Quality Engineering Activities in Object-Oriented Software Development

by

Cecilia Geldrez

A thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree

Master of Computer Science

Ottawa-Carleton Institute of Computer Science
Department of Computer Science
Faculty of Computer Science
University of Ottawa

December 1995

ISBN 0-612-15622-2

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Abstract

Testability is a measure of how easy (less complex, less tedious, less boring, less costly) the effective testing of implementations is made.

In Object-Oriented development, testability is a result of (1) the characteristics of Object-Oriented implementations, (2) the test support environment, (3) the characteristics of representations (i.e. notations) and (4) the software process in which testing is conducted [67]. In this thesis, we examine in details the last two factors.

When addressing the characteristics of notations, we determine that notations should exhibit the basic *testability attributes* of completeness, correctness, consistency and incorporation. In order for notations to exhibit these attributes, they must be verifiable, modifiable and traceable. The latter three are referred to as *testability requirements*.

A notation that is *verifiable* can be tested for *testability attributes*; a notation that is *modifiable* can accommodate changes to achieve *testability attributes*; finally a notation that is *traceable* can be traced to other notations to achieve *testability attributes*.

The rational for using notations that exhibit *testability requirements* is that these notations can be assessed for *testability attributes* via testing activity types. This provides the ability of (1) uncovering errors early in the process where they are less costly to fix, (2) generating test cases that can be applied to the implementation, (3) guiding the testing of the implementation and (4) facilitating the location of source(s) of error(s) for modification.

We identify testing activity types that are decoupled from the syntax of notations and from development methodologies. These testing activity types define what needs to be tested (e.g. different semantics) as well as coverage criteria. They are generic in the sense that they are refinable (i.e. adaptable to different levels of detail) in order to be uniformly applied at all phases of development.

When addressing the software process in which testing is conducted, we propose that there must be an integration of development and testing activity types. Also, we propose a way of handling change in a consistent manner such that testability attributes are not lost. Examples of these changes include error correction, accommodation of new requirements, etc.

Acknowledgment

First, I would like to thank my supervisor Dr. Jean-Pierre Corriveau for his time spent in my thesis. He spent long hours reading, correcting and providing me with technical corrections and comments. His guidance, moral support, availability and good humor were the key for my submission and defense. Again, Dr. Corriveau, I absolutely could not have done it without you!!

I also would like to thank Dr. Robert L. Probert for his financial support and contributions in the area of validation and verification in software development.

I would like to acknowledge Bell-Northern Research for providing me with a software development environment where the inspiration for this work originated.

Last but not least, I would like to acknowledge my family for their support throughout this thesis. My most special thanks go to my parents for their encouragement. muchas gracias!

List of Figures

List of Tables

List of Acronyms

CASE	Computer Aided Software Engineering
DOORS	Dynamic Object-Oriented Requirements System
GIOOTDM	Generic Integrated Object-Oriented Testing and Development Methodology
GOODM	Generic Object-Oriented Development Methodology
OMT	Object-Modeling Tool
OOAD	Object-Oriented Analysis and Design
OO	Object-Oriented
RTM	Requirements Tracing and Management
SDL	System Description Language
SDT	System Description Language Design Tool

Chapter 1 Introduction

1.1 Introduction

Since the 1980's, the Object-Oriented (OO) paradigm has received great attention in industry and academia. The reason for this attention is that OO methodology aims at producing easier to model and to understand, extendible, reusable and reliable software. These claims are becoming facts as new applications of the paradigm emerged and substantial improves in productivity are reported [1].

Until recently, as stated in [2], relatively little attention has been paid to OO testing. As shown in [3,4,5,6] among others, most of this attention has been paid to OO implementation testing (i.e. testing that involves code). Very little emphasis has been placed on testing OO notations. The latter aims at ensuring that notations are complete, correct, consistent and incorporated in themselves and with respects to other notations.

The advantage of finding high level errors before coding is well recognized in software engineering [7]. The testing of implementations is facilitated by moving it upwards in the software life-cycle, thus allowing to find high-level errors early in the development process where they are less costly to fix. Therefore, the *testability* of implementations (i.e. a measure of how easy the effective testing of the implementation is made) increases.

As discussed in [7], testing activities involved at Requirements, Analysis and Design phases lead to an integration of the development process with the testing process. In OO software this integration is discussed in [8]. This integration is crucial to the quality of OO software for mainly two reasons.

First, testing OO software differs from testing Procedural software due to encapsulation, inheritance, polymorphism, the exponential interaction among objects and the lack of a one-to-one mapping between a function and a unit of software [9]. These added complexities make the testing phase of OO software more challenging than in Procedural software. Testing activities applied early in the OO software development process uncover high-level errors and allow for the creation of test suites (i.e. test cases and expected outputs) that can be applied to the implementation.

Second, OO systems tend to evolve using the Spiral Model [10]. This methodology allows an *iterative incremental* approach in which the phases of the development process (i.e. Requirements, Analysis, Design, Implementation and Testing) are repeated (*iterative*) and the system is developed through successive refinements increasing in level of detail (*incremental*). In such a methodology early validation and verification techniques become very important. If errors are not detected during the testing phase of a given iteration of the Spiral, they will be propagated to the next iteration. The cost of such propagation errors can be tremendous. This is because software engineers will be building their systems on top of a version with errors, therefore creating new errors. This ripple effect will have a great impact in the quality of the system.

The solution is to increase the probability of detecting errors. This can be achieved by (1) introducing testing activities at Requirements, Analysis and Design phases and not confining testing to an after implementation activity and (2) ensuring that Implementations are indeed derived from the previously tested Designs. The focus of the work presented in this document is the first point (i.e. introducing testing activities at Analysis and Design).

1.2 Motivation

The importance of introducing testing activities early in the Object-Oriented Software Development Methodology was described in the previous section: they aim at increasing the testability of OO implementations and therefore increase Object-Oriented Software quality.

While the details (i.e. when to test a component, what to test and how to test) of OO implementations have been addressed, as in [3], the details of testing notations used in the OO paradigm still remain a topic of research. The major motivation for the work presented, is therefore the need for a concrete methodology for performing testing activities early in the OO software development process.

We believe this concrete methodology should include three aspects. First, a *process* describing when testing activities should occur. This largely corresponds to the "when". Second, the *testing activities* that should take place in the process. This corresponds to the "what". Third, the *methods* that implement the different testing activities. The latter corresponds to the "how".

