code into UmpleOnline

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

// this example generates traces using the
// File tracer
tracer File;

class Student
{
  Integer id;
  trace id;
}
```

Load the above code into UmpleOnline

Java native logging API

```
view plain

// this example generates traces using the Java
// native logging API
tracer JavaAPI;

class Student
{
  Integer id;
  name;
```

```
  // trace id with default logging level info
  trace id;
  // trace name to logging level severe
  trace name logLevel severe;
}
```

Load the above code into UmpleOnline

Lttng tracer

```
view plain

// this example generates tracespoints using
// the Lttng tracer
tracer Lttng;

class Student
{
  Integer id;
  trace id;
}
```

Load the above code into UmpleOnline

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 UmpleOnline

log4j tracer with customized log4j xml configuration file

```
view plain
```

```
// this example generates traces using the
// log4j tracer
//
// Customized log4j xml configuration file
// is generated (log4j2.xml)
//
// Root logger level will be debug
//
// Info logging level will be written into
// console output
//
// Error logging level will be written into file
tracer log4j root=debug info = console error = file;

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 UmpleOnline

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.

Loading and saving

- **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: http://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.

- **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), and if you forget your URL, or someone messes up the saved contents at that 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 'Source to Copy' 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 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 'http://'*. 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: http://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.

  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, which is the default. 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), or a state diagram (S button or ctrl-S). To change the Graphviz diagrams, you have to edit Umple textually.

- **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 squares 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.

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
  - Control + G: Show GraphViz class diagram
  - Control + S: Show GraphViz state diagram
  - Control + L: Show Composite Structure diagram
  - Control + D: Show/Hide the diagram
  - Control + A: Toggle attributes

    - 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 http:// left off.

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 http://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.

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 rhe 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**.

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

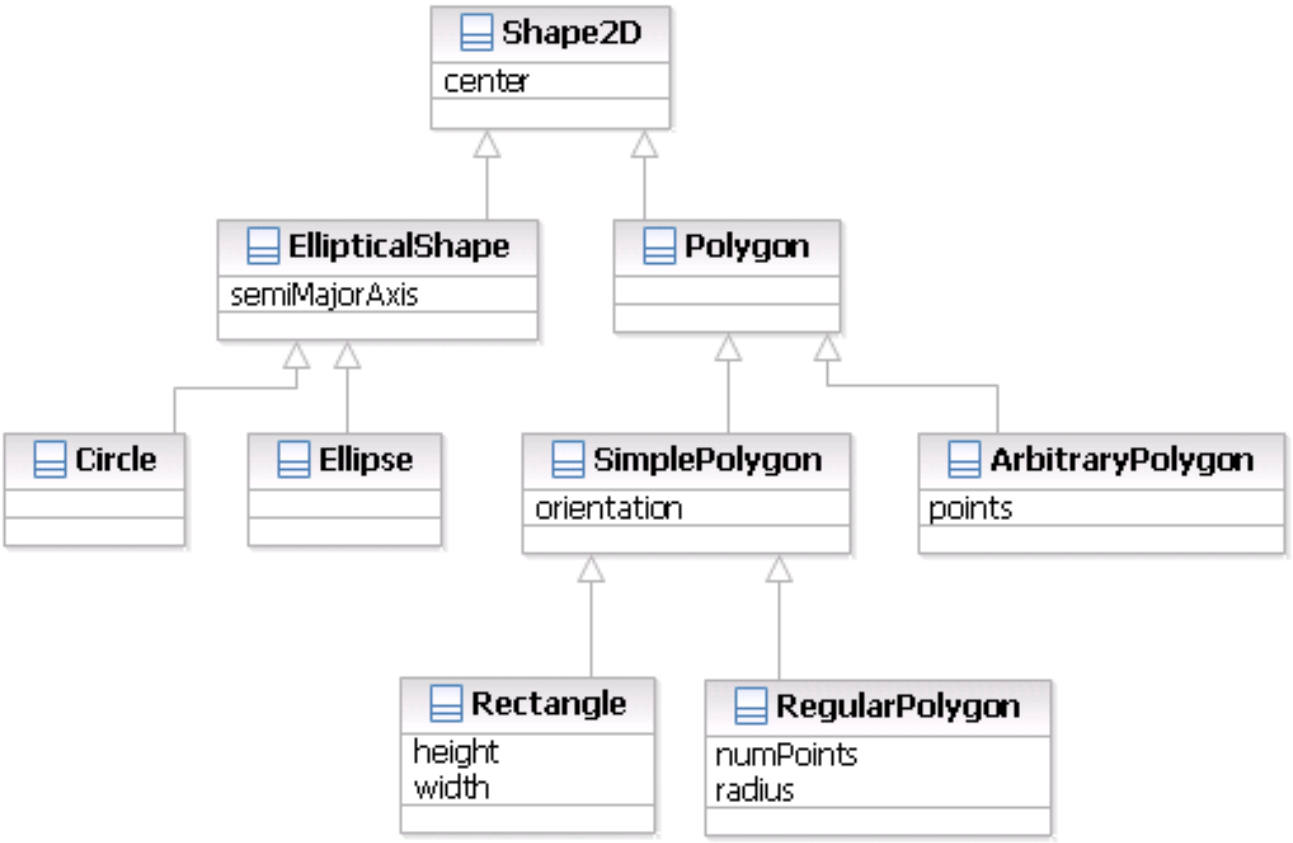
view plain

```
/*
This example demonstrates an example of using the
internal, depends, external, before and after
keywords to interface umple code to code in
another system.

The internal keyword is suitable for situations
where the developer doesn't want setter/getters
to be generated. GUI windows are an example of
such a case. A GUI window/unit contains several
components but the rest of the program may only
need the value stored within those components. So
it's a good practice to hide UI components from
classes other than the containing one.

*/

// required to make HelloInternals class a JFrame
external JFrame{}

class HelloInternals {
  // HelloInternals extends JFrame
  isA JFrame;

  // importing required classes
  depend javax.swing.JFrame;
  depend javax.swing.JLabel;

  /* messageLabel is a component of the frame;
  often we don't want sub-components of a GUI unit
  to be settable/gettable. By making them internal
  Umple will avoid generating setter/getter for
  messageLabel. Using lazy Umple will avoid adding
  a constructor parameter for this component */
  lazy internal JLabel messageLabel;

  // the contents of messageLabel
  String message;

  /* before getting the message, this code updates
  the message attribute using the text from
```

```
  messageLabel */
  before getMessage {
    message=messageLabel.getText();
  }

  /* after setting the messae, this code updates
  messageLabel to contain the newly updated
  message */
  after setMessage {
    messageLabel.setText(message);
  }

  // using after constructor is a good way of
  // initiating a GUI unit
  after constructor {
    getContentPane().setLayout(null);

    messageLabel=new JLabel(message);
    messageLabel.setBounds(10, 10, 200, 20);
    add(messageLabel);

    pack();
    setSize(250, 200);
  }
}
```

Load the above code into UmpleOnline

2D Shapes System



Example

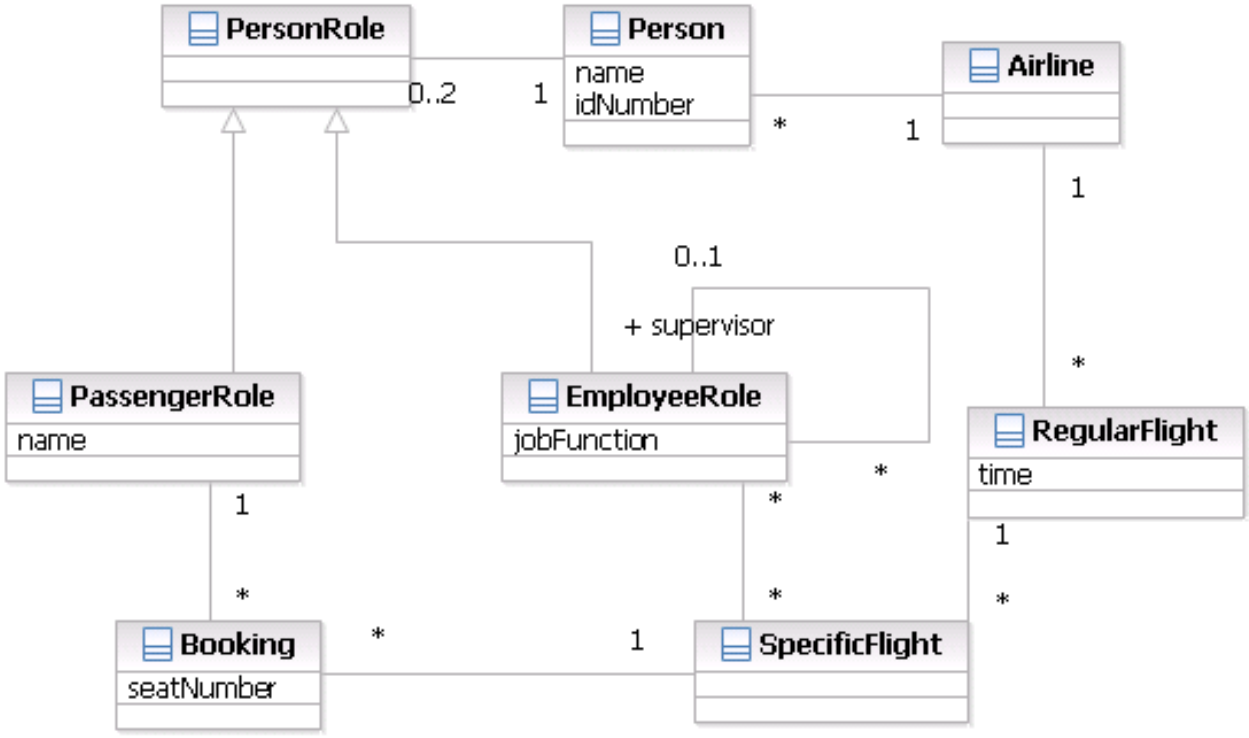
```
view plain
// 2D Shapes class hierarchy - sample UML class diagram in Umple
// From Book by Lethbridge and Laganiere, McGraw Hill 2004
// Object-
Oriented Software Engineering: Practical Software Engineering using UML and Java
// See http://www.lloseng.com

//Namespace for core of the system.
namespace Shapes.core;

class Shape2D {
 center;
}
//Abstract
class EllipticalShape {
 isA Shape2D;
 semiMajorAxis;
}
//Abstract
class Polygon {
 isA Shape2D;
}
class Circle {
 isA EllipticalShape;
}
```

```
class Ellipse{
 isA EllipticalShape;
}
class SimplePolygon {
 orientation;
 isA Polygon;
}
class ArbitraryPolygon {
 points;
 isA Polygon;
}
class Rectangle {
 isA SimplePolygon;
 height;
 width;
}
class RegularPolygon {
 numPoints;
 radius;
 isA SimplePolygon;
}
```

Load the above code into UmpleOnline

Airline System

A simple airline system that manages flight and passenger information



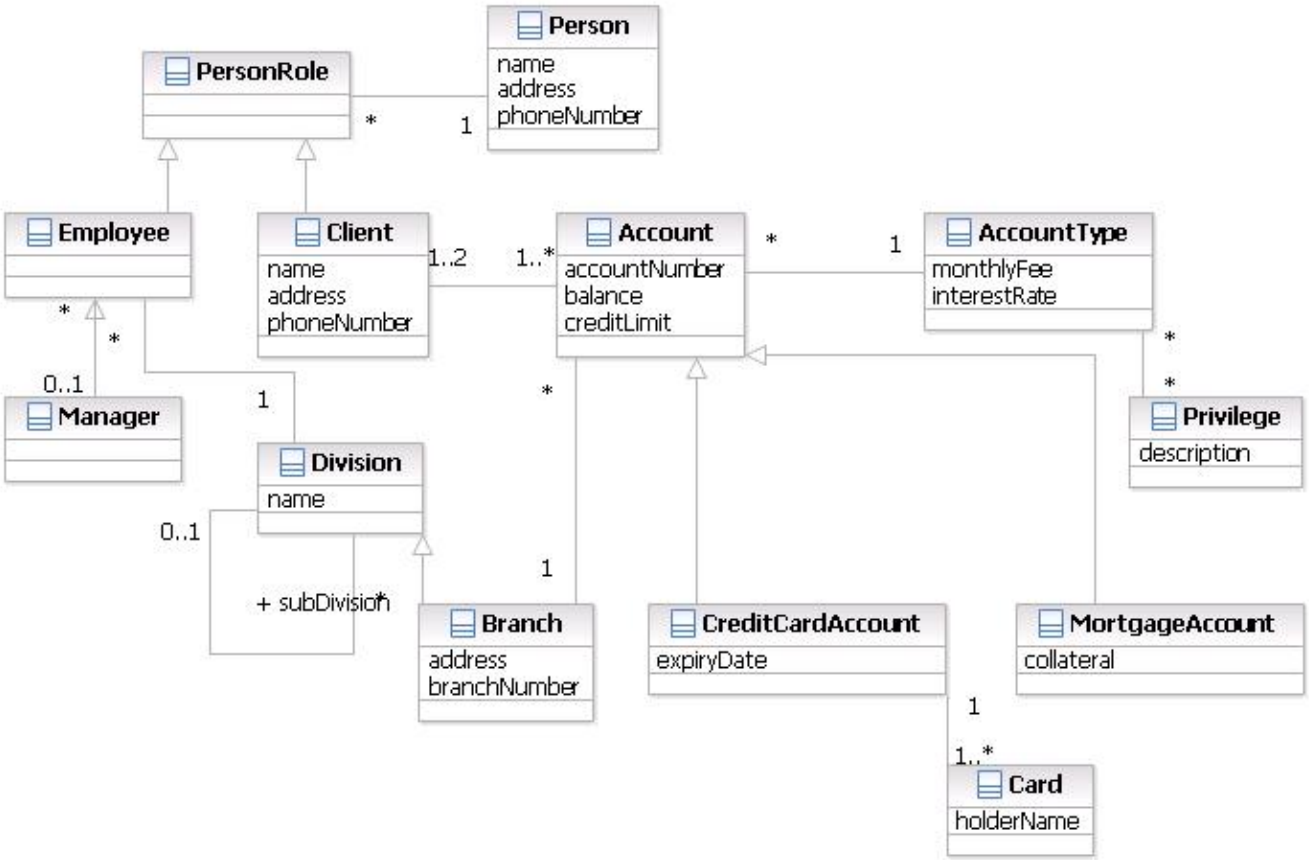
Example

```
view plain

// Airline system - sample UML class diagram in Umple
// From Book by Lethbridge and Laganiere, McGraw Hill 2004
// Object-
Oriented Software Engineering: Practical Software Engineering using UML and Java
// See http://www.lloseng.com

namespace Airline;

class Airline{
  1 -- * RegularFlight;
  1 -- * Person;
}

class RegularFlight{
  Time time;
  unique Integer flightNumber;
  1 -- * SpecificFlight;
}

class SpecificFlight{
  unique Date date;
}

class PassengerRole
{
```

```
  isA PersonRole;
  immutable String name ;
  1 -- * Booking;
}


class EmployeeRole
{
  String jobFunction ;
  isA PersonRole;
  * -- 0..1 EmployeeRole supervisor;
  * -- * SpecificFlight;
}

class Person
{
  settable String name;
  Integer idNumber;
  1 -- 0..2 PersonRole;
}

class PersonRole{}

class Booking{
  String seatNumber;
  * -- 1 SpecificFlight;
}
```