Motivation in the *process* area: **no way of handling change**

In the area of *process*, a testing phase should follow every development phase, thus leading to "analyze a little, test a little, design a little, test a little, code a little, test a little and analyze a little, ..." [8]. The Integrated Object-Oriented Testing and Development Process in [8], describes such an ordering of phases. However, it is missing a crucial aspect: it does not provide a way of handling change in a consistent manner. Examples of these changes include error correction, accommodation of new requirements, etc. The usual tendency in software engineering is to accommodated changes at the phase where they are discovered. For example, if a change to Requirements notations is identified at Design, the usual tendency is to modify the Design, perhaps update the Requirements and then continue development from Design on. Even though some methodologies support this type of "*spaghetti development* (Design --> Requirements --> Design)", such as in [22], it results in a loss of traceability of notations within a given iteration and from one iteration to the next. Once traceability is lost, it is no longer possible to determine if a notation is derived from or traceable to another tested notation, thus questioning the effort spend testing the latter notation, since the errors corrected can reoccur in the derived notation. A Grey-Box analysis technique, "semantic instrumentation" [7] offers a means of enforcing consistent change handling for traditional software design, but has not been extended to the OO paradigm.

Motivation in the *testing activities* area: **not generic and ad-hoc**

Testing activities are currently addressed in an ad-hoc manner (i.e. no definition of what is being tested and why) and are notation and methodology specific (e.g. "Every Use Case should be traceable through the relationships illustrated in the Object Model" [8]). This causes a proliferation of testing activities (i.e. one for each notation).

What is missing is a generic set of testing activities *(testing activity types)* decoupled from development methodologies and syntax of notations. When designing OO software, we observe that at each development phase, the system to be developed is considered from two angles: (1) the level of abstraction and (2) the View or semantics provided. It is the merge of these two angles that needs to be carefully examined in order to arrive to *testing activity types*.

While examining this merge, we observe that different notations can be used to describe a specific level of abstraction and that within that level of abstraction, these different notations can be grouped based on the semantics they provide. This classification is called a Model. A Model is thus a classification of notations based on the level of abstraction and semantics they provide (e.g. notations such as SDLs [12] and State Charts [13], can be classified as belonging

to the State Machine Model as illustrated in Fig. 1.2). Furthermore Models can themselves be grouped into Views by considering only the semantics they provide (e.g. the State Machine Model provides the Dynamic View, as illustrated in Fig. 1.2).

Testing activity types should be generic to a group of notations (i.e. View). They should (1) define what needs to be tested, (2) define coverage criteria, and (3) assess for completeness, correctness, consistency and incorporation of notations within that View and with respect to other Views. Thus, a notation can be tested by applying *testing activity types* to the View the notation provides.

Testing activity types should be generic in the sense that they should be refinable (adaptable to different levels of details) to be uniformly applied at all phases of development. The latter requirement arises from the observation that in OO software development, uniform principles of object-orientation are applied throughout the development process: thus one can talk of OO Design being a refinement of OO Analysis and OO Implementation being a refinement of OO Design.

Motivation in the *method* area: **not generic and based in code generation**

A *method* corresponds to the implementation of a *testing activity*. By implementation, we understand: the test case design for the *testing activity* (e.g. use of Black, Grey and White Box testing techniques [7, 59]); how the *testing activity* will be executed (e.g. how to observe and control the notations being tested); the generation of test case reports, etc. Note that by "implementation", we do not necessarily mean machine executable implementation.

As in the case of *testing activities*, *methods* are notation specific. This results in their proliferation. The motivation is to arrive to generic implementations *(method types)* independent of notation. A *method type* corresponds to the implementation of a *testing activity type* . For example, Deadlock Detection[11] can be considered a *testing activity type* and Reachability Analysis [11] a *method type* (please refer to Fig. 1.2).

Also, there is a lack of *methods* that test notations without relying on code generation. Most tools that claim to test notations, actually test the code generated from notations (e.g. Argos [55]).

1.3 Contribution

Our contribution is in providing a concrete methodology (i.e. process, testing activities and methods) for testing early in the Object-Oriented Software Development Methodology, to increase the testability of OO software.

Fig. 1.1 illustrates the contribution of our work. The terminology in Fig. 1.1 is discussed in details in Chapters 4; we provide a brief overview at this point to clarify the discussion presented in this chapter. Fig. 1.1 illustrates the class configuration of our methodology (the notation used is Object-Modeling Tool (OMT) [14,39]). Fig. 1.2 illustrates an example instance (object) configuration of the class configuration in Fig. 1.1.

In the area of *process*, we propose the `Generic Integrated Object-Oriented Testing and Development Methodology` (GIOOTDM). We define the methodology as a composition of a `Process`, a set of `Development Phases` (e.g. Analysis, Design, etc.) and `Testing Phases` (e.g. Validation, Verification, etc.). Our `Process` defines these `Phases`. This definition includes the type of phase (e.g. Analysis, Design, Validation, Verification, etc.), the ordering of the phases (e.g. Analysis followed by Verification followed by Design) and specific points in software development where change (i.e. error correction, enhancements, new requirements, etc.) should take place.

To arrive to *testing activity types*, we first classify `Notations` into `Models` and `Models` into `Views`. A `Model` describes a semantic `View` of a system as well as a specific level of abstraction. Every `Model` (e.g. State Machine Model) can in turn be represented by different `Notations` (e.g. State Charts, SDLs). One or more `Models` illustrate a `View`. Therefore, `Notations` illustrate the `Views` of a system as shown by the derived association in italic. Having identified this classification, we can define `Testing Activity Types` based on `Views`. A `Testing Activity Type` tests one or more `Views`. They indirectly test `Models` and `Notations` as illustrated by the derived association in italics. The definition of a `Testing Activity Type` includes what it should test, the coverage criteria for Notations within that View and assessment of testability attributes (i.e. completeness, correctness, consistency and incorporation of Notations).

In order to verify that our proposed *testing activity types* are applicable and scalable, we provide examples in Chapters 6 and 7.

The area of *methods* is not addressed by our work; however testing activity types provide the basic building blocks for creating *method types*.

Finally, we emphasize that our proposed manner of dealing with change provides a framework for metric collection (e.g. number of errors, number of iteration involved in fixing an error, etc.). Also, we suggest that *testing activity types* provide concrete guidelines or a checklist for reviewing notations. Current and new notations should be accompanied with both the View they describe (therefore, with the *testing activity types* that they will exercise) as well as the *method type* used to achieve the *testing activity types*. This provides a metric for how testable a notation is. To conclude the work, we describe a high level proposal for tool development explaining how testing activity types can be used.

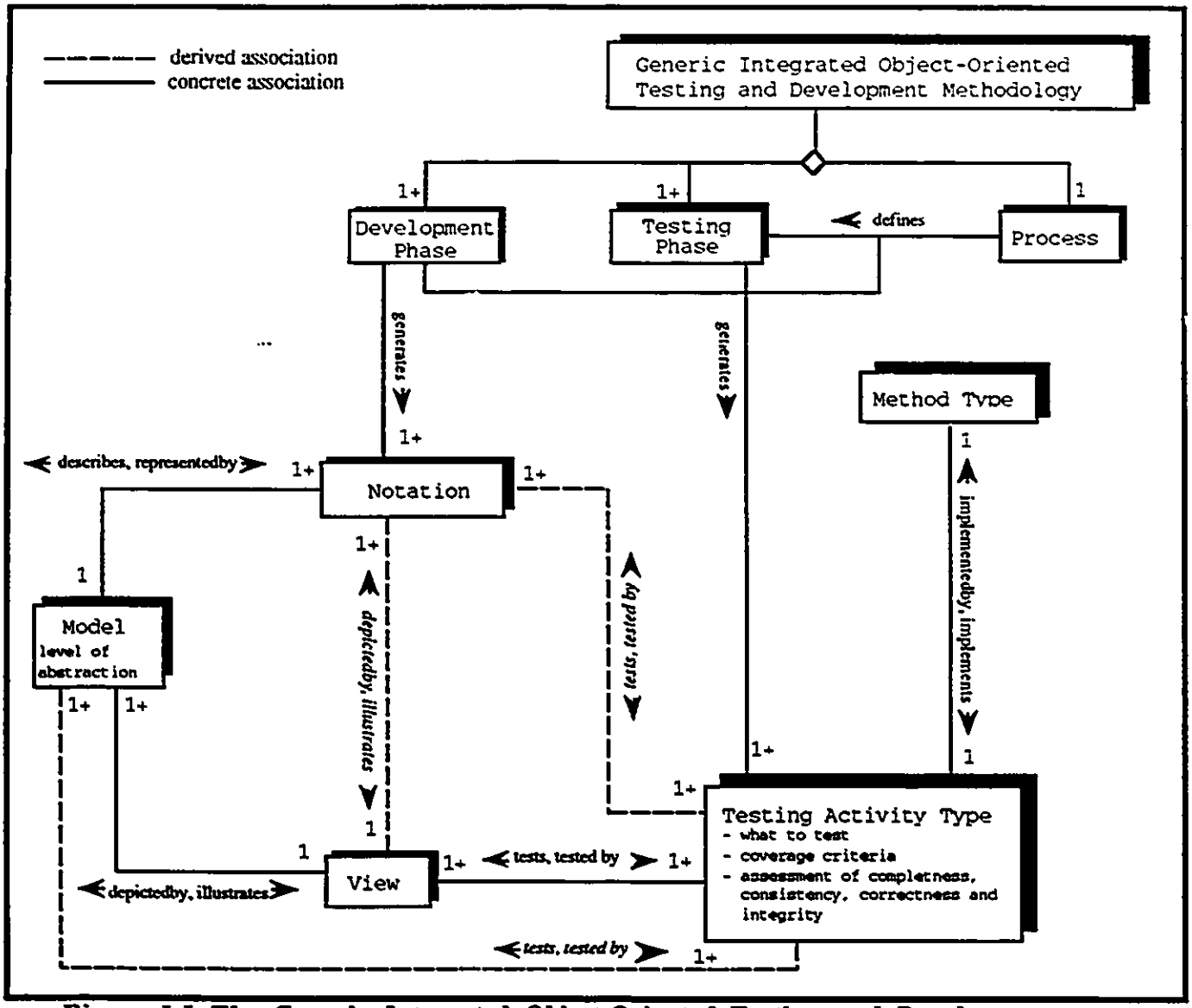


Figure 1.1 The Generic Integrated Object-Oriented Testing and Development Methodology

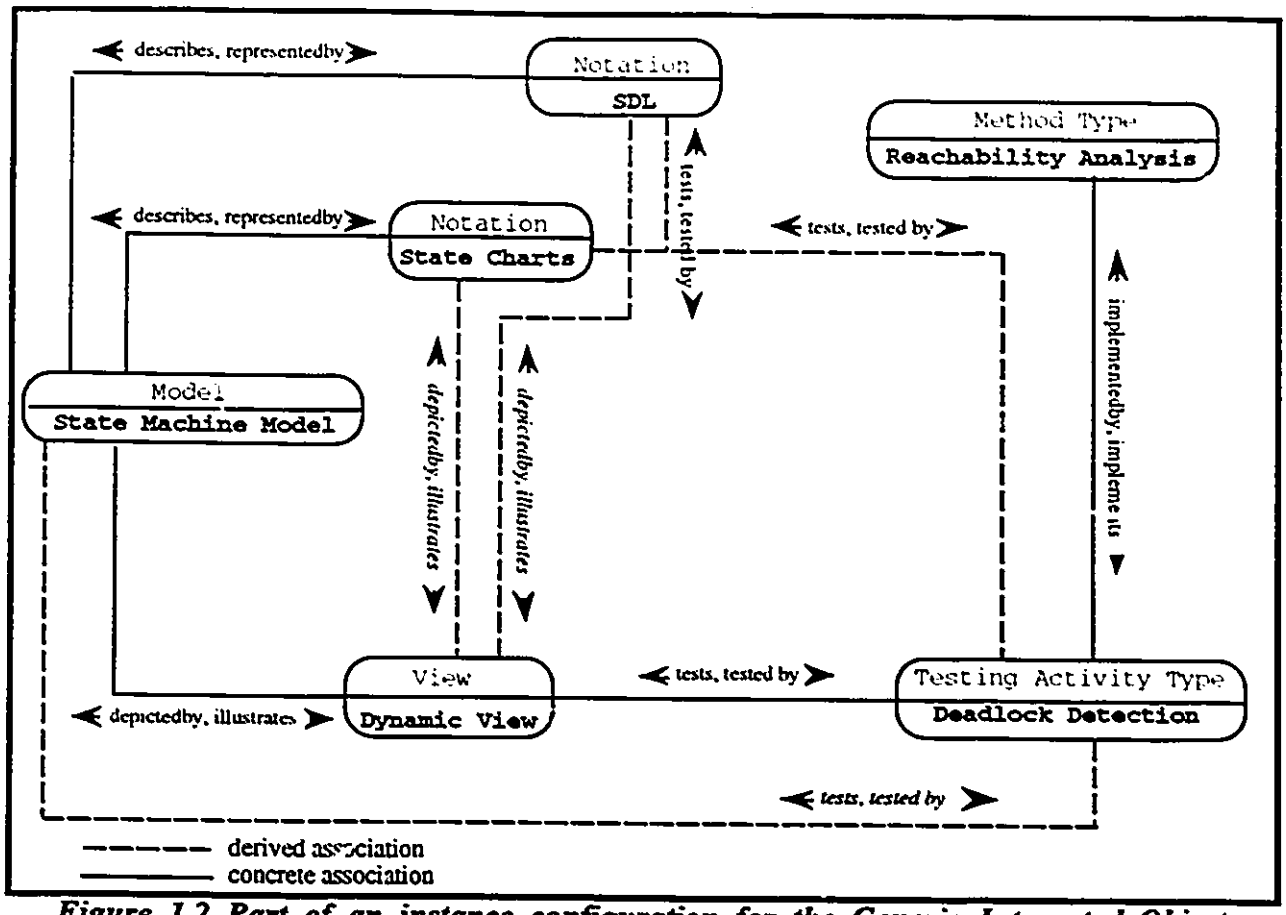


Figure 1.2 Part of an instance configuration for the Generic Integrated Object Oriented Testing and Development Methodology

1.4 Organization of the thesis

Chapter 2 describes background information and terminology used throughout this document. We provide an overview of the OO paradigm as well as a critique of its benefits as claimed by several authors.