Load the above code into UmpleOnline

Banking System



Example

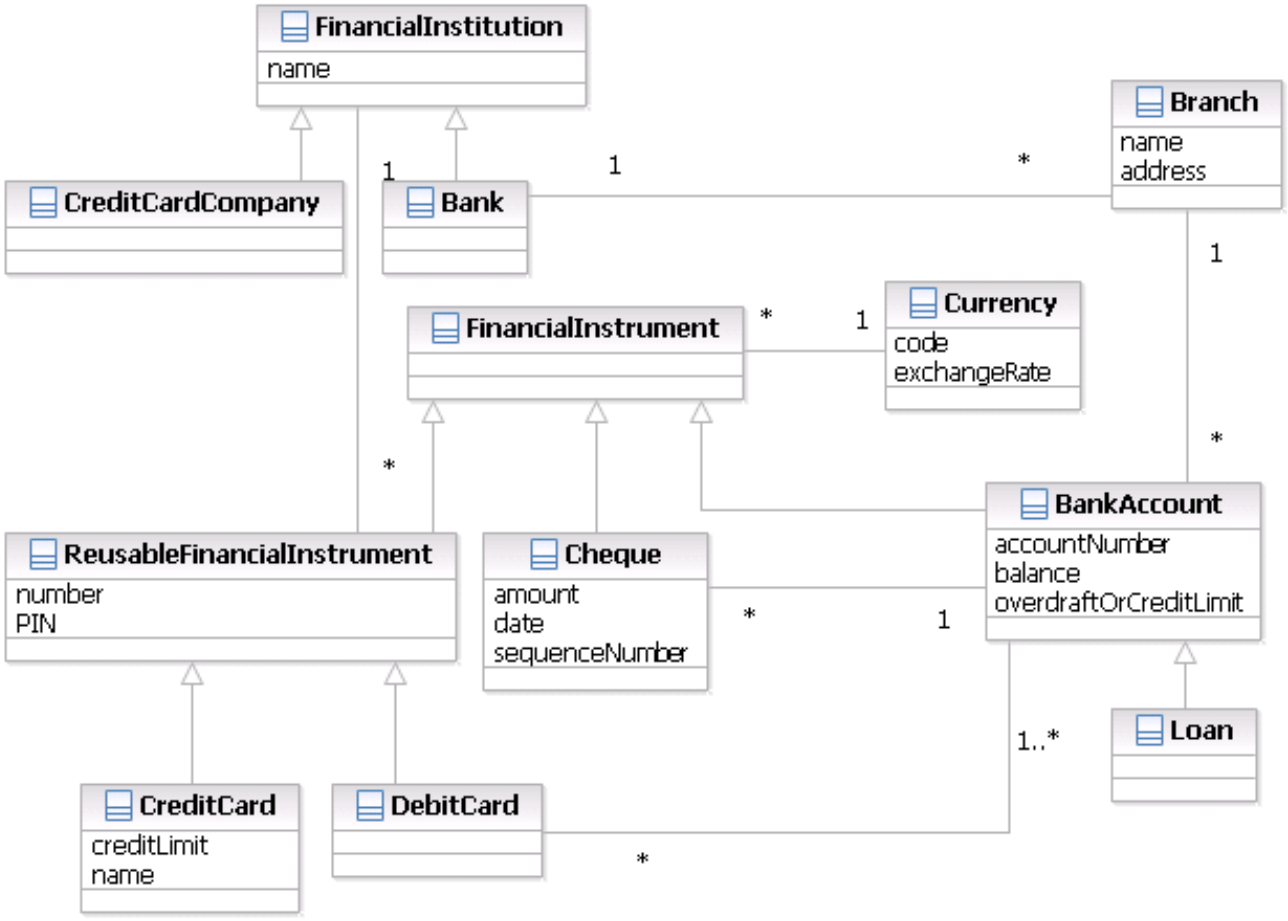
```
view plain

// Sample UML class diagram for a banking system, written in Umple

//Namespace for core of the system.
namespace BankingSystem.core.humanResources;
class PersonRole{}

class Person{
 name;
 address;
 phoneNumber;

 1 -- * PersonRole;
}

class Employee{
 isA PersonRole;
}

class Client
{
 isA PersonRole;
 name;
 address;
 phoneNumber;
 1..2 -- 1..* Account;
```

```
}

class Manager {
 isA Employee;
 0..1 -- * Employee;
}

//Accounts,  privileges, 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;
}

//Anything physically tangable
namespace BankingSystem.core.tangableResources;
class Card
{
 holderName;
}

class Branch {
 isA Division;
 address;
 branchNumber;

 1 -- * Account;
}

class Division{
 name;
```

```
34.
35.  1 -- * Employee;
36.  0..1 -- 0..* Division subDivision;
37. }
38.
39.
```

Load the above code into UmpleOnline

Banking System (b)



Example

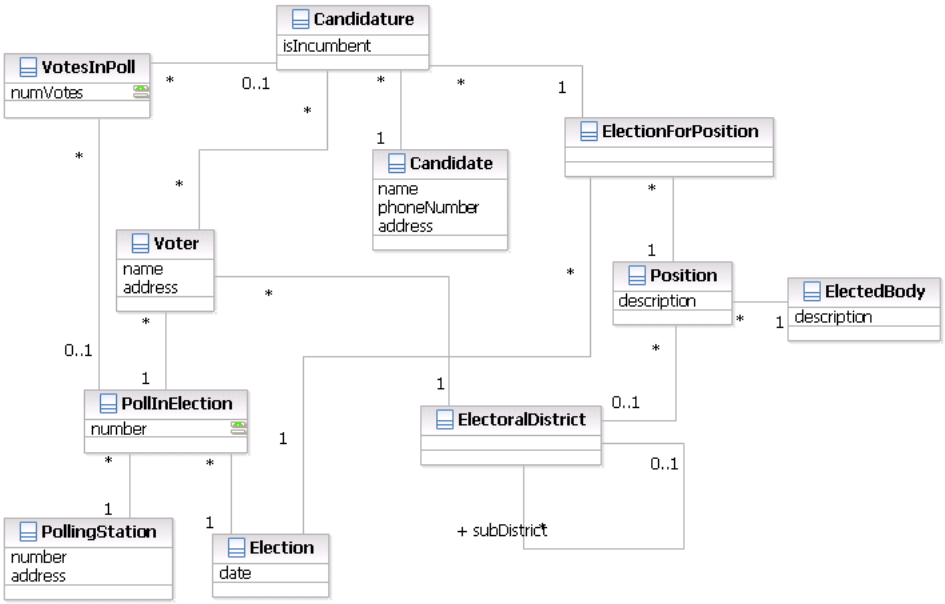
view plain

```
/*
  Banking System - sample UML class diagram written in Umple
  Last updated: February 21, 2011
*/
//Namespace for core of the system.
namespace BankingSystem.core;

class FinancialInstitution {
 name;
 1 -- * ReusableFinancialInstrument;
}

class CreditCardCompany{
 isA FinancialInstitution;
}

class Bank{
 isA FinancialInstitution;
 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;
}

class Branch
{
 name;
 address;
 1 -- * BankAccount;
}

class Loan{
 isA BankAccount;
}
```

Load the above code into UmpleOnline

Election System



Example

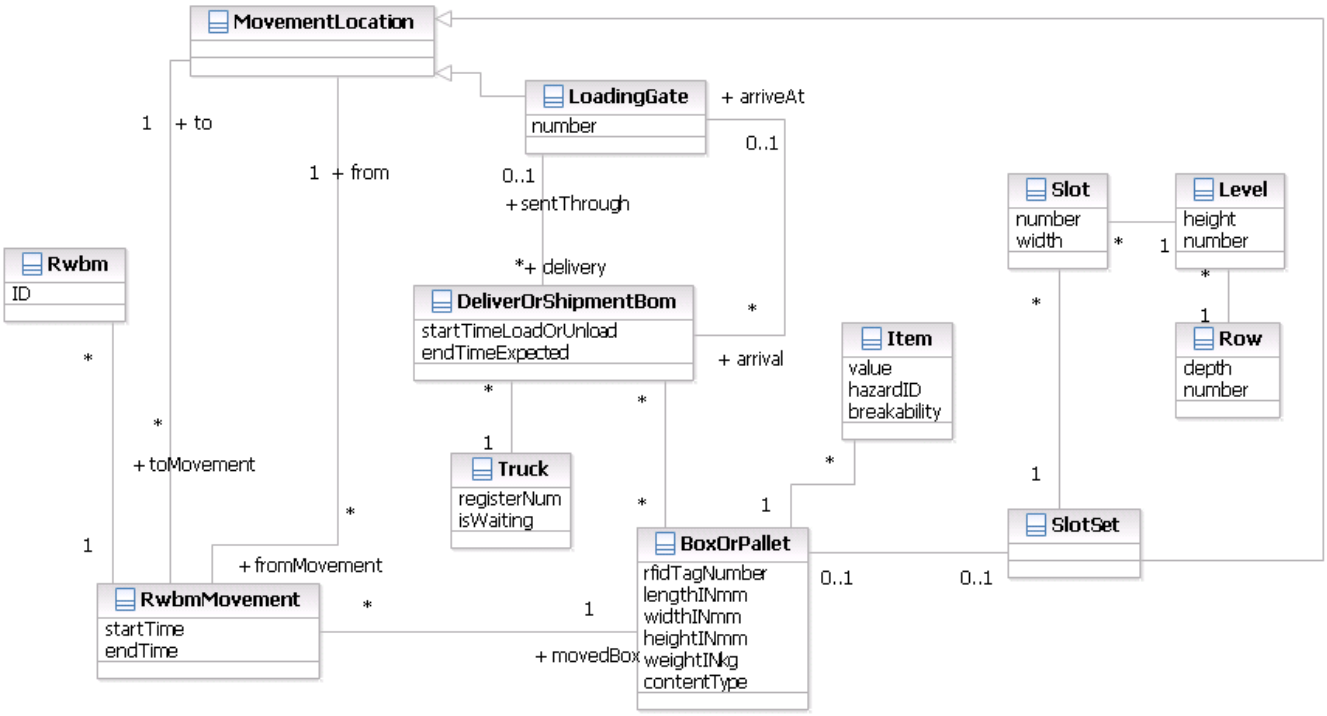
view plain

```
// UML class diagram for a system for managing elections, written in Umple
namespace electorial;

// association should be ->
class PollingStation {
 Integer number;
 address;

 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. once every four ye
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";
 }
}

class Candidate {
 name;
 Integer phoneNumber;
 address;

 1 -- * Candidature;
}

class PollInElection {
 Integer number;

 1 -- * Voter;
}

associationClass VotesInPoll{
 * Candidature;
 * PollInElection;
 Integer numVotes;
}

class Voter{
 name;
 address;
 * -- * Candidature;
}

class ElectoralDistrict{
 1 -- * Voter;
 0..1 -- * Position;
 0..1 -- * ElectoralDistrict subDistrict;
}
```

Load the above code into UmpleOnline

Warehouse System



Example

view plain

```
// UML class diagram of a system for managing a warehouse
namespace warehouse;

class MovementLocation{}

class SlotSet{
 isA MovementLocation;
 0..1 -- 0..1 BoxOrPallet;
 1 -- * Slot;
}

class Slot{
 Integer number;
 Double width;
}

class Level{
 Integer height;
 Integer number;
 1 -- * Slot;
}

class LoadingGate{

 Integer number;
 isA MovementLocation;
}

class Truck{
 Integer registerNum;
```

```
 Boolean isWaiting;

 1 -- * DeliverOrShipmentBom;
}
class Rwbm{
 Integer id;
}

class Item{
 Double value;
 Integer hazardID;
 Integer breakability;
}

class Row{
 Integer depth;
 Integer number;

 1 -- * Level;
}

//Note that here we have the choice to use
// Time or Double, if the customer requires Double due to
// some interoperability issues with a legacy syste, we can easily
// accomodate them and use Double instead of Time.
class RwbmMovement{
 Double startTime;
 Double endTime;

 * toMovement -- 1 MovementLocation to;
 * fromMovement -- 1 MovementLocation from;
 * -- 1 BoxOrPallet movedBox;
 1 -- * Rwbm;
}

// Both associations should be ->
class DeliverOrShipmentBom {
 Double startTimeLoadOrUnload;
 Double endTimeExpected;
 * delivery -- 0..1 LoadingGate sentThrough;
 * arrival -- 0..1 LoadingGate arriveAt;
}

class BoxOrPallet{
 Integer rfidTagNumber;
 Double lengthINmm;
 Double widthINmm;
 Double heightINmm;
 Double weightINkg;
 String contentType;

 * -- * DeliverOrShipmentBom;
 1 -- * Item;
}


```

Load the above code into UmpleOnline

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

```
// UML class diagram that models a canal as a
// network of segments, with Locks. See the
// State Machine section for a state machine for
// each lock
class CanalNetwork {
  name;
  0..1 -- * CanalNetwork subNetwork;
  0..1 -- * Craft activeVessels;
  * -- * SegEnd;
}
```

```
class SegEnd {
  name;
  GPSCoord location;
}

class Segment {
  Float waterLevel; // m above sea level
  1..* -- 2 SegEnd;
}

class Lock {
  isA Segment;
  Float maxLevel;
  Float minLevel;
}

class Bend {
  isA SegEnd;
}

class EntryAndExitPoint {
  isA SegEnd;
}

class MooringPoint {
  isA SegEnd;
}

class Obstacle {
  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 {
```

```
69.     Float lattitide;
70.     Float longitude;
71. }
72.
```

Load the above code into UmpleOnline

Canal Lock State Machine

Canal Lock State Machine

See also the class diagram and requirements for the canal system.