Chapter 3 presents the fundamental factors required to provide testability in OO software, as described in [67]. The work in this document is based on 2 of these factors: (1) the characteristics of the representation (notations) and (2) the software process in which testing is conducted. In Chapter 3, we also provide our extensions to the first of these factors.

To address the second factor, in Chapter 4, we introduce a Generic Object-Oriented Development Methodology (GOODM) whose focus is to provide a way of handling change in a consistent manner. This is followed by a classification of Notations used in the OO world, into Models and Views, as described by our work. The rational for the classification is from a testing perspective and becomes apparent in Chapters 6 and 7.

Chapter 5 starts by providing the benefits for applying testing activity types early in the development process. We then describe the testing activity types we propose for GOODM.

Chapter 6 describes in details Verification Testing activity types with examples. Chapter 7 describes in details Validation and Incorporation Testing activity types also with examples. These examples verify the applicability of our proposed testing activity types.

Chapter 8 presents the Generic Integrated Object-Oriented Testing and Development Methodology by combining the work presented in Chapters 4, 6 and 7.

Chapter 9 concludes the work and proposes future research topic in the area of testing OO Models and Notations, as well as our proposal for tool development based on method type.

We also provide two appendixes at the end of this document. The first appendix provides an overview of OO programming testing challenges. The second describes notations used in the OO paradigm that are not very widely used.

Fig. 1.3 describes the organization of the work presented in this document.

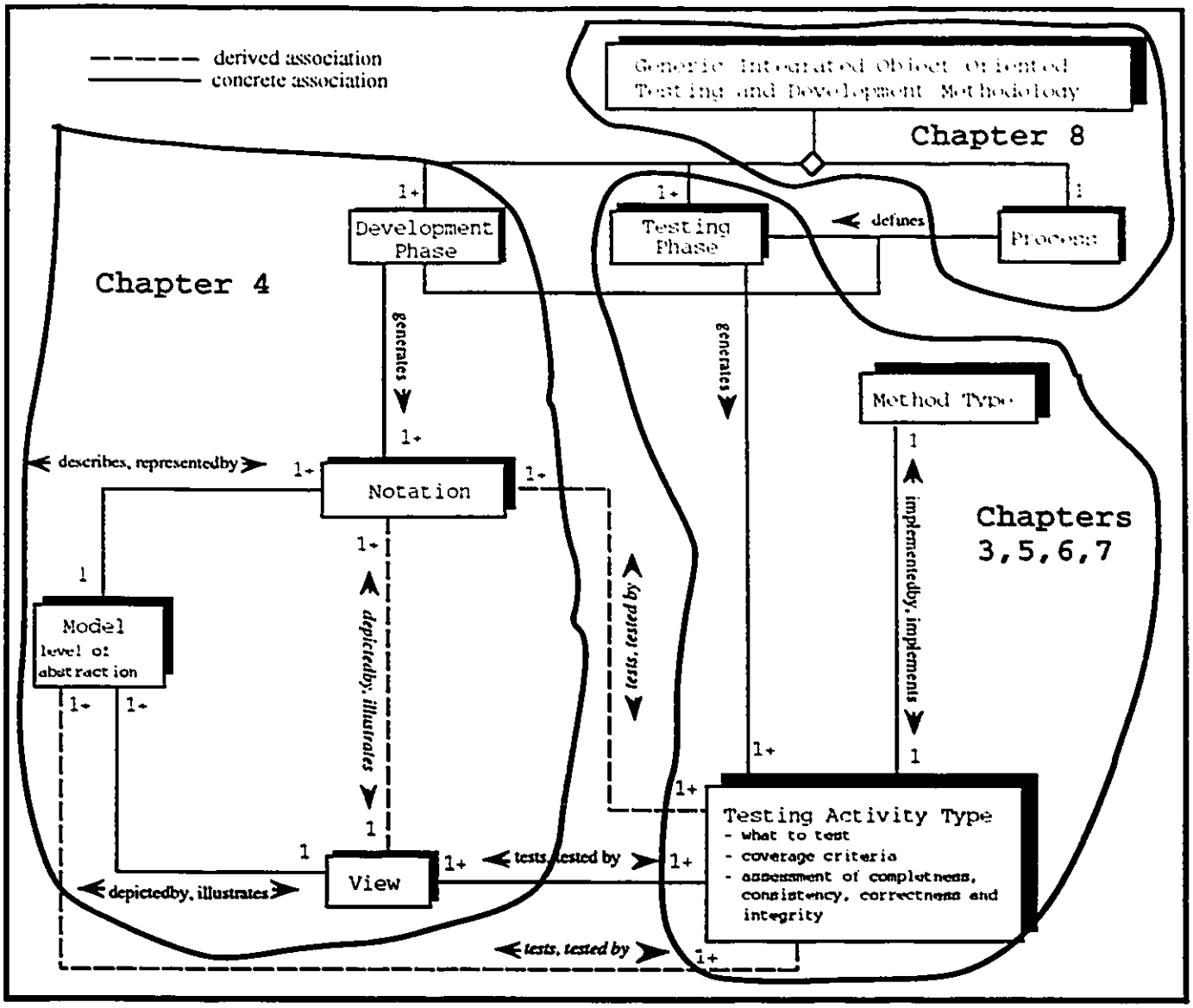


Figure 1.3 Organization of the thesis

Chapter 2 Background

2.1 Introduction

The purpose of this chapter is to describe background information and nomenclature used throughout this document.

We start by presenting the OO paradigm and its advantages over more established paradigms like the *function/data* paradigm. This is followed by a critique of the claimed benefits.

2.2 The Object-Oriented Paradigm

In the OO paradigm, the system to be modeled is regarded as a collection of objects that communicate via sending and receiving messages (i.e. operations) to accomplish a goal. This goal is usually expressed as a specification or requirement in a natural language description form. The system to be developed is not seen as set of nested procedures (derived by a top-down functional decomposition of the system) but as a conglomerate of interacting objects.

The following terms and their definitions will be used throughout this document.

The term *object* and *instance* are used interchangeable. They refer to an identifiable tangible entity that is able to save a state and offers a number of operations to either examine or affect this state. An object is seen only in terms of its interface which details the operations available for use by other objects; the implementation details of these operations remain hidden within the coded module that describes the object.

A *class* describes a group of objects (instances of a class) with similar attributes (instance variables), common operations, common relationships to other objects of other classes and common semantics.

The following illustrates the terms described.

```
CLASS:                Person
OBJECT:               aPerson
ATTRIBUTES:           weight, height
OPERATION:            eat
  IMPLEMENTATION:     how aPerson eats
OPERATION:            walk
  IMPLEMENTATION:     how aPerson walks
```

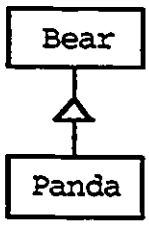
The following relationships among classes and object are presented. The notation used is OMT [14,39].

Relationships are the means that classes communicate. Links (an instance of a Relationship) are the means that objects communicate.

2.2.1 Inheritance

The inheritance relationship among classes denotes an *is-a* relationship. It acts as a mechanism for expressing commonality between two or more classes (*generalization*) or as a mechanism for specifying that one class is a special type of another class (*specialization*). The generalized classes are denoted as the *superclasses* of the specialized classes. The specialized classes are denoted as *subclasses* of the generalized classes.

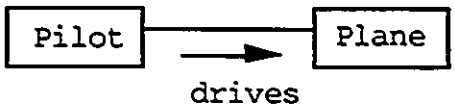
The figure below illustrates an Inheritance relationship between classes `Bear` and `Panda`. `Bear` is the superclass and `Panda` is the subclass. `Panda` inherits a copy of `Bear`'s operations and attributes, as well as all relationships that `Bear` is involved in. A subclass can have more than one superclass. This is refer to as Multiple Inheritance.



2.2.2 Association

An Association is a bi-directional relationship between two classes. An operation in class A invokes an operation of class B as part of its interface or implementation and in turn an operation in class B invokes an operation of class A as part of its interface or implementation. When an Association is uni-directional, it is sometimes refer to as a Uses relationship as described in [22].

The figure below illustrates a Uses relationship between a class `Pilot` and a class `Plane`. For the class `Pilot` to be associated with the class `Plane` means that the interface or implementation of `Pilot` can access or reference the interface of `Plane` but not its instance variables or private data. The class `Pilot` or an instance of this class is refer to as the Client. The class `Plane` or an instance of this class is refer to as the Server.



Associations can have a cardinality (i.e. the number of instances of the classes involved in the Association). In our example,

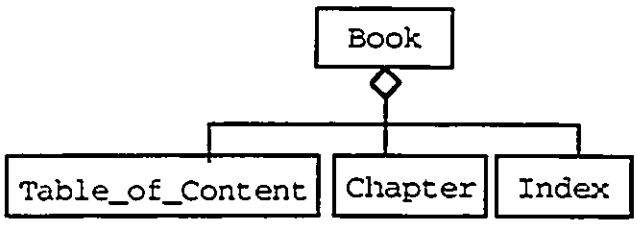
- *(1:1)* Uses relationship means that *1* instance of `Pilot` uses *1* instance of `Plane`.

- *(1:n)* Uses relationship means that *1* instance of `Pilot` uses *n* instances of `Plane`.

- *(m:n)* Uses relationship means that *m* instances of `Pilot` uses *n* instances of `Plane`.

2.2.3 Aggregation

The Aggregation relationship is a special type of Association between two classes, the Aggregation class and the Part class. The figure below illustrates an Aggregation relationship.



Class `Book` is referred to as the Aggregation class and classes `Table_of_Content`, `Chapter` and `Index` are referred to as the Part classes.

The Aggregation relationship can also have different cardinality. The Aggregation class can be view as a container of multiplicity 1.

- *(1:1)* Aggregation relationship means that *every* instance of `Book` must have *1* instance of `Index` (as an example) and that *every* instance of `Index` must be part of *1* instance of `Book`.

- *(1:n)* relationship means that *1* instance of `Book` must have *n* instances of `Index` and those *n* instances of `Index` must be part of the instance of `Book`.

There is a difference between the semantics of an Aggregation relationship and a Uses relationship. The Aggregation class can access the interface of its Part classes. The Client class can access the interface of its Server class. The reverse is not true for the Aggregation relationship. That is, for a Uses relationship, a Server class can in turn act as a Client for its Client if a reverse Uses relationship exists between the Client and Server; for an Aggregation relationship, a Part class cannot act as a Aggregation class of its Aggregation class. In terms of visibility, this means that the Part class cannot access the interface of its Aggregation class.

2.3 Why the Object-Oriented Paradigm

"The OO paradigm has a growing number of converts. Many people believe that OO will put a dent in the software crisis. There is a glimmer of hope that OO software will become more like engineering. Objects whatever they are now, may become for software what nuts, bolts and beams are for construction design, what *2-by-4s* and *2-by-6s* are for home construction and what chips are for computer hardware construction"[15]. Why is the OO paradigm so attractive?

The reason for this attention is based on the claims that OO technology aims at producing easier to model and to understand, extendible, reusable and reliable software. These claims are becoming facts as new applications of the paradigm emerge and substantial improves in productivity are reported [1,14].

However as stated in [15], before making the quantum leap, OO methods still have to prove themselves with respect to more established software development paradigms, specially for large-scale software (1000 classes and more). Large- scale projects like the Cooperation project at NCR, briefly described in [16], and the Generic Services Framework (GSF) project at Bell Northern Research [17] are starting to show that OO methods can indeed scale-up to such large applications.

As stated in [18], the existing paradigms for system development can be basically divided into function/data and OO paradigms.

The function/data paradigm distinguishes between function and data: the organization of a system is based on functionality and the manipulation of data structures is distributed between the resulting functions. The function/data division originated from the Von-Neuman [21] model where functions or procedures are active and have behavior, and data is passive and can be thought of as a holder of information which become affected by functions or procedures (i.e. *data flows* or *is sent between functions*).

The OO paradigm on the other hand views functions and data as highly integrated. This integration occurs in an *object*. In the OO paradigm, the system to be modeled is regarded as a collection of interacting objects that communicate via sending and receiving messages (operations) to accomplished a goal. Thus, the Von-Neuman model of passive data and active functions is not applicable in the OO paradigm, where data is active and functions are passive (i.e. *functions flow* or *are sent between data*).

We can provide a rational for the major benefits of the OO paradigm (i.e. aims at producing easier to model and to understand, extendible, reusable, and reliable software).

We will show how the basic features of the paradigm (i.e. abstraction, encapsulation and inheritance) can promote, but sometimes hindered the stated benefits.

2.3.1 Easier to model software

In [18], the author states that in the OO paradigm, the semantic gap between reality and what is to be designed is smaller than the one in the function/data paradigm.