Example

view plain

```
// UML state machine diagram for a canal lock,
// represented in UML

class Lock
{
  Boolean boatGoingDown = false;
  Boolean boatGoingUp = false;
  Boolean boatBlockingGate = false;

  lockState {
    BothDoorsClosedLockFull {
      // Waiting for boat
      boatRequestsToEnterAndGoDown
        -> OpeningUpperGate;
      boatRequestsToEnterAndGoUp
        -> LoweringWater;
    }

    OpeningUpperGate {
      upperGateFullyOpen -> UpperGateOpen;
    }

    UpperGateOpen {
      entry / {setBoatGoingUp(false);}
      boatInLockRequestingToGoDown
        -> / {setBoatGoingDown(true);}
        ClosingUpperGate;
      after(180000) [!boatBlockingGate]
        -> ClosingUpperGate;
    }

    ClosingUpperGate {
      upperGateFullyClosed [boatGoingDown]
        -> LoweringWater;
      upperGateFullyClosed [!boatGoingDown]
        -> BothDoorsClosedLockFull;
    }

    LoweringWater {
      waterLevelMatchesDownStream
        -> OpeningLowerGate;
    }


    BothDoorsClosedLockEmpty {
      // Waiting for boat
      boatRequestsToEnterAndGoUp
        -> OpeningLowerGate;
```

```
      boatRequestsToEnterAndGoDown
        -> RaisingWater;
    }

    OpeningLowerGate {
      lowerGateFullyOpen -> LowerGateOpen;
    }

    LowerGateOpen {
      entry / {setBoatGoingDown(false);}
      boatInLockRequestingToGoUp
        -> / {setBoatGoingUp(true);}
          ClosingLowerGate;
      after(180000) [!boatBlockingGate]
        -> ClosingLowerGate;
    }

    ClosingLowerGate {
      lowerGateFullyClosed [boatGoingUp]
        -> RaisingWater;
      lowerGateFullyClosed [!boatGoingUp]
        -> BothDoorsClosedLockEmpty;
    }

    RaisingWater {
      waterLevelMatchesUpStream
        -> OpeningUpperGate;
    }
  }
}
```

Load the above code into UmpleOnline

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

```
// Umple class diagram representing card games
// in the Oh Hell and Whist family

class Card {
  suit;  // "hearts", "clubs", "diamonds", "spades"
  rank;  // A23456789JQK
  Boolean isJoker; // suit and rank are null for a Joker
}

class CardSet {
  // Could be a complete deck or a trick
  1 -- * Card;
}

class NotDealtCardSet {
  // This would be initialized with a complete 52-card set
  // at the start of each game
  // During dealing, the cards would be distributed to players
  isA CardSet;
}

class Trick {
  // A group of cards with one contributed by each player, and
  // eventually won by a player
  // One is built after each player has played a card
  isA CardSet;
}

class Player {
   // These are the people playing.
   name;

   // A hand is dealt at the start of a game
   // The cards a player has in his or her hand
   0..1 -- * Card hand;

   // The tricks won by a player this game
   1 -- * Trick;

   // All hands and tricks are cleared at the end of each game
   // ready for dealing again

   // The following is used only in Oh Hell
   Integer currentTricksBid;
}
```

```
46.
47.  class ScoringTeam {
48.    // for Oh Hell, it is each player for himself, so there is one player
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 winner
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 the
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 UmpleOnline

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

```
// UML Class diagram representing a system for taking online orders for Pizza

class Account
{
  emailAddress;
  contactPhoneNumber;
  password;
  name;
  address;
  preferredCredCard;
  expiryDate;
  0..1 -- * Order;
}

class Order
{
  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;
}

class Driver
{
  name;
}
```

Load the above code into UmpleOnline

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

```
// UML class diagram in Umple showing a simple
// system for managing the data maintained by a
// community association

class Facility
{
  id;
  type;
  openPeriod;
  1 -- * Booking booked;
}

class Resident
{
  name;
  Boolean under18;
  emailAddress;
  telephoneNumber;
  executivePosition;
}

class CommunityResidence
{
  streetAddress;
  Integer apartmentNumber;
  feePaidToDate;
  1 -- * Resident;
}

class Event
{
  Date date;
  Time time;
  Float fee;
  name;
```

```
}

class Booking
{
  Date date;
  Time startTime;
  Time endTime;
  Float feePaid;
  Boolean isReservedForFCFS;
  * -- 0..1 Resident;
}

class Under18Event
{
  isA Event;
}

class Rink
{
  isA Facility;
}

class TennisCourt
{
  isA Facility;
}

class CommunityAssociation
{
  singleton;
  1 -- * CommunityResidence;
  1 -- * Facility;
  1 -- * Event;
}

```

Load the above code into UmpleOnline

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

```
// UML class diagram in Umple showing the data managed by
// a co-operative education system that has to place students with employers

class Program
{
  name;
  1..* -- * Student;
}

class Application
{
  * -- 1 Student;
  * -- 1 Job;
  1..* -- 0..1 Interview;
  1 -- 0..1 Offer;
}

class Job
{
  description;
  location;
  Date startDate;
  Date endDate;
  * -- 1..* Program;
}

class Employer
{
  name;
  1 -- * Job;
  0..1 -> * TimeSlot;
}

```

```
class Interview
{
  location;
  0..1 -> 0..1 TimeSlot;
}

class TimeSlot
{
  startTime;
  endTime;
  status;
}

class Student
{
  Integer stNum;
  1 -- 1 Transcript;
  0..1 -> * TimeSlot;
}

class Resume
{
  text;
  0..1 -- 1 Student;
}

class Transcript
{
  text;
}

class Offer
{
  ranking;
  Boolean accepted;
}
```

Load the above code into UmpleOnline

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

```
// UML class diagram of a vending machine
class VendingMachine {
   singleton;
   1 -- * Dispenser slots;
}

class Dispenser {
  Character row;
  Character column;
}

// It is possible for a dispenser to have
// one product type 'in front' and another in behind
class ProductInDispenser {
   * -- 1 Dispenser;
   * -- 1 ProductType;
   Integer numberProductsOfThisTypeLeft;
   }

class ProductType {
   Integer pricePerUnit; // cents
}

class VendingMachine {
   1 -- * CoinHolder;
}

class CoinType {
  Integer value; // cents
  Integer weight; // grams
  Integer diameter; // micrometers
}

class CoinHolder {
   * -- 1 CoinType canHold;
   Integer numCoinsCapacity;
   Integer currentNumberOfCoins;
```

```
}

// Every product purchase is recorded
class ProductTransaction {
  * -- 1 ProductInDispenser;
  1 -- * CoinHolderTransaction; // coins entered and change given
}

// The number of coins transferred to or from a CoinHolder
class CoinHolderTransaction {
   Integer numCoinsInOrOut; // negative if used to provide change
   * -- 1 CoinHolder sourceOrDestination;
}

//Positions used for diagram display
class VendingMachine
{
  position 50 29 119 45;
  position.association Dispenser:slots    VendingMachine 120,23 0,8;
  position.association CoinHolder__VendingMachine 27,45 5,0;
}

class Dispenser
{
  position 290 29 149 80;
}

class ProductInDispenser
{
  position 166 157 291 63;
  position.association ProductInDispenser    ProductType 10,63 30,0;
  position.association Dispenser__ProductInDispenser 217,0 56,80;
}

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    ProductTransaction 112,46 179,0;
  position.association ProductInDispenser__ProductTransaction 3,0 93,63;
}

class CoinHolderTransaction
```

```
{
  position 283 501 201 63;
  position.association CoinHolder:sourceOrDestination__CoinHolderTransaction 16,0 2
}
```

Load the above code into UmpleOnline

Another Example

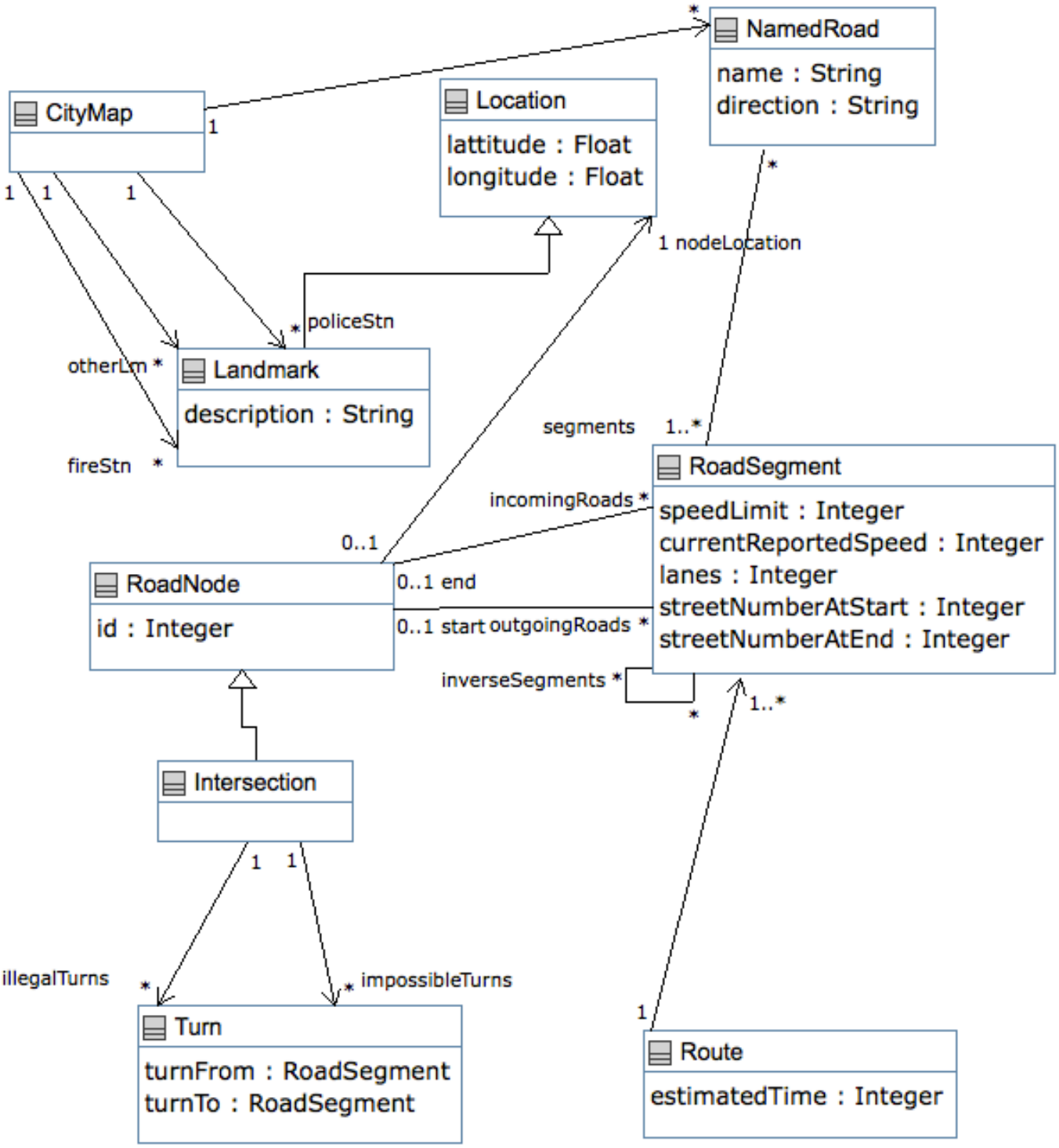
view plain

```
// UML state machine diagram of a vending machine
// There is also a class diagram example separately
class VendingMachine
{
  controlUnit {
    ReceivingMoney {
      enterCoin  -> DeliveringItem;
    }

    DeliveringItem {
      deliveryComplete -> ReturningMoney;
    }

    ReturningMoney {
      returningComplete -> Waiting;
    }

    Waiting {
      pressSelection  -> ReceivingMoney;
    }
  }
}
```

Load the above code into UmpleOnline

Routes And Locations

This example has been built for the Comparing Modeling Approaches Workshop



Example

view plain

```
// RoutesAndLocations.ump  Class diagram represented in Umple
//
// Author: Timothy C. Lethbridge

namespace routesAndLocations;
```

```
96.
97.  /*
98.
99.  This file describes reusable model elements that are used in the
00.  bCMS Crisis Management System. It is necessary to have a rudimentary
01.  Graphical Information System such as described in this submodel as part
02.  of bCMS in order to plan routes and to track where crises are. This file
03.  has the necessary classes.
04.
05.  */
06.
07.  /*
08.   * A CityMap contains the precompiled streets and landmarks
09.   * This must be read in from the database on system initialization
10.   */
11.  class CityMap {
12.    singleton;
13.    0..1 -> * Landmark fireStn;
14.    0..1 -> * Landmark policeStn;
15.    0..1 -> * Landmark otherLm;
16.    0..1 -> * NamedRoad;
17.  }
18.
19.  /*
30.   * A Location describes a place such as an intersection, landmark, bend in a road, etc.
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 { } touristDestination { } other { } }
43.  }
44.
45.  /*
46.   * RoadNodes are places where RoadSegments connect. A node with just one incoming
47.   * and one outgoing is used to handle changes in direction, e.g. as a road turns
48.   * a corner, changes in speed limit, changes in number of lanes, and other factors.
49.   * When there is more than one outgoing or incoming, the node represents points
50.   * where traffic flow can split and join.
51.   * A crossroads is one type. Entry into a roundabout would be another.
52.   * Highway merges and exits are other kinds as are parking lots and entries
53.   * into or out of fire stations..
54.   *
55.   * RoadSegments leaving the city lead to no RoadNode
56.   * RoadSegments entering the city come from no RoadNode
57.   */
58.  class RoadNode {
59.    Integer id;
60.     // Could be at a landmark
61.    0..1 -> 1 Location nodeLocation;
62.    0..1 end -- * RoadSegment incomingRoads;
63.    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 because
 * for example, sometimes several numbered highways share a segment
 */
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 can drive.
 * The ends of each segment have been modeled using RoadNodes.
 * Note that distance can be calculated from the speed limit, and locations
 */
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 map
 * taking into account speed, congestion, etc.
 */
class Route {
```

```
22.     // Ordered list of segments
23.     0..1 -> 1..* RoadSegment;
24.
25.     // Time seconds at current flow speeds
26.     Integer estimatedTime;
27.   }
28.
29.
```

Load the above code into UmpleOnline

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

```
// CoordinationStateMachine.ump
// This UML state machine diagram in Umple is for the
// bCMS car crash management system
// See http://cserg0.site.uottawa.ca/cma2012/CaseStudy.pdf
// and http://www.cs.colostate.edu/remodd/v1/content/repository-model-driven-development-remodd-overview
// Author: Timothy C. Lethbridge

/*
 * State machine for co-ordination of the
 * crisis, as seen from one SC. If
 * communication is operating properly the
 * other coordinators' state machines will
 * generally be in the same states and
 * substates.
 *
 * Modelled given Section 4 of the requirements
 * This is a mixin to class crisis.
 */
class Crisis {

  // Number of ms allowed to negotiate routes,
  // establish credentials
  Integer timeoutInMs = 20000;

  // True if we are the initiator of the crisis
  // request, false otherwise
  Boolean thisSideInitiator;
  Boolean thisSideInitiatorOfTermination;

  Boolean routeAgreedYet;

  // True of we are the PSC, false of we are
  // the FSC
  Boolean police;

  // Top level state machine
  crisisCoordinationStage {

    // 0. When initiated the crisis is in
    // noCrisis state
    // It has been created but not yet
    // populated, awaiting initiation
    // of coordination and data..
    noCrisis {
      entry / {setThisSideInitiator(false);}
      // We are initiating and requesting the
```

```
    // other side to respond
    initiateCrisis / {
      setThisSideInitiator(true);
      sendInitiateCrisisRequestToOtherSC();
      } -> establishingCommunication;

    // We are responding to a request from
    // the other SC
    receiveInitiateCrisisRequest ->
      establishingCommunication;
  }

  // 1. Establish communication and
  // identification when a crisis begins
  establishingCommunication {
    entry / {sendSecureCredentials();}
    secureCredentialsConfirmed ->
      managingCrisis;

    // Return to idle after some delay if
    // other side initiated and did not
    // respond. It might have been a
    // hacker. Alternatively if there is a
    // real crisis, normal manual processes
    // would happen.
    after (timeoutInMs)
      [!isThisSideInitiator()] -> noCrisis;
  }

  // Managing crisis involves continually
  // exchanging information
  // Negotiating or renegotiating routes and
  // other crisis details
  // These are handled concurrently.
  managingCrisis {
    // A crisis is underway. We initially
    // start with empty data
    // which will be populated over time
    entry / {initiateEmptyCrisis(); }

    // 2.4.5.6. Exchanging details about what
    // each knows about the crisis
    // So the information each coordinator
   // has is the same.
    exchangingDetails {
      // Requirement Scenario 2
      ourUpdateToCrisisData
        / {sendCrisisData();}
        -> exchangingDetails;
      receiveCrisisData
        / {updateCrisisData();}
        -> exchangingDetails;

      // Requirement Scenario 4
      ourVehicleDispatched
        / {sendVehicleDispatch();}
        -> exchangingDetails;
      receiveVehicleDispatched
```

```
        / {updateTheirDispatch();}
        -> exchangingDetails;

      // Requirement Scenario 5
      ourVehicleArrived
        / {sendVehicleArrived();}
        -> exchangingDetails;
      receiveVehicleArrived
        / {updateTheirArrival();}
        -> exchangingDetails;

      // Requirement Scenario 6
      ourVehicleMetObjective
        / {sendVehicleMetObjective();}
        -> exchangingDetails;
      receiveMetObjective
      / {updateTheirMetObjective();}
      -> exchangingDetails;

      // Requirement 5.a - breakdown
      breakdown / {dispatchAndUpdateOther();}
        -> exchangingDetails;

      // Requirement 5.b -
      // congestion/blockage
      // TO DO

      // Requirement 5.c escalate crisis -
      // renegotiate routes
      // TO DO


    } // End of exchangingDetails
      //concurrent substate
    ||

    // 7. Both parties must agree to close
    // the crisis; either can initiate
    crisisEndManagement {

      // Normal substate of crisisEndManagement - no end in sight yet
      ongoingCrisis {

        // We could initiate termination
        initiateTermination / {
          setThisSideInitiatorOfTermination(true);
          sendTerminationRequestToOtherSC();
          } -> waitingForTerminationConfirmation;
        receiveTerminationRequestFromOtherSC
        ->
          waitingForUserTerminationAgreement;
      }

      waitingForUserTerminationAgreement {
        do {confirmWithUserToTerminate();}
        confirmTermination -> tearDown;
        after (timeoutInMs) -> ongoingCrisis;
      }
```

```
53.
54.       // Substate of where we are waiting for
55.       // the other end to agree
56.       waitingForTerminationConfirmation {
57.         receiveTerminationConfirmation
58.           -> tearDown;
59.
60.         // If the other side has not agreed,
61.         // we keep the crisis active
62.         after (timeoutInMs) -> ongoingCrisis;
63.       }
64.
65.       // End of crisis
66.       tearDown {
67.         entry / {deleteCrisis();}
68.         -> noCrisis;
69.       }
70.     } // End of crisisEndManagement
71.       // concurrent substate
72.     ||
73.
74.     // 3. Negotiating route plan
75.     negotiatingRoutePlan {
76.
77.       // Negotiation happens in parallel with
78.       // reporting timeout
79.       negotiation {
80.         entry / {setRouteAgreedYet(false);}
81.         informOfNumberOfVehicles
82.           / {sendNumberOfVehicles();}
83.           -> negotiation;
84.         receiveNumberOfVehicles [isPolice()]
85.           -> planningRoute;
86.         receiveRouteProposal [!isPolice()]
87.           -> approvingRoute;
88.
89.         // Requirement 3.3.a2.a1
90.         receiveNoAgreeableRoute
91.           [!isPolice()] -> noRouteAgreement;
92.
93.         // The PSC plans the route -- only
94.         // PSC can be in this state
95.         planningRoute {
96.           do {planRoute();}
97.           routePlanComplete
98.             / {sendPlanToFSC(); }
99.             -> awaitingRouteApproval;
100.
101.          // Requirement 3.3.a2.a1 - no more
102.          // possible routes
103.          noMoreRoutes
104.            / {sendNoMoreRoutesToFSC(); }
105.            -> noRouteAgreement;
106.        }
107.
108.        // The FSC approves the route  --
109.        // only FSC can be in this state
110.        approvingRoute {
```

```
        do {userConfirmRouteAcceptable();}
        routeAcceptable
          / {sendApprovalToPSC();}
          -> routeAgreed;
        routeUnacceptable
          / {sendDisapprovalToFSC();}
          -> negotiation;
      }

      // The PSC awaits for approval from
      // the FSC. Only the PSC can be in
      // this state
      awaitingRouteApproval {
        receiveApprovalFromFSC
          -> routeAgreed;

        // Requirement 3.3.a - FSC
        // disagrees
        receiveDisapprovalFromFSC / {
          addRouteToDisapprovedChoices();
          } -> planningRoute;
      }

      routeAgreed {
        entry / {setRouteAgreedYet(true);}
      }

      noRouteAgreement {
        // requirement 3.3.a2.a1
      }
    } // End of Negotiation concurrent
      // substate of negotiatingRoutePlan
    ||
    managingTimeliness {
      timeAcceptable {
        // Requirement 3.a1. Negotiations
        // are taking excessive time
        after (timeoutInMs)
          [!isRouteAgreedYet()]
          -> timeUnacceptable;
      }
      timeUnacceptable {
        // Although negotiations may yet
        // complete, we are going to send
        // out our own vehicles because too
        // much time has passed
        do {
          promptAndLogReasonForTimeout(); }
      }
    } // End of managingTimeiness
      //concurrent substate of
      // negotiatingRoutePlan
  } // End of NegotiatingRoutePlan
    //concurrent substate
 } // End of Managing Crisis top level state
} // End of crisisCoordinationStage state
  // machine
} // End of state machine mixin for
```

```
29.    // Crisis
30.
31.
```

Load the above code into UmpleOnline

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 at the University of Ottawa with funds from research grants and donations. Should these research grants cease and donations not suffice, then support for Umple may cease unless others take over responsibility for hosting. Similarly, the code is hosted on Github. If Github decides to cease hosting projects for free, Umple 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, and these are likely to 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 purely 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 and thorough 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 at the University of Ottawa. 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 etc.).

Data saved automatically in this way remains stored at the University of Ottawa for up to two days. 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 two 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.

If you choose 'Save as URL' then your model is stored for an extended period, subject to deletion rules described below. Such a file can be edited and deleted by anyone to whom you give the URL, or by anyone who guesses the URL.

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 do intend to install tools to survey users about their experiences with Umple. We will likely use external tools such as SurveyMonkey for such surveys; people who complete such surveys would then be subject to the privacy rules of such external tools. Users would be requested to give informed consent prior to taking such a survey, and such informed consent would first be 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 others.

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 two days. 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. 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.

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.

- Java
- Php
- Ruby
- C++ is under development

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
   |Php
   |RTCpp
   |SimpleCpp
   |Ruby
   |Cpp
   |Json
   |StructureDiagram
   |Yuml
   |Violet
   |Umlet
   |Simulate
   |TextUml
   |Scxml
   |GvStateDiagram
   |GvClassDiagram
   |GvFeatureDiagram
   |GvClassTraitDiagram
   |GvEntityRelationshipDiagram
   |Alloy
   |NuSMV
   |NuSMVOptimizer
   |Papyrus
   |Ecore
   |Xmi
   |Xtext
   |Sql
   |Umple
   |UmpleSelf
   |USE
   |Test
   |SimpleMetrics
   |Uigu2] ( [=suboptionIndicator:-s
   |--suboption] " [**suboption] " )* ;

generate_path : generate [=language:Java
   |Php
   |RTCpp
```

|SimpleCpp
|Ruby
|Cpp
|Json
|StructureDiagram
|Yuml
|Violet
|Umlet
|Simulate
|TextUml
|Scxml
|GvStateDiagram
|GvClassDiagram
|GvClassTraitDiagram
|GvFeatureDiagram
|GvEntityRelationshipDiagram
|Alloy
|NuSMV
|NuSMVOptimizer
|Papyrus
|Ecore
|Xmi
|Xtext
|Sql
|UmpleSelf
|USE
|test] " [**output] " [=override:--override
|--override-all]? ([=suboptionIndicator:-s
|--suboption] " [**suboption] ")* ;

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 http://grammar.umple.org

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source license found at:
// http://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- : [[glossary]]
| [[generate]]
| [[distribute]]
| [[generate_path]]
| [[filter]]
| [[useStatement]]
| [[namespace]]
| [[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
    |Php
    |RTCpp
    |SimpleCpp
    |Ruby
    |Cpp
    |Json
    |StructureDiagram
    |Yuml
    |Violet
    |Umlet
    |Simulate
    |TextUml
    |Scxml
    |GvStateDiagram
    |GvClassDiagram
    |GvFeatureDiagram
```

```
  |GvClassTraitDiagram
  |GvEntityRelationshipDiagram
  |Alloy
  |NuSMV
  |NuSMVOptimizer
  |Papyrus
  |Ecore
  |Xmi
  |Xtext
  |Sql
  |Umple
  |UmpleSelf
  |USE
  |Test
  |SimpleMetrics
  |Uigu2] ( [=suboptionIndicator:-s
  |--suboption] " [**suboption] " )* ;


generate_path : generate [=language:Java
  |Php
  |RTCpp
  |SimpleCpp
  |Ruby
  |Cpp
  |Json
  |StructureDiagram
  |Yuml
  |Violet
  |Umlet
  |Simulate
  |TextUml
  |Scxml
  |GvStateDiagram
  |GvClassDiagram
  |GvClassTraitDiagram
  |GvFeatureDiagram
  |GvEntityRelationshipDiagram
  |Alloy
  |NuSMV
  |NuSMVOptimizer
  |Papyrus
  |Ecore
  |Xmi
  |Xtext
  |Sql
  |UmpleSelf
  |USE
  |test] " [**output] " [=override:--override
  |--override-all]? ( [=suboptionIndicator:-s
  |--suboption] " [**suboption] " )* ;
```

```
// 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] */


//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_classes.grammar

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source license found at:
// http://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]] | [[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]]
  | [[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]]
| [[classDefinition]]
| [[position]]
| [[displayColor]]
| [[invariant]]
| [[softwarePattern]]
| [[depend]]
| [[association]]
| [[inlineAssociation]]
| [[singleAssociationEnd]]
| [[attribute]]
| [[stateMachine]]
| [[enumerationDefinition]]
| ;
| [[extraCode]]

innerClass : [[innerStaticClass]] | [[innerNonStaticClass]]

innerStaticClass : static [[classDefinition]]

innerNonStaticClass : inner [[classDefinition]]

// 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]]
   | [[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
   |Alloy
   |UmpleSelf] ( , [=codeLang:Java
   |Alloy
   |RTCpp
   |SimpleCpp
   |Cpp
   |Php
   |Ruby
   |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]]
   | [[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/umple_traits.grammar

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source license found at:
// http://umple.org/license

traitDefinition : trait [name] [[traitParameters]]? { [[traitContent]]* }

traitContent- : [[mixsetDefinition]]
   | [[testCase]]
   | [[requiredModelElements]]
   | [[comment]]
   | [[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/umple_fixml.grammar

//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 second part of the Umple grammar is available at
// https://github.com/umple/umple/blob/master/cruise.umple/src/umple_patterns.grammar

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/umple_state_machines.grammar

// 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]]
  | [[state]]
  | [[trace]]
  | [[mixsetDefinition]]
  | [=ll]
  | [[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]]
   | [=changeType:+| - |- |*]? [[stateEntity]]
   | [[standAloneTransition]]
   | [[concreteMethodDeclaration]]
   | [**extraCode]

stateEntity- : [=||]
   | [[mixsetDefinition]]
   | [[entryOrExitAction]]
   | [[autoTransition]]
   | [[transition]]
   | [[activity]]
   | [[state]]
   | [[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/umple_traces.grammar

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source license found at:
// http://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|Tri
[=option:Time|time|Thread|thread|File|file|Line|line|Trace|trace|Trigger|trigger|TraceFile|tracefile|TraceLine|traceline|Tri
//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/umple_constraints.grammar

// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source license found at:
// http://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 part of the Umple grammar is available at
// https://github.com/umple/umple/blob/master/cruise.umple/src/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]])*

// 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]] | [[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/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/umple_layout.grammar

// 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 second part of the Umple grammar is available at
// https://github.com/umple/umple/blob/master/cruise.umple/src/umple_exceptions.grammar

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:
// http://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 part of the Umple grammar is available at
// https://github.com/umple/umple/blob/master/cruise.umple/src/umple_filter.grammar
//
// Copyright: All contributors to the Umple Project
// This file is made available subject to the open source license found at:
// http://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/umple_mixsets.grammar

//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]] | [[mixsetInlineDefinition]])

mixsetInnerContent- : { [[extraCode]] }

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.

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 machines and other features

API GENERATED IN ALL CASES

void delete() removes object, correctly handling cascade deletion and referential integrity

any methods in the code are emitted as-is; they are made public if this is not declared

API GENERATED FROM ATTRIBUTES

T is the type of the attribute (String if omitted); note that Umple builtin types such as Integer become native t
z is the attribute name

Note that the API for array attributes is shown as a column in the associations table further down

Umple concept		basic	initialized	lazy	defaulted	lazy im
Umple notation		T z;	T z = val;	lazy T z;	defaulted T z = val;	lazy im
Querying value						
T getZ()	return the value	yes	yes	yes	yes	yes
boolean isZ()	return true if value is true	if T Boolean	if T Boolean	if T Boolean	if T Boolean	if T Boo
boolean equals(Object)	s equal to another?					
Mutating						
boolean setZ(T)	mutates the attribute	yes	yes	yes	yes	yes; tru
boolean resetZ()	restores original default				yes	
Other						
T getDefaultZ()	returns original default				yes	
int hashCode()	Likely unique value					
constructor args		T				
initial value		constructor	val	null/0/false	val	null/0/fa

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 names unless a role name is
r is a role name used when referring to W (optional, except in reflexive associations)

Umple concept		? To many	? To many with role	optional	? To one	require
Umple notation		-- * W;	-- * W r;	-- 0..1 W;	-- 1 W;	-- 1..* W
Querying link(s)						
W getW()	return the W			yes; W or null	yes	
W getW(index)	pick a specific linked W	yes	yes; getR(index)			yes
List<W> getWs()	ges immutable list of links	yes	yes; getR()			yes
boolean hasWs()	is cardinality > 0?	yes	yes; hasR()			yes
int indexOfW(W)	look up specific W -1 if none	yes	yes; … ofR(W)			yes
int numberOfWs()	cardinality	yes	yes; … ofR()			yes
Mutating						
boolean setW(W)	add a link to existing W			yes	if * -- or 0..1 --	
W addW(args)	construct a new W and adds link	if * -- * W	if 1 -- * W r			

boolean addW(W)	add a link to existing W	yes	yes; addR(W)			yes
boolean setWs(W…)	add a set of links					yes
boolean removeW(W)	remove link to W if possible	yes	yes; removeR(W)			yes
Other						
static int minimumNumberOfWs()	lower bound	yes	yes; … ofR()			yes
static int maximumNumberOfWs()	upper bound if > 1					
static int requiredNumberOfWs()	if multiplicity if > 1					
boolean isNumberOfWsValid()	breaching constraints					yes; tru
constructor args					W (W args if 1 -- 1)	
initial value		empty	empty	empty	constructor va	empty