The function/data paradigm sees a requirements specification as things to do [23]; the OO paradigm sees it as the set of items (entities) the things to do must be applied to. Requirement specifications are however, normally formulated in terms of what the system shall perform, what functionality the system should support and what items should exist in the system. In the function/data paradigm, the requirements specification must be reformulated into how this will be done for functional decomposition. Thus, we see a shift in the thinking process from *whats* to *hows*. In the OO paradigm, however, the system to be developed is structured around real life entity objects (the *whats*) from the problem domain; the internals of the system are then structured from this model. Thus, there is less semantic gap and no mind shift as in the function/data paradigm.

Therefore, applying the OO paradigm to system development results in software that is easier to model: development follows a natural thinking process from requirements specification (specified in terms of *whats* not *hows*) to system implementation. First the *whats* are identify: "similar things or instances in the problem domain are identified and abstracted as *objects*, characteristics of these instances are abstracted as *attributes* and associations between these instances are abstracted as *relationships*" [19]. Second the various operations and their effects on objects (classes) are identified. Finally, the *hows* are identified by combining specified operations of the objects (*whats*). Thus, one can think that the *hows* become a by-product of the operations on the *whats*.

2.3.2 Reusable software

The major vehicle for reusability in the OO paradigm is Inheritance.

With specialization, class libraries allow implementors to select from a library of classes, those classes that exhibit similar structure (i.e. attributes and relationships to other classes) and behavior (operations) to the required new classes, modify the selected classes and obtain the new classes to be implemented.

Other vehicles for reusability in the OO paradigm are *frameworks* and *patterns* [22]. A framework is a collection of classes that provide a set of services for a particular domain. A pattern is a reoccurring collaboration among instances of classes.

Inheritance allows software to be constructed from existing software. In the OO paradigm, reusable components are found or constructed from classes (i.e. objects) which embody actions (i.e. operations) and data. In the function/data paradigm, reusable components are found or constructed from functions or procedure which embody only actions. As stated in [20], it is difficult to build reusable components, especially in typed languages, if they embody actions alone and ignore data. Lets assume, we wish to reuse an action (i.e. function) to store an element (`employee_record`) in a list of employee records. We discover a potential function, `put (List, Element)`, that we could reuse. We quickly find that the function is closely coupled with the type of data it acts on (list of integers), since the function puts the element `Element` in the list `List`, only if it is greater than the first element in the list `List`. We can no longer reuse the function as is. If we wish to reuse the function, we need to add a conditional clause to verify the data type (integer versus employee record, in our example). Now assume, that a new requirement states that elements must be stored differently (e.g. LIFO or FIFO) for a list of integers than for a list of employee records. To satisfy this requirement and still use the same function, we must add a new parameter to the function indicating the type of put: `put (List, Element, TypeOfPut )`. As stated in [18], the problem with reusing a function in the function/data paradigm is that it is difficult to find the correct abstraction of the function to be reused, and thus one resolves to adding condition clauses to the function in order to satisfy the different data formats of new requirements.

What is needed is a general purpose function (`put`, in our example) that does not require knowledge of the implementation details of the function.

In the OO paradigm the solution is to reuse a `List` data structure that implements a generic `put` operation. The `List` data structure does not care about the type of elements it stores. Both the list of integers and the list of employee records could become subclasses of the generic `List` class and would be able to reuse the generic operations of the latter (e.g. `put`, `retrieve`, etc.). In the case of the list of integers, since it has special requirements for the `put` operation, it would redefine the operation in its subclass to store integers in a LIFO manner. The same occurs for the requirement stating that the list of employees must store elements in FIFO manner.

As shown, data (data structures, attributes, relationships) and actions (operations) are better components of reusability than actions alone. Thus, here the OO paradigm as a clear advantage over the function/data paradigm.

Another reason attributed to the difficulty of building reusable components in the function/data paradigm, is the very nature of functional decomposition: top-down. In [20], the author states that top-down development does not promote reusability, since the development of a function at level n in the tree-like top-down approach, is triggered by specific requirements at level $n+1$. The requirements exist in a specific context and it is in this context that the function will be developed: "if the need for a command line analyzer was encountered in a precise context, it is unlikely that the results will be usable for analyzing command lines with a different structure" [20]. In the OO paradigm, software tends to be developed using both Top-Down and Bottom-Up approaches. First classes (objects) are identified from the problem domain, then operations on these classes are identified, finally ordering of operations of classes are identified in order to fit function or procedures around these classes. This process is repeated and each time the artifacts (classes, operations, attributes and relationships) are refined (e.g. classes are generalize or specialized).

2.3.3 Easier to understand software

The statement that the OO paradigm produces easier to understand software is true to a certain extent and depends on what we are trying to understand.

If one is trying to understand the relations between the system and reality (requirements specification), it is easier in the OO paradigm than in the function/data paradigm as shown in section 2.3.1, since the semantic gap between the system and reality is smaller.

If one is trying to understand the set of manipulations (not their effects) and definition of data structures, then it is also easier in the OO paradigm. This is because of abstraction and that information is reasonably local to an object (note that even with inheritance, the search is significantly easier than in the function/data paradigm since searching occurs always up in the class hierarchy). *Abstraction* allows an object to be seen only in terms of its interface which details the manipulations (operations) available on that object. In the function/data paradigm a data structure is scattered among the manipulations (i.e. functions) that use it, therefore it is more difficult to search for that information: one must follow the invocation order of functions that manipulate the data structure.

If one is trying to understand the effect of a manipulation on a data structure, then it is easier in the function/data paradigm because of the nature of the paradigm (i.e. input, process, expected output). In the OO paradigm the effects of manipulations are encapsulated within an object and therefore harder to observed from the outside of the object (e.g. an object state is completely hidden within an object's implementation).

If one is trying to understand collective behavior of objects (here we mean functionality, i.e. a collection of objects' operations with a goal), then it is more difficult in the OO paradigm. This is because of the *localization* problem described in [9]: an object may provide many functions and one function may involve several objects. Also, the fact that there is typically no Main function or Program that drives the execution of the system as in the case of the function/data paradigm, adds to the complexity of understanding objects interactions. These object interactions may be exponential and many may not lead to a certain function required by the system. In the function/data paradigm, this is not the case. A function or procedure is a powerful abstraction mechanism; complex behavior can be composed out of or decomposed into simpler units, thus allowing easier understanding of the overall functionality of the system.

2.3.4 Extendible and reliable software

Reliability and extendibility should not be discussed separately since one should talk of how reliable a system is after it undergoes an extension (i.e. how safe are the extensions). A possible measure for this can be the effort spend testing an extension to ensure reliability.

The statement that the OO paradigm produces more extendible and more reliable software is based on the observations of Meyer's in [20]: "In the evolution of a system, functions tend to be the more volatile part, ... over time, data structures are the really stable aspects of a system" [20]. Since in the OO paradigm, data is primary and functions are secondary, software ought to be more extendible. This is true, but we will show that this is dependent on the type of extension. Note that the type of extensions presented in the function/data and OO paradigm are not at all exhaustive, but we believe we cover the most evident type of extensions.

In the discussion that follows, unless specified otherwise, an extension will be referred to as a change that can be an addition, deletion or modification (e.g. renames).

In the function/data paradigm changing a function or procedure can be difficult depending on the type of change: (1) if its parameters are changed, then all calls to the function or procedure must be updated and retested; (2) if its functionality or behavior is changed, then all functions or procedures that call it must be retested; (3) if a data structure is modified or deleted, then one must update and retest all the functions or procedures that act on that data structure; (4) if a data structure is added, then new functions or procedures may need to be added and tested to act on the new data structure and existing functions or procedures may need to be updated with conditional clauses and retested to accommodate the new data structure; (5) changes to variables (whether local or global) will require updating and retesting all nested functions or procedures that use the variable; (6) finally, as shown in section 2.3.2, changes required to a function to accommodate reusability, will usually end-up cluttering the function or procedure with conditional or case statements to verify new data types, thus resulting in hard to read programs.

In the OO paradigm, as in the case of the function/data paradigm; (1) if the interface of an operation in an object is changed then all operation that call that object must update their parameters; (2) if the functionality or behavior of an operation is changed, then encapsulation claims that as long as the interface of the operation remains unchanged, the changes should be transparent to objects that invoke the operation, however the changed operation should be retested to ensure that its functionality is the same as in the previous version; (3) if attributes of a class are changed, then all classes that set or read the attributes must be updated to either call new operations on those attributes (if the attributes are new), delete calls to operations that set or read the attributes (if the attributes are deleted) and possibly rename the operation that set or read the attributes (if the attributes are renamed).

Due to inheritance, when data structures are changed in the OO paradigm, extendibility becomes a challenge. Changes to a superclass can lead to quite unstable class hierarchies that require much retesting, since all subclasses may potentially be affected by the change. Changes to subclasses can also cause side effects as explained in [4]. Fig. 2.1 illustrates this. Class B inherits instance variable v and operations `m1` and `m2` from class A. Class B implements operation `m3` which initializes variable v to 1. The following order of method would cause problems: `m3, m2, m1` in class B since `m2` uses v initialized to `0` and not `1`.

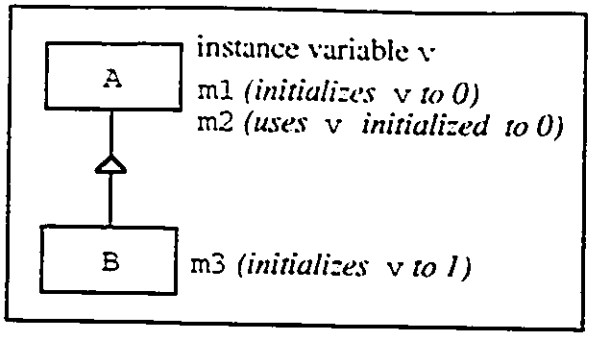


Figure 2.1 Addition of a new operation in a subclass can cause undesirables side effects

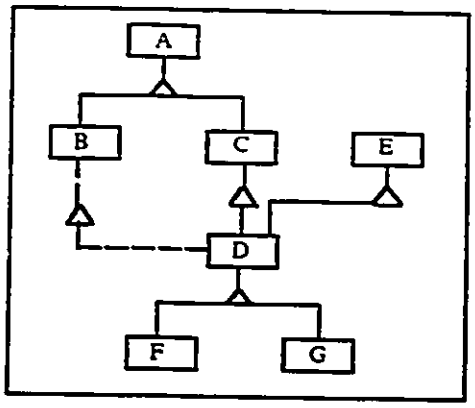


Figure 2.2 Changes in the connectivity of a class structure can cause undesirables side effects

Changes in the connectivity (i.e. adding, deleting or changing inheritance links) in a class structure can also cause unwanted side-effects. Consider the class structure depicted in Fig. 2.2: removing the link between classes D and E, implies that all operations in E can no longer be invoked by D and its subclasses; adding a link between D and B causes Multiple Inheritance in D from B and C and Repeated Inheritance from A, and their unwanted side-effect (for more details, please refer to the Appendix on Object-Oriented Programming Testing Challenges, at the end of this document).

Overall extensions of software in the OO paradigm can cause unreliable systems. In [18], the author observes that the type of changes that can cause unreliable systems could occur less often then safe changes, however, he states that the claim has not yet been scientifically investigated.

Finally, every paradigm has its advantages and disadvantages; the following quote from [23] illustrates our point of view the best: "The key issue is not whether the system is object- or function- structured but whether it is organized primarily around transactions or data types. Within either paradigm, a dual solution is possible that reverses the advantages".

Chapter 3 Testability in Object-Oriented Software Development

3.1 Introduction

The purpose of this chapter is to describe the concept of Testability in software development. We will present the fundamental factors required to provide testability in OO software, as described in [67].

3.2 Testability

As stated in [41], testing is tedious because it requires the derivation of large amounts of data. It is boring because of its repetitive nature and it is unavoidable because it is often the only way to detect errors in a system.

Testability is as a measure of how easy (less complex, less tedious, less boring, less costly) the effective testing of the implementation is made [3].

3.2.1 Origins of Testability

The term Testability originated in hardware design, more specifically in circuit and in semiconductor fabrication [42,43]. In this context, testability is defined as a measure of how easily a piece of hardware (circuit) can be tested to ensure it performs its intended function. A circuit that can be completely tested with minimal time and effort possesses the greatest degree of testability [44].

The advantage, among others, of introducing testability features in hardware design allows (1) failures to be detected early in the process, thus permitting appropriate action to be taken early in the process, (2) reduces the amount of time required to completely test a product and (3) reduces the need for external test equipment by including low cost features as an integral part of the product. Therefore, it increases the quality of the product, thus increasing customer satisfaction, reduces test cost, manufacturing costs and maintainability costs, and it assists engineering debugging or testing [44].

A few myths that delay the introduction of testability into hardware products are discussed in [44]. The major myth is that testability features are expensive to introduce. To this, the authors respond that a modest investment in testability results in a large improvement in fault detection and isolation. They also state that apparent savings gained by the omission of testability will ultimately be lost by longer development schedules and increase manufacturing costs. In [43], the author observes that products designed with testability features have a 10% to 40% reduction in the product's time to market.

In [45,46] a success story is described by Resdel Engineering where compliance with MIL-STD-2165 was required for a new avionics equipment for the U.S. navy. The requirements for the sonobuoy receiver system were very stringent and included redundancy and BIT (built-in-test) and BITE (built-in-test-equipment) to detect 98% of the failures. The product was delivered on schedule, problems were encountered and tackle in the early design stages, thus achieving a proper balance between performance, cost and supportability of the system assemblies.