API GENGERATED FROM STATE MACHINES

sm is the name of the state machine

Umple concept		Sm with events	Ems with no events			
enum getSm()	current state for statemachine sm	yes	yes			
String getSmFullName()	current state for statemachine sm	yes	yes			
enum getSmS2()	state of substate s2	yes				
boolean e()	event method e (for all events)	yes	yes			
boolean setSm(enum)	set state directly		yes			
Substates allowed		yes				
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
1.  // This example generates the warning message
2.  class X {
3.    singleton;
4.    Integer x;
5.  }
6.
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
1.  // The following shows how to avoid the warning
2.  class X {
3.    singleton;
4.    lazy Integer x;
5.  }
6.
```

Load the above code into UmpleOnline

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

// This example generates the warning message
class X {
  singleton;
  0..1 -- 1 Y;
}

class Y {
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// The following shows how to avoid the warning
class X {
  singleton;
  0..1 -- 0..1 Y;
}

class Y {
}

```

Load the above code into UmpleOnline

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

// This example generates the warning
class X3lazy {
  lazy Integer a = 1;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// This shows how the warning can be removed.
class X3lazy {
  Integer a = 1;
}

```

Load the above code into UmpleOnline

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
1.  // This example generates the error message
2.  class X {
3.    1 -- 0..1..2 Y;
4.  }
5.
6.  class Y {
7.  }
8.
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
1.  // The following shows how to avoid the error
2.  class X {
3.    1 -- 0..2 Y;
4.  }
5.
6.  class Y {
7.  }
8.
```

Load the above code into UmpleOnline

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

// This example generates the error message
class X {
  1 -- * Y;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// The following shows how to avoid the error
class X {
  1 -- * Y;
}
class Y {}

```

Load the above code into UmpleOnline

Another Solution to The Above So the Message No Longer Appears

```
view plain

// It is also possible to designate that
// the class Y is external in the following manner
class X {
  1 -- * Y;
}
external Y {}

```

Load the above code into UmpleOnline

Another Example

```
view plain

// The following will also generate the error message
class X {}
class Y {}

```

```
association {
  1 X -- * Z;
}

```

Load the above code into UmpleOnline

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

// This example generates the error message
class X {
  defaulted Integer a;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// The following shows how to avoid the error
class X {
  defaulted Integer a = 5;
}

```

Load the above code into UmpleOnline

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;
  kev { name }
  key { id }
}

```

Load the above code into UmpleOnline

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 UmpleOnline

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

// This example generates the error message
class A {}

class B {}

associationClass C {
  * A;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// The following shows how to avoid the error
class A {}

class B {}

associationClass C {
  * A;
  * B;
}

```

Load the above code into UmpleOnline

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 UmpleOnline

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 UmpleOnline

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
// This example generates the warning message
class X {
  singleton;
}

class Y {
   0..1 -- * X;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
// The following shows how to avoid the warning
class X {
  singleton;
}

class Y {
   0..1 -- 1 X;
}

```

Load the above code into UmpleOnline

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

// This example generates the error message
class X {
  isA X;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// The following shows how to avoid the error
class Y {}

class X {
  isA Y;
}

```

Load the above code into UmpleOnline

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 UmpleOnline

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 UmpleOnline

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

// The following example generates
// the error message
class X {
   Integer x;
}

class Y {
  String s;
  immutable;
  0..1 -> 0..1 X;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// One way to solve this is to make
// the association point to
// another immutable class
// But note that this only works for
// one-way associations
class X {
   Integer x;
   immutable;
}

class Y {
  immutable;
  String s;
  0..1 -> 0..1 X;
}

```

Load the above code into UmpleOnline

Another Solution to The Above So the Message No Longer Appears

```
view plain

// Another solution is to not require
// the association to be immutable,
// just the attributes. But this may
// not match the intended semantics.
```

```
95. class X {
96.    immutable Integer x;
97. }
98.
99. class Y {
00.    immutable String s;
01.    0..1 -- 0..1 X;
02. }
03.
```

Load the above code into UmpleOnline

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

```
// This example generates message 14
class B {
  a;
}

class X14immut {
  isA B;
  immutable;
  b;
}

```

Load the above code into UmpleOnline

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

// The following example shows how to
// generate this warning.
// Removing the immutable keyword will
// solve the problem.
class X {
  immutable;
  Integer y;
  sm {
    S1 {
    }
  }
}

```

Load the above code into UmpleOnline

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

```
// The following example generates the error message
// A solution is to make class X immutable
class X {
  a;
}
class Y {
  immutable;
  isA X;
}
```

Load the above code into UmpleOnline

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

// The following example generates the error message
// Change the -- to -> to solve the problem
class Y {
  immutable;
  1 -- * Z;
  isA X;
}

class Z {
  immutable;
  v;
}

```

Load the above code into UmpleOnline

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 UmpleOnline

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.

Example

```
view plain

// The following example shows how
// to generate error 19.
// The solution is to at least one
// role name one or both of the Y ends
// and at least one role name on one
// of both of the X ends.
class X {
}

class Y {
  0..1 -> * X;
  0..1 -> * X;
}
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// Using role names rn1 and rn2 to
// avoid error 19
class X {
}

class Y {
  1 -- * X;
  1 rn2 -- * X rn1;
}
```

Load the above code into UmpleOnline

Another Example

```
view plain
// Example of an association class
// definition that generates error 19
// The solution is to add a role name
// such as menteeAssignments between
// the first * and Member

class Member {
  name;
}

associationClass Assignment {
  Date dateEstablished;
  * Member mentor;
  * Member mentee;
}
```

Load the above code into UmpleOnline

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

// This example generates the message
class X20classintfarrow {
  1 -- * IA;
}

interface IA {}

```

Load the above code into UmpleOnline

Another Example

```
view plain

// This example generates the message
class X20classintfarrow {
  1 <- * IA;
}

interface IA {}

```

Load the above code into UmpleOnline

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
// The following example shows
// how to generate this error.
class X {
   0..1 -- 0..1 X;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
// The following example shows how
// to fix the error if this is an
// asymmetric association.
class X {
   0..1 -- 0..1 X right;
}

```

Load the above code into UmpleOnline

Another Solution to The Above So the Message No Longer Appears

```
view plain
// The following example shows how to fix
// the error if this is a symmetric association.
class X {
   0..1 self right;
}

```

Load the above code into UmpleOnline

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 UmpleOnline

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 UmpleOnline

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;
}

class B {}

```

Load the above code into UmpleOnline

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

// This example generates the error message
class X {
  1 -- * Y sorted {endDate};
}

class Y {
   Date endDate;
}

```

Load the above code into UmpleOnline

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
// This example generates the error message
class X {
   * -- 1 Y sorted {nom};
}

class Y {
  name;
}

```

Load the above code into UmpleOnline

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

```
// The following example has error 26
// since the key has 'a' twice
class X26dupiteminkey {
  a;
  b;
  key {a, b, a}
}
```

Load the above code into UmpleOnline

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

// This example generates the message
class X27itemkeynotdef {
  a;
  key {b};
}

```

Load the above code into UmpleOnline

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

// This example generates the message
class X28attrconsrnotdef {
  Integer a;
  [b > 5]
}

```

Load the above code into UmpleOnline

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

// This example generates the message
class X29attrtypeconstr {
  a;
  [a > 5]
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// The following shows how to avoid the message
class X29attrtypeconstr {
  Integer a;
  [a > 5]
}

```

Load the above code into UmpleOnline

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

```
// This example generates the message
// Namespace b, encountered later, will
// be the namespace of the class
namespace a;

class X30redefnamespace {
}

namespace b;

class X30redefnamespace {
}
```

Load the above code into UmpleOnline

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

// This example generates the message because
// namespace n is never used
namespace n;
namespace m;

class X21namespacenotused {}

```

Load the above code into UmpleOnline

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
1.  // This example generates the error message
2.
3.  class Course {
4.    * -- * Course courses;
5.  }
6.
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
1.  // The following shows how to avoid the error
2.
3.  class Course {
4.    * -- * Course course_s ;
5.  }
6.
```

Load the above code into UmpleOnline

Another Example

```
view plain
1.  // This example generates the error message
2.
3.  class Course {
4.    * courses -- * Course;
5.  }
6.
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
1.  // The following shows how to avoid the error
2.
3.  class Course {
```

```
  * course_s -- * Course;
}

```

Load the above code into UmpleOnline

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

// This example generates the warning
// since T is not defined
class S {
  isA T;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// This example solves the problem
// by defining T as an external class
// Umple will not generate any code for T,
// but now knows it exists
class S {
  isA T;
}
external T {};

```

Load the above code into UmpleOnline

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

// The following will generate this error

class P1 {}
class P2 {}

class Sub {
  isA P1, P2;
}

```

Load the above code into UmpleOnline

Another Example

```
view plain

// The following is another way of
// formulating the same declarations,
// also resulting in the error

class P1 {}
class P2 {}

class Sub {
  isA P1;
  isA P2;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// If all but one of the parents is
// declared as an Interface, the problem is solved

interface P1 {}
class P2 {}

class Sub {
  isA P1;
  isA P2;
}

```

Load the above code into UmpleOnline

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

// The following will generate this warning

class X {
  const String A; //same as const String a = "";
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// The following will resolve the issue

class X {
  const String A = "Something Interesting";
}

```

Load the above code into UmpleOnline

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

```
// The following will generate this warnbing

class X {
  1 -> * B first;
  1..3 -> * B second;
  0..2 -> * B third;
}

class B {}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

view plain

```
// The following solves the problem
// by either making the association unndirected
// or making the bounds have soft constraints

class X {
  0..1 -> * B first;
  * -> * B second;
  0..2 -- * B third;
}

class B {}

```

Load the above code into UmpleOnline

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
1.  // This example generates error 37
2.  class X {
3.    const Y A;
4.  }
5.
6.  class Y {
7.    Integer a;
8.  }
9.
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
1.  // This example resolves error 37
2.
3.  class X {
4.    const Y A=new Y(3);
5.  }
6.
7.  class Y {
8.    Integer a;
9.  }
0.
```

Load the above code into UmpleOnline

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

//The error is generated because of
//the name conflict between attr
//and the attribute canSetAttr,
//which is automatically generated
//for an immutable attribute

//The same occurs for the attr2
//and nextAttr2
class A {
  immutable attr;
  canSetAttr;

  autounique attr2;
  nextAttr2;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The error can be fixed by
//modifying the names of the
//attributes
class A {
  immutable attr;
  otherAttr;

  autounique attr2;
  otherAttr2;
}

```

Load the above code into UmpleOnline

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

```
1.  //Since B is not defined,
2.  //the warning will be generated
3.  interface A {
4.    isA B;
5.  }
6.
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

view plain

```
1.  //The problem can be solved by
2.  //defining B
3.  interface A {
4.    isA B;
5.  }
6.
7.  interface B {}
8.
```

Load the above code into UmpleOnline

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

// In this example a singleton class
// has subclasses and it generates an error
class Airplane
{
  singleton;
}
class F16
{
    isA Airplane;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//I this example the singleton class
// does not have a subclass and is correct
class Airplane
{

}
class TheAirplane
{
  singleton;
}
class F16
{
    isA Airplane;
}

```

Load the above code into UmpleOnline

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

//Causes the error due to the return type of
//method1 being Integer
class A
{
  queued Integer method1() { /* implementation */ }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//No error is raised, as both methods
//have a void return type
class A
{
  queued method1() { /* implementation */ }
  queued void method2() { /* implementation */ }
}

```

Load the above code into UmpleOnline

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

```
// Classes A, B and C are in the
// default namespace. Class D is in
// namespace m. All these classes are
// linked to each other via associations,
// therefore, and because they are not all
// in the same namespace, an import text
// will be generated. However, many programming
// languages do not support import from
// the default namespace so the namespace of
// C will become m because it is linked to
// D, and the same goes for B because C is
// now in namespace m, and then A is placed
// in m because B is now in m.

class A{}
class B{*--*A;}
class C{*--* B;}
namespace m;
class D{*--* C;}
namespace -;


// The same issue occurs when an
// entity in a non-default namespace
// extends or implements an entity
// in the default namespace

interface E{}
interface F{isA E;}
interface J{isA F;}
namespace n;
interface K{isA J;}


// Here an import text will be generated for D in X
// because D is in a different namespace
// This feature is not yet supported for interfaces
// and the import text will not be generated for K in Y

namespace p;
class X{isA D;}
// interface Y{isA K;}

```

Load the above code into UmpleOnline

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 UmpleOnline

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 UmpleOnline

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

// This example generates the message
class X {
  Integer z = 1;
  key { z }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// The following shows how to avoid the message
class X {
  Integer z;
  key { z  }
}

```

Load the above code into UmpleOnline

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<OtherClass> otherClassGroup;
06. }
07. class OtherClass {}
08.
```

Load the above code into UmpleOnline

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 A {
04.   OtherClass[] otherClassGroup;
05. }
06. class OtherClass {}
07.
08.
```

Load the above code into UmpleOnline

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 A {
04.   0..1 -- * OtherClass;
05. }
06. class OtherClass {}
07.
```

Load the above code into UmpleOnline

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
// This example generates the message
class A {
  id;
  key { }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
// The following shows how to avoid the message.
class A {
  id;
  key { id }
}

```

Load the above code into UmpleOnline

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

//The error will be issued, attr
//and Attr both generating setAttr
//and getAttr causing a name conflict
class A {
  attr;
  Attr;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The following example is a
//way to fix the error
class A {
  attr;
  attr2;
}

```

Load the above code into UmpleOnline

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
1.  // This example generates the message
2.  class A{
3.    String test1(){return("hello world");}
4.    String test1(){return("dlrow olleh");}
5.  }
6.
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
1.  // The following shows how to avoid the message.
2.  class A{
3.    void test1(){}
4.    void test2(){}
5.  }
6.
```

Load the above code into UmpleOnline

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

// This example generates the error message
class X {
  sm {
    s1 {
      e1 -> s2;
    }
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// This example solves the problem
class X {
  sm {
    s1 {
      e1 -> s2;
    }
    s2 {}
  }
}

```

Load the above code into UmpleOnline

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

// This example generates the error message
class X {
  sm {
    s1 {
      e1(String s) -> s2;
    }
    s2 {
      e1(Float f) -> s2;
    }
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// This example solves the problem
class X {
  sm {
    s1 {
      e1(String s) -> s2;
    }
    s2 {
      e1(String s) -> s2;
    }
  }
}

```

Load the above code into UmpleOnline

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

// This example generates the message
// because the state machine a
// clashes with the attribute a
class X52assattnamestatemachine1 {
  a;
  a {
    s1 {}
    s2 {}
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// The sample generates the message
// because the state machine a
// clashes with the association role name a
class X52assattnamestatemachine2 {
  1 -- 0..1 B a;
  a {
    s1 {}
    s2 {}
  }
}
class B {}

```

Load the above code into UmpleOnline

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
// This example generates the message
class X53conctoplevel {
  sm {
    s1 {
      do {System.out.println(
        "Reached s1");}
    }
    ||
    s2
    {
      do {System.out.println(
        "Reached s1");}
    }
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
// The following shows how to avoid the message
class X53conctoplevel {
  sm {
    s0 {
      s1 {
        do {System.out.println(
          "Reached s1");}
      }
      ||
      s2
      {
        do {System.out.println(
          "Reached s1");}
      }
    }
  }
}

```

Load the above code into UmpleOnline

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

```
1.  class X {
2.    sm {
3.      s1 {
4.        e-> s2;
5.        e-> s3;
6.      }
7.      s2 {}
8.      s3 {}
9.    }
0.  }
1.
```

Load the above code into UmpleOnline

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 UmpleOnline

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

//The warning will be generated
//as there are no events to be
//queued in the state machine sm
class A {
  queued sm {
    s0 {
    }
    s1 {
    }
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The following avoids the warning
//by adding an event to the state machine
class A {
  queued sm {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }
}

```

Load the above code into UmpleOnline

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

//The warning will be generated
//as there are no events to be
//pooled in the state machine sm
class A {
  pooled sm {
    s0 {
    }
    s1 {
    }
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The following avoids the warning
//by adding an event to the state machine
class A {
  pooled sm {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }
}

```

Load the above code into UmpleOnline

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

```
//The error will be generated as A
//contains a queued, pooled and
//regular state machine at once
class A {
  pooled sm1 {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }

  queued sm2 {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }

  sm3 {
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

view plain

```
//The error can be fixed by having each
//state machine in a different class
class A {
  pooled sm1 {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }
}

class B {
```

```
  queued sm2 {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }
}

class C {
  sm3 {
  }
}
```

Load the above code into UmpleOnline

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

```
//The error will be generated as
//class A contains both a queued
//and a pooled state machine
class A {
  pooled sm1 {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }

  queued sm2 {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }
}
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

view plain

```
//The error can be fixed by having each
//state machine in a different class
class A {
  pooled sm1 {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }
}

class B {
  queued sm2 {
    s0 {
      e -> s1;
    }
```

```
18.     s1 {
19.     }
20.   }
21. }
22.
```

Load the above code into UmpleOnline

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

```
//The error will be generated as
//class A contains both a regular
//and a pooled state machine
class A {
  pooled sm1 {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }

  sm2 {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

view plain

```
//The error can be fixed by having each
//state machine in a different class
class A {
  pooled sm1 {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }
}

class B {
  sm2 {
    s0 {
      e -> s1;
    }
```

```
18.     s1 {
19.     }
20.   }
21. }
22.
```

Load the above code into UmpleOnline

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

```
//The error will be generated as
//class A contains both a regular
//and a queued state machine
class A {
  queued sm1 {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }

  sm2 {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

view plain

```
//The error can be fixed by having each
//state machine in a different class
class A {
  queued sm1 {
    s0 {
      e -> s1;
    }
    s1 {
    }
  }
}

class B {
  sm2 {
    s0 {
      e -> s1;
    }
```

```
18.     s1 {
19.     }
20.   }
21. }
22.
```

Load the above code into UmpleOnline

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

//The warning is generated for
//sm due to the use of the unspecified
//event
class A {
  pooled sm {
    s0 {
      unspecified -> s1;
    }
    s1 {
    }
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The warning can be avoided by using
//a queued state machine to be able
//to use the unspecified event
class A {
  queued sm {
    s0 {
      unspecified -> s1;
    }
    s1 {
    }
  }
}

```

Load the above code into UmpleOnline

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

//The error occurs as state machine
//sm uses the reserved name H as a
//state name
class A {
  sm {
    H {
    }
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The error can be avoided by
//using a non reserved state
//name
class A {
  sm {
    s1 {
    }
  }
}

```

Load the above code into UmpleOnline

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

//The following generates the error,
//OtherState not existing within sm
class A {
  sm {
    s1 {
      e -> OtherState.H;
    }
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The error can be resolved by
//declaring OtherState in sm
class A {
  sm {
    s1 {
      e -> OtherState.H;
    }
    OtherState {
      OtherStateA {
      }
      /* ... other substates */
    }
  }
}

```

Load the above code into UmpleOnline

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

//The error is generated because
//s2 does not contain a substate
//and therefore no history to
//be managed
class A {
  sm {
    s1 {
      e -> s2.H;
    }
    s2 {

    }
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The following does not generate
//the error as s2 contains substates
//and can keep the last substate in
//memory
class A {
  sm {
    s1 {
      e -> s2.H;
    }
    s2 {
      s2a {
      }
      s2b {
      }
      /* ... more substates */
    }
  }
}

```

Load the above code into UmpleOnline

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

//The following generates the warning
//as s1 could designate both s1 and s2.s1
//By default, it is s1
class A {
  sm {
    s1 {
      e -> s1;
    }

    s2 {
      s1 {
      }
    }
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The warning does not occur
//by removing the ambiguity
class A {
  sm {
    s1 {
      e -> s2.s1;
    }

    s2 {
      s1 {
      }
    }
  }
}

```

Load the above code into UmpleOnline

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

//The warning is generated for
//s3, as no transition leads to it
class A {
  sm {
    s1 {
      e -> s2;
    }
    s2 {
    }
    s3 {
    }
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The warning can be avoided
//by adding a transition with s3
//as destination state
class A {
  sm {
    s1 {
      e -> s2;
    }
    s2 {
      e2 -> s3;
    }
    s3 {
    }
  }
}

```

Load the above code into UmpleOnline

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

```
1.  //The warning is generated due
2.  //to the transition destination
3.  //of e -> s3.s2a not being defined in sm
4.  class A {
5.    sm {
6.      s1 {
7.        e -> s3.s2a;
8.      }
9.      s2 {
0.        s2a {
1.        }
2.      }
3.    }
4.  }
5.
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

view plain

```
1.  //The following example does
2.  //not generate the warning
3.  //as s2.s2a is found in sm
4.  class A {
5.    sm {
6.      s1 {
7.        e -> s2.s2a;
8.      }
9.      s2 {
0.        s2a {
1.        }
2.      }
3.    }
4.  }
5.
```

Load the above code into UmpleOnline

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

//The warning is generated due
//to the auto-transition to s2
//conflicting with the auto-
//transition to s3
class A {
  sm {
    s1 {
      entry /{doSomething();}
      -> s2;

      do {doSomethingElse();}
      -> s3;
    }
    s2 {
    }
    s3 {
    }
  }
  void doSomething() {}
  void doSomethingElse() {}
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The warning is avoided here
//by keeping only one auto-
//transition
class A {
  sm {
    s1 {
      entry /{doSomething();}
      do {doSomethingElse();}
      -> s2;
    }
    s2 {
    }
  }
  void doSomething() {}
  void doSomethingElse() {}
}
```

.7.

[Load the above code into UmpleOnline]

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

// This example generates the message
class A{
  String test1(){return("hello world");}
  int test1(){return(1);}
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// The following shows how to avoid the message.
class A{
  void test2(){}
  Integer test1(){}
}
// @@@skipcompile no return from test1

```

Load the above code into UmpleOnline

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

// The following shows how to generate the warning

class InvalidFinalState {
  status{
    on{
      turnoff -> off;
      powerOff-> FINAL;
    }

    off{
      turnOn -> on;
    }

    final FINAL{
      entry/{entry();}
      do{exe();}
      reboot -> on;
      nestedSm {
        s1 {
          -> s2;
        }
        s2 {

        }
      }
      exit/{exit();}
    }
  }
}
// @@@skipcompile no method entry written for line 15

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// The following shows how to avoid the warning

class X {
  status{
    on{
      turnoff -> off;
      powerOff-> FINAL;
    }
    off{
```

```
      turnOn -> on;
    }
    final FINAL{
      entry/{entry();}
    }
  }
}
// @@@skipcompile no method entry written line 13

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

view plain

```
// The following shows how to avoid the warning

class X {
  status{
    on{
      turnoff -> off;
      powerOff-> FINAL;
    }
    off{
      turnOn -> on;
    }
    final FINAL{

    }
  }
}

```

Load the above code into UmpleOnline

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
// The following example generates the error
// "s1" has two parallel state machines that are named "t1"

class X {
  sm{
    s1{
      t1{
        goT2-> t2;
      }
      t2{ }
      ||
      t1{
        goT4-> t4;
      }
      t4{}
    }
  }
}
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
// The following example shows how to avoid the error

class X {
  sm{
    s1{
      t1{
        goT2-> t2;
      }
      t2{ }
      ||
      t3{
        goT4-> t4;
      }
      t4{}
    }
  }
}
```

Load the above code into UmpleOnline

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

// The following example generates the error
// "s1" has a state named "Final"

class X {
  sm {
    s1 {
      Final {}
    }
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// The following example does not generate the error

class X {
  sm {
    s1 {
      goToFinal-> Final;
    }
  }
}

```

Load the above code into UmpleOnline

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
1.  class X {
2.    timer {
3.      s1 {
4.        after(1) -> s2;
5.      }
6.      s2 {
7.      }
8.    }
9.  }
0.
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
1.  class X {
2.    sm  timer {
3.      s1 {
4.        after(1) -> s2;
5.      }
6.      s2 {
7.      }
8.    }
9.  }
0.
```

Load the above code into UmpleOnline

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

//Class A contains an association
//with abstract class B, causing
//the error by using mandatory
//multiplicity
class A {
  1 -- 1 B;
}

abstract class B{
  abstract;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The following association does not
//cause the error
class A {
  0..1 -- 0..1 B;
}

abstract class B{
  abstract;
}

```

Load the above code into UmpleOnline

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

//The attribute attr cannot be initialized
//as the values are not correct
class A {
  String[] attr = {invalid.values};
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//attr can now be initialized
class A {
  String[] attr = {"valid", "values"};
}

```

Load the above code into UmpleOnline

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

```
// The following shows how to generate the warning

class ConflictingModifiers {
  status{
    on{
      turnoff -> off;
      public void printState() {
          System.out.println("on");
      }
    }

    off{
      private void printState() {
          System.out.println("off");
      }
    }
  }
}
```

Load the above code into UmpleOnline

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

// For derived attributes remove list indicator []
// For list initializer, there should be a semicolon on the end
class Office {
  Integer number;
  String[] emails = {"abc@umple.org", "def@umple.org"}
  Integer [] incomingNumber = {765432, 987654}
}
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//emails and incomingNumber can now be initialized
class Office {
  Integer number;
  String[] emails = {"abc@umple.org", "def@umple.org"};
  Integer [] incomingNumber = {765432, 987654};
}
```

Load the above code into UmpleOnline

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

//Class A does not contain
//the appropriate attribute,
//the constraint causes the error
class A {
  [model: A has attribute named attr]
}
// @@@skipcompile

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The class A now contains an
//attribute with the appropriate name
class A {
  attr;
  [model: A has attribute named attr]
}

```

Load the above code into UmpleOnline

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

//The error is raised, A does
//not contain an attribute of
//type B
class A {
  [model: A has attribute of B]
}

class B {
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//A now contains an
//attribute of type B
class A {
  B attr;
  [model: A has attribute of B]
}

class B {
}

```

Load the above code into UmpleOnline

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

//The error occurs as A
//does not respect the constraint
//Alternative syntax is indicated
//in comments

class A {
  //[model: A child of B]
  //[model: A inherits from B]
  [model: A isA B]
}

class B {
}
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//A respects the constraint
//and avoids the error

class A {
  isA B;
  [model: A isA B]
}

class B {
}
```

Load the above code into UmpleOnline

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

//The error is produced, the
//constraint is not respected because
//A is not a superclass of B
//Alternative syntax in comment
class A {
  //[model: A parent of B]
  [model: A superclass B]
}

class B {
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The constraint is now respected
//and the error does not occur
class A {
  [model: A superclass B]
}

class B {
  isA A;
}

```

Load the above code into UmpleOnline

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

```
1.  //The class A generates the error
2.  //by not containing the required
3.  //association
4.  class A {
5.    [model: A -- B]
6.  }
7.
8.  class B {
9.  }
0.
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

view plain

```
1.  //The error is not generated
2.  //A contains an association
3.  //fulfilling the requirements
4.  //of the constraint
5.  class A {
6.    0..1 -- *B;
7.    [model: A -- B]
8.  }
9.
0.  class B {
1.  }
2.
```

Load the above code into UmpleOnline

E095 Duplicate Enumerations

Umple semantic error reported when duplicate enumerations are detected

In Umple, enumerations must have unique names.

Example

```
view plain
1.  // This example causes the error
2.  enum Month {x,y,z}
3.  enum Month {o,p,q}
4.
5.  class A{
6.    Month m;
7.    Month p;
8.  }
9.
```

Load the above code into UmpleOnline

Another Example

```
view plain
1.  // This example causes the error
2.
3.  class A{
4.    enum Month {x,y,z}
5.    enum Month {o,p,q}
6.  }
7.
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
1.  // This example does not cause the error.
2.  // The enumeration within the class
3.  // is prioritized over the enumeration
4.  // defined outside of the class
5.
6.  enum Month {x,y,z}
7.
8.  class A{
9.    enum Month {o,p,q}
10.   Month m;
11.   Month p;
12. }
13.
```

Load the above code into UmpleOnline

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

// This example generates the error
// since the enumeration "X" will be added
// to the class "X" because of attribute "t".

enum X{ Red, Blue, Green }

class X{
  X t;
}

```

Load the above code into UmpleOnline

Another Example

```
view plain

// This generates the error
// because the enumeration "X"
// has the same name
// as the class it's defined in

class X{
  enum X { Red, Blue, Green }
}

```

Load the above code into UmpleOnline

Another Example

```
view plain

// This example generates the error
// since enumeration "Y"
// conflicts with interface "Y"

interface Y {
  name;
}

class X {
 isA Y;
}

```

```
13. enum Y { Red, Blue, Green }
14.
```

Load the above code into UmpleOnline

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 UmpleOnline

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 UmpleOnline

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
// This example generates the error
// because enumeration "Y" has the
// same name as state machine "Y"

class X {
 enum Y { Red, Blue, Green }
 Y {
  s1 {
    goToS2 -> s2;
  }
  s2 { }
 }
}

```

Load the above code into UmpleOnline

Another Example

```
view plain
// This example generates the error
// because enumeration "Y" will be
// added to class "X" since the "attr"
// parameter of "showY" is "Y".

enum Y { Red, Blue, Green }

class X {
  showY(Y attr) {
    System.out.println(attr); }
  Y {
   s1 {}
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
// This example does not generate the error

class X {
```

```
4.  enum Y { Red, Blue, Green }
5.  Z {
6.   s1 {
7.     goToS2 -> s2;
8.   }
9.   s2 { }
0.  }
1. }
2.
```

Load the above code into UmpleOnline

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
1.  // Class with non-alphanumeric name (100)
2.  class @ {}
3.
4.
```

Load the above code into UmpleOnline

Another Example

```
view plain
1.  // Warnings from attribute starting with
2.  // upper case letter (131)
3.  class X {
4.    Attrib;
5.  }
6.
7.  class Y {}
8.
```

Load the above code into UmpleOnline

Another Example

```
view plain
1.  // Attribute name with special characters (130)
2.  class X {
3.    na$me;
4.  }
5.
6.
```

Load the above code into UmpleOnline

Another Example

```
view plain
1.  // Attribute name with special characters
2.  // that looks like an association (132)
3.  // If this is supposed to be an association,
```

```
// put spaces before and after the -- and *
class X {
  1--*Y;
}

class Y {}
```

Load the above code into UmpleOnline

Another Example

```
view plain

// Attribute type with special characters
// that looks like an association (140)
// If this is supposed to be an association,
// put spaces before and after the --
class X {
  1--* Y;
}

class Y {}
```

Load the above code into UmpleOnline

Another Example

```
view plain

// State machine name with special
// characters (150)
class X {
  ed%4 {
    s1 {}
  }
}
```

Load the above code into UmpleOnline

Another Example

```
view plain

// State  name with special characters (152)
class X {
  ed4 {
    s*1 {}
  }
}
```

Load the above code into UmpleOnline

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

// This example generates the warning
// because it is ambiguous if "param"
// is an object of class "Y" or
// the enumeration "Y".

class X {
  enum Y { Red, Blue, Green }
  showY(Y param) {
    System.out.println(param); }
}

class Y { }

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// This example does not generate the warning

class X {
  enum Y { Red, Blue, Green }
  showY(Y param) {
    System.out.println(param); }
}

class Z { }

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// This example does not generate the warning

enum Z { Red, Blue, Green }

class X {
  showY(Y param) {
    System.out.println(param); }
}

class Y { }
```

1.

[Load the above code into UmpleOnline]

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

// This example generates the warning
// because "goToS2" has "param" which could be
// either an object of class "Y"
// or the enumeration "Y"

class X {
  enum Y { Red, Blue, Green }
  sm {
    s1 {
      goToS2(Y param) /{
        System.out.println(param); } -> s2;
    }
    s2 {}
  }
}

class Y { }

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// This example does not generate the warning

class X {
  enum Y { Red, Blue, Green }
  sm {
    s1 {
      goToS2(Y param) /
        { System.out.println(param); } -> s2;
    }
    s2 {}
  }
}

class Z { }

```

Load the above code into UmpleOnline

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

// This example generates the error

class X {
  enum Y { Red, Blue, Green }
  0..1 -- * Y;
}

class Y{ }

```

Load the above code into UmpleOnline

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 }
  0..1 -- * Z;
}

class Z { }

```

Load the above code into UmpleOnline

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
// This example generates the error

class X {
  enum Y { Red, Blue, Green }
  0..1 <@>- * Y;
}

class Y {
  name;
}

```

Load the above code into UmpleOnline

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 }
  0..1 <@>- * Z;
}

class Z {
  name;
}

```

Load the above code into UmpleOnline

W106 Enumeration in Uni-directional Association

Umple semantic warning reported when an enumeration is used in a uni-directional association.

Example

```
view plain
// This example generates the warning

class X {
  enum Y{ Red, Blue, Green }
  0..1 -> * Y;
}

class Y { }

```

Load the above code into UmpleOnline

Another Example

```
view plain
// This example generates the warning

class X {
  enum Y{ Red, Blue, Green }
  0..1 <- * Y;
}

class Y { }

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain
// This example does not generate the warning

class X {
  enum Y { Red, Blue, Green }
  0..1 -> * Z;
}

class Z { }

```

Load the above code into UmpleOnline

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

// The following example shows how to generate
// this error.
interface X {
  const A="dog";
  const A="cat";
}


```

Load the above code into UmpleOnline

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

// The following will generate
// this warning on every line of the class
class X {
  Date d = "20130708";
  Integer i = "abc";
  String s = 123;
  Boolean b = 1;
  Time t = "120000";
}


```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// The following shows how to solve the above problem
class X {
  Date d = "2013-07-08";
  Integer i = 7;
  String s = "123";
  Boolean b = true;
  Time t = "12:00:00";
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// The following shows how to solve
// the above problem not even showing the type at all
class X {
  d = "2013-07-08";
  i = 7;
  s = "123";
  b = true;
```

```
  t = "12:00:00";
  m = 2.3;
}

```

Load the above code into UmpleOnline

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

// This case raises the warning
// because it is probably an error
// Umple will consider "x" to have type "public"
class X {
  public x;
}


```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// This is probably what was intended,
// attributes have private access by default
// but the programmer would call
// public String getX() to read it
// and public boolean setX(String) to set it
class X {
  x;
}


```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// Although not recommended, the programmer
// can also bypass Umple and directly specify
// both the visibility directive and the type.
// Umple will not see that as an attribute
// and will inject the line of code into the
// output directly.
class X {
  public String x;
}


```

Load the above code into UmpleOnline

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

class Person {
   * -> * Person friends;
}

class Student {
   isA Person;
   * -> * Dog friends;
}

class Dog {
}

```

Load the above code into UmpleOnline

Another Example

```
view plain

// This example shows a valid specialization
// of the friends association in which the
// multiplicity 0..3 is refined to *, so a
// student may have any number of dogs but
// Persons in general are limited to 3.
// This conforms to the Liskov substitution
// principle: The preconditions on Person
// operations such as addDog limiting to 3
// are being relaxed in the subclass Student.

class Person {
   * -> 0..3 Dog friends;
}

class Student {
   isA Person;
   * -> * Dog friends;
}

class Dog {
}

```

Load the above code into UmpleOnline

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

//This example shows an error
// in the name of trait
trait T@ {
    //traits elements
}

```

Load the above code into UmpleOnline

Another Example

```
view plain

//This example shows an error
// in the name of trait
trait 1T {
    //traits elements
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//This example shows a valid name for the trait
trait Color {
    //traits elements
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//This example shows a valid name of traits
trait Equality {
    //traits elements
}

```

Load the above code into UmpleOnline

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

// This example shows a warning
// in the name of traits because
// they start with lower case.
trait color {
    //traits elements
}

trait equality{
    //traits elements
}
// @@@skipcompile no code

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//This example shows a valid name for the trait
trait Color {
    //traits elements
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//This example shows a valid name of traits
trait Equality {
    //traits elements
}

```

Load the above code into UmpleOnline

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

//In  this example, trait "T"
// uses trait "T1" which is not available.
interface I{
   //elements
}

class A {
   isA I;
   isA T;
}

trait T{
  isA T1;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//In  this example, the error
// has been resolved just by
// defining trait T1.
interface I{
   //elements
}

class A {
   isA I;
   isA T;
}

trait T{
  isA T1;
}

trait T1{
   //elements
}

```

Load the above code into UmpleOnline

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

// In  this example, Identifier of
// trait "I" is not unique.
class A{
    isA I;
}

class I {
  //elements
}

trait I{
  //elements
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//In  this example, Identifier of
// trait "T" is unique.
class A{
    isA T;
}

class I {
  //elements
}

trait T{
  //elements
}

```

Load the above code into UmpleOnline

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

// In this example, there is an explicit
// use of a trait inside of itself.
class A{
    isA T;
}

trait T{
  isA T;
}

```

Load the above code into UmpleOnline

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

// In this example, there is a cycle
// created in a hierarchy.
class A{
    isA T;
}

trait T{
  isA T1;
}
trait T1{
    isA T;
}

```

Load the above code into UmpleOnline

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

// In this example, there is an error
// because class C1 does not implement
// interface I. Any bound value to template
// parameter TP must implement interface I.
trait T1<TP isA I>{
  /*implementation*/
}
interface I{/*implementation*/}
class C1{/*implementation*/}
class C{
  isA T1< TP = C1 >;
  /*implementation*/
}

```

Load the above code into UmpleOnline

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

// In this example, the required method
// "FinalResult()" in Trait "T" is
// not satisfied by class "A".
class A{
    isA T;
}

trait T{
    Integer FinalResult();
    void calculate(){
        //method FinalResult() will be used here;
    }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// In this example, the require method
// "FinalResult()" in Trait "T" is
// satisfied by class "A".
class A{
    isA T;
    internal Integer x;
    Integer FinalResult(){
        return x/100;
    }
}

trait T{
    Integer FinalResult();
    void calculate(){
        //method FinalResult() will be used here;
    }
}

```

Load the above code into UmpleOnline

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

// In this example, there is a conflict
// in class "A" on the method "test()".
class A{
    isA T1;
    isA T2;
}
trait T1 {
    void test(){
    //special implementation by T1
    }
}
trait T2 {
    void test(){
    //special implementation by T2
    }
}

```

Load the above code into UmpleOnline

Another Example

```
view plain

// In this example, there is
// a conflict in trait "T"
// on the method "test()".
class A{
    isA T;
}
trait T{
    isA T1;
    isA T2;
}
trait T1{
    void test(){
    //special implementation by T1
```

```
    }
}
trait T2{
    void test(){
    //special implementation by T2
    }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

view plain

```
// In this example, there is a
// no conflict in class "A".
class A{
    isA T1;
    isA T2;
}
trait T1 {
    isA M;
}
trait T2 {
    isA M;
}
trait M{
    void test(){
    //special implementation by M
    }
}

```

Load the above code into UmpleOnline

Another Example

view plain

```
// In this example, there is a conflict
// in class "A" because method "test"
// is overridden in trait "T2".
class A{
    isA T1;
    isA T2;
}
trait T1 {
    isA M;
}
trait T2 {
    isA M;
    void test(){
    //special implementation by T2
    }
}
trait M{
```

```
18.     void test(){
19.     //special implementation by M
20.     }
21. }
22.
```

Load the above code into UmpleOnline

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

// In this example, there are
// two modifications ("add") for
// the method "show()" when trait "T2"
// is used inside of trait "T1".
class A{
   isA T1;
}
trait T1{
   isA T2 <+show(),+show()>;
}
trait T2{
   void show(){
   //implementation
   }
}

```

Load the above code into UmpleOnline

Another Example

```
view plain

// In this example, there are
// two modifications ("add" and "remove")
// for the method "show()" when trait "T2"
// is used inside of trait "T1".
class A{
   isA T1;
}
trait T1{
   isA T2 <-show(),+show()>;
}
trait T2{
   void show(){
   //implementation
   }
}

```

Load the above code into UmpleOnline

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

// In this example, there is an error
// because trait "T1" tries to remove
// method "show()" from trait "T2"
// while it is not available
// in trait "T2".
class A{
    isA T1;
}
trait T1{
    isA T2 <-show()>;
}
trait T2{
}

```

Load the above code into UmpleOnline

Another Example

```
view plain

// In this example, there is an error
// because trait "T1" tries to add
// just method "show()" from trait "T2"
// to itself. while it is not
// available in trait "T2".
class A{
    isA T1;
}
trait T1{
    isA T2 <+show()>;
}
trait T2{

}

```

Load the above code into UmpleOnline

Another Example

```
view plain

// In this example, there is an error
```

```
// in class "A" because it tries to
// change visibility of method "test()"
// which is not available in trait T.
// Consider that the method "test()"
// defined in trait "T" is a
// required method.
class A{
    isA T<test() as private>;
}
trait T{
    void test();
}

```

Load the above code into UmpleOnline

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

// In this example, there is an error
// because a bidirectional association
// is created with interface I,
// which is not valid.
interface I {
  /*implementation*/
}
trait T <RelatedClass> {
  0..1 -- * RelatedClass;
  /*implementation*/
}
class C{
 isA T<RelatedClass=I>;
 /*implementation*/
}

```

Load the above code into UmpleOnline

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

```
// In this example, there is an error
// because trait T has two template
// parameters with the same name.
trait T<TP,TP>{
  TP name;
}
class C1{
  isA T;
}
```

Load the above code into UmpleOnline

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

// In this example, there is an error
// because class A cannot bind
// type String to template parameter Y,
// which is not defined. for trait T.
class A{
  isA T< X= Integer, Y = String>;
}
trait T<X>{
    X variable;
}

```

Load the above code into UmpleOnline

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 UmpleOnline

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 UmpleOnline

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

// In this example, there is a warning
// because in class "A" there will be
// two attributes with the name called "name".

class A{
   isA T;
   name;
}
trait T{
   name;
}

```

Load the above code into UmpleOnline

Another Example

```
view plain

// In this example, there is a
// warning because in trait "T"
// there will be two attributes
// with the name called "name".
class A{
   isA T;
}
trait T{
   isA T1;
   name;
}
trait T1{
   name;
}

```

Load the above code into UmpleOnline

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

// In this example, there is an error
// because class "A" doesn't bind
// a value to one of the template
// parameters of trait "T" named "Y".
class A{
    isA T<X = B>;
}
class B{
    //elements
}
trait T<X,Y>{
    //elements
}

```

Load the above code into UmpleOnline

Another Example

```
view plain

// In this example, there is an error
// because class "A" doesn't bind
// a value to template parameters
// of trait "T".
class A{
    isA T;
}
class B{
    //elements
}
trait T<X,Y>{
    //elements
}

```

Load the above code into UmpleOnline

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

// In this example, there is an error
// because in class "A" there will be
// two methods with the name "ShowInConsole".
class A{
   isA T<show() as ShowInConsole>;
}
trait T{
   isA T1;
   void show(){/*T*/}
}
trait T1{
   void ShowInConsole(){/*T1*/}
}

```

Load the above code into UmpleOnline

Another Example

```
view plain

// In this example, there is an error
// because in class "A" there will be
// two methods with the name "ShowInConsole".
class A{
   isA T<show() as ShowInConsole>;
   isA T1;
}
trait T{
   void show(){/*T*/}
}
trait T1{
   void ShowInConsole(){/*T1*/}
}

```

Load the above code into UmpleOnline

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 UmpleOnline

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

// In this example, there is an error
// because class C binds type C1 to the
// template parameter of trait T,
// while type C is not available
// in the system.
trait T<TP>{
/*implementation*/
}
class C{
  isA T<TP=C1>;
  /*implementation*/
}

```

Load the above code into UmpleOnline

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 UmpleOnline

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

// In this example, there is an error
// because type I used to constraint
// template parameters TP in trait T
// does not exist.
trait T<TP isA I>{
  /*implementation*/
}
class C1{
  /*implementation*/
}
class C1{
  isA T< TP = C1 >;
  /*implementation*/
}

```

Load the above code into UmpleOnline

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

```
// In this example, there is an error
// because trait T puts multiple
// inheritance constraint on its
// template parameter TP.
trait T<TP isA C1&C2>{
  /*implementation*/
}
class C1{
  /*implementation*/
}
class C2{
  /*implementation*/
}
```

Load the above code into UmpleOnline

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

// In this example, there is an error
// because class C1 is not subclass
// of class C2.
// Any bound value to template parameter
// TP must be a subclass of class C2.
trait T<TP isA C2>{
  /*implementation*/
}
class C{
  isA T< TP = C1 >;
  /*implementation*/
}
class C1{/*implementation*/}
class C2{/*implementation*/}

```

Load the above code into UmpleOnline

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

```
// In this example, there is an warning
// because two traits T1 and T2 have
// state machines with the same name but
// with different initial states.
trait T1{
  sm{
    s1{ e1 ->  s2;}
    s2{ e3 ->  s3;}
    s3{ e2 ->  s2;}
  }
}
trait T2{
  sm{
    t1{ t1 ->  t2;}
    t2{ t3 ->  s3;}
    s3{ t2 ->  t2;}
  }
}
class C1{
  isA T1,T2;
}

```

Load the above code into UmpleOnline

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

// In this example, there is an error
// because class C tries to
// rename states s0 of trait T two times.
trait T {
  sm{
    s0{
      e1-> s1;
      s11{ e12-> s12; }
      s12{ e11-> s11; }
    }
    s1{ e0-> s1;  }
  }
}
class C {
  isA T<sm.s0 as state0, sm.s0 as state01>;
}

```

Load the above code into UmpleOnline

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

```
// In this example, there is an error
// because state machine sm3 is
// not available in trait T.
trait T {
  sm1{
    s0 {e1-> s1;}
    s1 {e0-> s0;}
  }
  sm2{
    s0 {e1-> s1;}
    s1 {e0-> s0;}
  }
}
class C3 {
 isA T<sm3 as mach1>;
}

```

Load the above code into UmpleOnline

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

// In this example, there is an error
// because event e2() is not available
// in state machine sm.
trait T {
  sm{
    s0{
      e1-> s1;
      s11{ e12-> s12; }
      s12{ e11-> s11; }
    }
    s1{ e0-> s1;  }
  }
}
class C {
  isA T<sm.e2() as event2>;
}

```

Load the above code into UmpleOnline

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

```
// In this example, there is an error
// because event e2() is not available
// in state machines of trait T.
trait T {
  sm{
    s0{
      e1-> s1;
      s11{ e12-> s12; }
      s12{ e11-> s11; }
    }
    s1{ e0-> s1;  }
  }
}
class C {
  isA T<*.e2() as event2>;
}

```

Load the above code into UmpleOnline

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

// In this example, there is an error
// because the initial state of a
// state machine cannot be removed.
trait T {
  sm{
    s0{
      e1-> s1;
      s11{ e12-> s12; }
      s12{ e11-> s11; }
    }
    s1{ e0-> s1;  }
  }
}
class C {
  isA T<-sm.s0>;
}

```

Load the above code into UmpleOnline

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 UmpleOnline

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

// In this example, there is an error
// because the keyword superCall can be
// pointed out to two used traits.
trait T1{
  sm{
    s1{ e1 -> /{action1();} s1; }
  }
}
trait T2{
  sm{
    s1{ e1 -> /{action2();} s2;}
    s2{}
  }
}
class C1{
  isA T1,T2;
  sm{
    s1{ e1 -> /{superCall; action3();} s3; }
    s3{}
  }
}

```

Load the above code into UmpleOnline

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

```
// In this example, there is an error
// because of having different ways
// to compose entries of two states
// S1 coming from traits T1 and T2.
trait T1{
  sm{
    s1{ entry /{action1();} }
  }
}
trait T2{
  sm{
    s1{ entry /{action2();} }
  }
}
class C1{
  isA T1, T2;
  sm{
    s1{}
  }
}

```

Load the above code into UmpleOnline

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

```
// In this example, there is an error
// because class C rename the region
// name r1 to r2 which is already
// available in the state machine sm.
trait T{
  sm {
    s1{
      r1{ e1-> r11; }
      r11{}
      ||
      r2{ e2-> r21; }
      r21{}
    }
  }
}
class C{
  isA T<sm.s1.r1 as r2>;
}

```

Load the above code into UmpleOnline

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

//The trace directive causes the
//warning to be issued as the attribute
//id is not found
class A {
  Integer identifier;

  trace id;
}
// @@@skipphpcompile See issue 596 (PHP tracing causes issues)

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The warning is resolved
//by correcting the name of the
//attribute to be traced
class A {
  Integer identifier;

  trace identifier;
}
// @@@skipphpcompile See issue 596 (PHP tracing causes issues)

```

Load the above code into UmpleOnline

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

//The tracer used is not recognized,
//the Console tracer will be used
tracer OtherTracer;

class A {
  Integer id;

  trace id;
}
// @@@skipphpcompile See issue 596 (PHP tracing causes issues)

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//Using a supported tracer avoids
//the warning
tracer File;

class A {
  Integer id;

  trace id;
}
// @@@skipphpcompile See issue 596 (PHP tracing causes issues)

```

Load the above code into UmpleOnline

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

// This causes the warning
interface Y {
  constant A=5;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// This does not cause the warning
interface Y {
  const A=5;
}

```

Load the above code into UmpleOnline

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

```
/*
Example to show how to define inline mixsets.
*/

// The mixset M1 has been defined as inline mixset. this because the definition of the n
mixset M1 class A {
  isA B;
  isA C;
  x;
}

// M1 also is inline mixset because it has been followed by a trait definition without cur
// The usuall definition of M1 is:
/*
mixset M1
{
  trait B
  {
    v;
  }
}
*/
mixset M1 trait B
{
  v;
}


// M2 is an inline mixset that defines an interface.
mixset M1 interface C { void method(); }


class D {
  // M3 adds an attribute for class D. It's an inline mixset since there is no mention to v
  // However, Umple parser understands this implicit definition. Therefore, the attr1 wil
  mixset M3 {attr1;}

```

```
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 UmpleOnline

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

//The name of the template is
//not valid and generates
//an error
class A {
  !template <<!output!>>
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The name of the template
//is valid in the following
class A {
  template <<!output!>>
}

```

Load the above code into UmpleOnline

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

//The emit method name
//cannot be main, the error
//is produced
class A {
  template <<!output!>>
  emit main()(template);
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//Using another method name
//solves the error
class A {
  template <<!output!>>
  emit method()(template);
}

```

Load the above code into UmpleOnline

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

//The template name cannot be
//found as it does not exist
class A {
  emit method()(template);
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The error is resolved by
//declaring the template
class A {
  template <<!output!>>
  emit method()(template);
}

```

Load the above code into UmpleOnline

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

//Template otherTemp refers to
//itself causing the error
class A {
  temp <<! output !>>
  otherTemp <<! <<@otherTemp>> !>>
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//No template refers to itself,
//the error does not occur
class A {
  temp <<! output !>>
  otherTemp <<! <<@temp>> !>>
}

```

Load the above code into UmpleOnline

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

//The template otherTemplate
//is not found, generating the
//error
class A {
  temp <<! <<@otherTemplate>> !>>
  emit method()(temp);
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//otherTemplate exists and can
//be referenced
class A {
  otherTemplate <<! output !>>
  temp <<! <<@otherTemplate>> !>>
  emit method()(temp);
}

```

Load the above code into UmpleOnline

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

//The error arises as temp refers
//to temp3, referring to temp2, which
//causes a cycle due to temp2 referring
//to temp
class A {
  temp <<! <<@temp3>> !>>
  temp2 <<! <<@temp>> !>>
  temp3 <<! <<@temp2>> !>>

  emit method()(temp);
}
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//There is no longer a cycle,
//the error is resolved
class A {
  temp <<! <<@temp3>> !>>
  temp2 <<! <<@temp>> !>>
  temp3 <<! output !>>

  emit method()(temp);
}
```

Load the above code into UmpleOnline

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

//Class A contains two templates
//of the same name, causing
//the warning
class A {
  temp <<! output !>>
  temp <<! otherOutput !>>
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The warning is resolved by
//giving the second template
//a distinct name
class A {
  temp <<! output !>>
  secondTemp <<! otherOutput !>>
}

```

Load the above code into UmpleOnline

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

//Class A contains two emit
//methods of the same name
class A {
  temp <<! output !>>
  otherTemp <<! output !>>

  emit aMethod()(temp);
  emit aMethod()(otherTemp);
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//Changing the name of the
//second emit method resolves
//the issue
class A {
  temp <<! output !>>
  otherTemp <<! output !>>

  emit aMethod()(temp);
  emit aMethod2()(otherTemp);
}

```

Load the above code into UmpleOnline

E3607 Port Name Cannot Be Resolved

Umple semantic error raised when a port name cannot be resolved

A port used in a class must be defined before usage by port bindings and active methods. The cause of the error is possibly a typographical mistake.

Example

```
view plain

//inB has not been defined
//in class A, causing the error
class A {
  public in Integer inA;
  public out Integer outA;

  inB -> outA;
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//Both inA and outA are
//defined; the error does
//not occur
class A {
  public in Integer inA;
  public out Integer outA;

  inA -> outA;
}

```

Load the above code into UmpleOnline

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

//The tags are not properly
//nested, causing E4500
<FIXML>
  <tag>
    <othertag>
      </tag>
  </othertag>
 </FIXML>

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//Each tag is now correctly
//closed
<FIXML>
  <tag>
    <othertag>
    </othertag>
  </tag>
 </FIXML>

```

Load the above code into UmpleOnline

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");
07.   }
08. }
09.
```

Load the above code into UmpleOnline

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, 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. Make sure the curly brackets match; make sure there are semicolons after transitions (unless the transitions have an action). If you are still stuck, comment out segments until you can narrow down the problem.

Example

view plain

```
1.  // This example generates the warning
2.  // because there is a missing semicolon
3.  class X {
4.    sm {
5.      a -> b
6.      b -> c
7.    }
8.  }
9.
0.
```

Load the above code into UmpleOnline

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

// The following will generate two such warnings
class X {
  blah blah blah;
  1 -* X;
}
```

Load the above code into UmpleOnline

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 UmpleOnline

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

//The getAttr method is already
//generated by the declaration of
//the attribute attr and generates
//the warning
class A {
  attr;
  getAttr() {/* Implementation */};
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//Modifying the name of the
//method can fix the warning
class A {
  attr;
  returnAttr() {/* Implementation */};
}

```

Load the above code into UmpleOnline

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

//The warning is raised twice in
//the following as both constraints
//are not properly constructed
class A {
  Integer a;
  Integer b;
  Integer c;

  [(a > b)) && a < c]
  [c < a | b > a]
}
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The warning is not generated
//by fixing the constraint syntax
class A {
  Integer a;
  Integer b;
  Integer c;

  [(a > b) && a < c]
  [c < a || b > a]
}
```

Load the above code into UmpleOnline

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

// This example generates the warning
// because one participating class is
// missing from the key
class Passenger {}

class Flight {}

associationClass Booking {
  Integer number;
  * Passenger passenger;
  * Flight flight;
  key {number, flight}
}
```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

// This example shows how to remove
// the warning by completing the key
class Passenger {}

class Flight {}

associationClass Booking {
  Integer number;
  * Passenger passenger;
  * Flight flight;
  key {number, flight, passenger}
}
```

Load the above code into UmpleOnline

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

//getAttr does not exist in the class
//which generates the warning
class A {
  before getAttr {
    doSomething();
  }

  doSomething(){/* Implementation */}
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//By declaring the attribute attr
//the getAttr method can now be used
class A {
  attr;

  before getAttr {
    doSomething();
  }

  doSomething(){/* Implementation */}
}

```

Load the above code into UmpleOnline

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

//The before statement refers to
//the getAttr method specifying no
//parameters, causing the warning
class A {
  attr;

  before getAttr() {
  }
}

```

Load the above code into UmpleOnline

Solution to The Above So the Message No Longer Appears

```
view plain

//The method is referred to without
//parameters in the following, avoiding
//the warning
class A {
  attr;

  before getAttr {
  }
}

```

Load the above code into UmpleOnline

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 UmpleOnline

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 UmpleOnline

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.

E1510 Use File Missing

Umple error messages 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.

Check the spelling of the filename, and check file permissions.

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.

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

```
// This demonstrates code that will throw the
// immutability exception
class Point2D
{
  immutable;
  Float x;
  Float y;
}

class Path
{
  immutable;
  1 -> * Point2D pathElements;
    public static void main(String[] argv) {

      Point2D p2d = new Point2D(0,0);


      // This will throw an exception since the
      // pathElements cannot be duplicate
      try {
        Path p = new Path(p2d,p2d);
      }
      catch (RuntimeException re) {
        System.out.println("Exception Caught: "+re+"\n");
      }
    }
}
```

Load the above code into UmpleOnline

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
17.     // be at least one program
18.     try {
19.       s1 = new Student("Abel", null);
20.     }
21.     catch (RuntimeException re) {
22.       System.out.println("Exception Caught: "+re+"\n");
23.     }
24.
25.     // After we create a program we can repeat the
26.     // attempt.
27.     p1 = new Program("Computer Science");
28.     s1 = new Student("Abel", p1);
29.
30.     // Attempting to change the program back to null
31.     // 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 UmpleOnline

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) {

    Company c1 = null;
    StockSymbol s1 = null;

    // This will throw an exception since the stock
    // symbol cannot be null
    try {
      c1 = new Company("Umpleco", s1);
    }
    catch (RuntimeException re) {
      System.out.println("Exception Caught: "+re+"\n");
    }

    // This will succeed: Umple 1--1 constructors
    // create both objects at once
    c1 = new Company("Umpleco","UMPC");

    s1 = c1.getStockSymbol();
    System.out.println("The company: "+
      c1+"\n"+
      "The stock symbol: "+
      s1+"\n");
  }
}
```

Load the above code into UmpleOnline

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

```
// This demonstrates code that will throw the
// uniqueness exception
class Student {
  unique Integer number;
  name;

  public static void main(String[] argv) {

    Student s1 = null;
    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");
  }
}
```

Load the above code into UmpleOnline

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
class Student {
   id = "ID123";
   name = "John";


   test checkSetId {
   Student s ();

   assertTrue(s1.setId("ID432"));

   }
}

```

Load the above code into UmpleOnline

Another Example

```
view plain
class Student {
   id;
   name;

```

```
    test checkSetId {
    Student s ("98CS","Jane");

    assertEqual(s.getId(),"98CS");

    s.setName("Roger");
    assertFalse(s.getName() == "Jane");

    }
}

```

Load the above code into UmpleOnline

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

```
class Advisor
{
  name;
  id;
  int tel;

  0..1 -- * Student;

  1 -- * Office;

  2 -- * Course;

}


class Student {
  name;

}


class Office {
  name;
  lazy address;



}


class Course {
  code;
  int numberOfStudents;
  description;

  Date startDate = "2020-12-26";
```

```
37.
38.    Time classTime = "12:59:59";
39.
40.
41.  }
42.
```

Load the above code into UmpleOnline

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.   iob;
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.
```

```
2.  use Job;
3.  use JobTest;
4.
5.
6.
```

Load the above code into UmpleOnline

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
class Student {

   String studentId;
   nameMemebershipId;
   address;
   studentAge;


 generic test checkifLogged(String attribute.regex(\w*Age)){
      Student st1 ( "S1425", "John", "Ottawa",20) ;
      String valueToCheck = st1.get<<attribute>>();

      assertTrue(valueToCheck > 18);


      }

}
```

Load the above code into UmpleOnline

Another Example

```
view plain
class Calculator{


      1--* Number;

      0..1->* Controller;


 generic test checkifLogged(0..1->* association){
      Calculator c1 () ;
      Number n1(c1);
      Number n2(c1);
      Controller cn1();
```

```
        Controller cn2();

        c1.addNumber(n1);
        c1.addNumber(n2);

        c1.addController(cn1);
        c1.addController(cn2);

        String numberInList = c1.numberOf<<association.toSingular()>>s();

        if(c1.numberOfNumbers() > 2)
        {
           assertTrue (numberInList > 2);
        }

        else {
           assertTrue (numberInList < 2)
        }

        }

}


class Number {


}

class Controller {


}
```

Load the above code into UmpleOnline

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){
```

```
13.         Calculator c1 ( 4, 5) ;
14.         String valueToCheck = p1.get<<method>>();
15.         ps1.getValue(<<method>>);
16.         boolean isLogged =  p1.checkIsLogged(valueToCheck);
17.         assertTrue(logged == "true");
18.         }
19.    generic test checkifLogged(String method Integer){
20.         // code omitted
21.      }
22. }
23.
24.
25.
```

Load the above code into UmpleOnline