3.2.2 Testability in Object-Oriented Software Development

Software development, as hardware development, is faced with the same challenges of competitive markets, aggressive schedules and stringent specifications. Thus the advantages of providing testability in software development are recognized both by industry [47] and research [7,48,49,67]. What are testability requirements and how to introduce testability in software engineering is still a popular topic of research. In this sense software engineering is behind hardware engineering.

Testability was defined as a measure of how easy (less complex, less tedious, less boring, less complex) the effective testing of the implementation is made.

In [67], the author conceives testability as a result of the following fundamental factors:

- the characteristics of the implementation
- the test support environment
- the characteristics of the representation
- the software process in which testing is conducted

The last two factors are the focus of our work. We will show, in the following sections our contribution to the latter.

3.2.2.1 The characteristics of the implementation

This is not within the scope of the work presented in this document, but it is briefly described based on the work in [67].

There are two ways of achieving testability of OO implementations. First, by allowing Built-In test capabilities, such as set/reset reporters and class assertions, which allow to effectively control and observe the source code. Second by *implementing for testability*, that is implementing by following some basic accepted principles of OO programming which provide controllability and observability, described in the next section. An example of such a principle is low coupling between classes. Classes that have high coupling (i.e. high interdependencies) are typically more difficult to control, thus reducing testability [67].

3.2.2.2 The test support environment

This is also not within the scope of the work presented in this document, but it is briefly described based on the work in [67].

Providing an effective test support environment will increase testability. For example, test-suites that can be reused, increase the amount of testing that can be done and hence increase testability. Also, test tools alleviate the testing of the source code greatly. For example dynamic testing tools such as program instrumentators (i.e. programs that augment source code with probes or variables to determine execution paths) and assertion processors allow to execute the source code and assess about its correct execution. Static testing tools allow to detect code that will never be executed, uninitialized variables, etc. [41].

3.2.2.3 The characteristics of notation

To provide testability, the author in [67], defines basic characteristics of notations[1] they must be verifiable, modifiable, traceable, complete, unambiguous and consistent. He states that test cases can be developed from notations that exhibit these characteristics, thus contributing towards testability.

These characteristics are mentioned in [67] but no definition is provided. We believe that they are the basic components of testing activities since they provide what needs to be tested. Our contribution is this area is to provide concrete definitions of the above characteristics taking into account the iteration style of development of OO systems.

First the characteristics should be divided into *testability requirements* and *testability attributes*. Notations that conform to *testability requirements* can exhibit *testability attributes*.

3.2.2.3.1 Testability Requirements of notations

Verifiability, modifiability and traceability are basic *testability* or *verifiability requirements* of notations:

(1) A **verifiable** or **testable** notation[2] can be tested to be assessed for its completeness, correctness, consistency and incorporation.

(2) **Modifiability** is directly related to verifiability: a modifiable notation should provide a set of legal transformations that can be verified. We discuss this in more details in Chapters 6 and 7.

(3) **Traceability** becomes important when a group of notation is being considered to make semantic connections between different notations. Traceability does not necessarily

---

[1]Note that in our discussion, the term *notation* corresponds to syntax of a representation.

[2]A *verifiable* or *testable* [46,51,62,63] notation is
Partionable (can be divided into independently testable units to facilitate the insertion of points of control and observation between these units)
Controllable (can be interactively set to a particular internal state or behavior)
Observable (allows a change of an internal state or behavior to be determined by observing externally behavior(s))
Processable (allows its test results to be easily interpreted and analyzed)

need to be an integral part of the syntax of a notation since annotations or tools like Requirements Tracing and Management (RTM) [68] and Dynamic Object-Oriented Requirements System (DOORS) [70] can capture traceability information. We define the following types of traceability that notations should exhibit:

horizontal traceability : Notations at a given phase (e.g. Analysis) of an iteration of the development process are complete[3] and correct[4] with respect to other notations used at that phase (Analysis in our case).

vertical upwards traceability : Notations at a given phase (e.g. Design) of an iteration of the development process are consistent[5] with respect to notations used at a previous phase (Analysis in our case) for that iteration. Thus, in our example, it is possible to assess whether a design reflects the analysis for a given iteration.

vertical downwards traceability : Notations at a given phase (e.g. Analysis) of an iteration of the development process are consistent with respect to notations used at the next phase (Design in our case) for that iteration. Thus, in our example it is possible to assess whether the analysis is reflected in the design for a given iteration.

forward iteration traceability : Notations at phase n of iteration i are incorporated[6] in notations at phase n of iteration $i+1$.

backward iteration traceability : Notations at phase n of iteration $i+1$ are incorporated in notations at phase n of iteration i.

[3] *Completeness* is a testability attribute of notations; it is described in the next section.
[4] *Correctness* is a testability attribute of notations; it is described in the next section.
[5] *Consistency* is a testability attribute of notations; it is described in the next section.
[6] *Incorporation* is a testability attribute of notations; it is described in the next section.

3.2.2.3.2 Testability Attributes of notations

Notations that conform to the testability requirements described in the previous section exhibit testability attributes since

- a *verifiable* notation can be tested for testability attributes
- a *modifiable* notation can accommodate changes to achieve testability attributes
- a *traceable* notation can be traced to other notations to achieve testability attributes.

We define the testability attributes of incorporation, completeness, correctness and consistency. The last three were selected by the authors in [8] as the basic attributes of notations. The incorporation attribute is proposed by our work, to accommodate the iterative type of software development, where notations must be consolidated (i.e. incorporated) from one iteration to the next. This is explained in more details later in this section.

We expand the discussion in [8] by providing details about the aspects of notations (i.e. syntax and semantics) and expanding on the definition of testability attributes. We will show in Chapters 6 and 7, which testing activity types test for completeness, correctness, consistency and incorporation, as well as how.

The Aspects of notations

The aspects of notations are illustrated in Fig. 3.1. The figure illustrates our proposed class configuration for a `Notation` based on the work in [38].

A `Notation` is a graphical or text based representation. It conforms to a `Syntax`, which generates a set of `Statements` that make up a `Language` for the `Notation`. It is this `Language` that provides the semantic for the `Model`. A `Syntax` is formed by an `Alphabet` and a `Grammar`. An `Alphabet` contains a set of `Modeling Constructs`, each of which has a unique `Representation`. The `Grammar` contains `Rules` that define how to legally combine `Modeling Constructs`. Fig. 3.2 illustrates an instance configuration of Fig. 3.1 for the State Charts notation.

As stated in [38], the syntax of notations relates the notation to the language notation by describing relations among the language constructs without considering their meaning. The semantics of notations relates notations to the domain by considering the meaning of relations among statements. In the latter case, we can say that a notation aims at providing the semantics to describe a View (i.e. Static, Communication, Dynamic and Processing, these are described in Chapter 4).

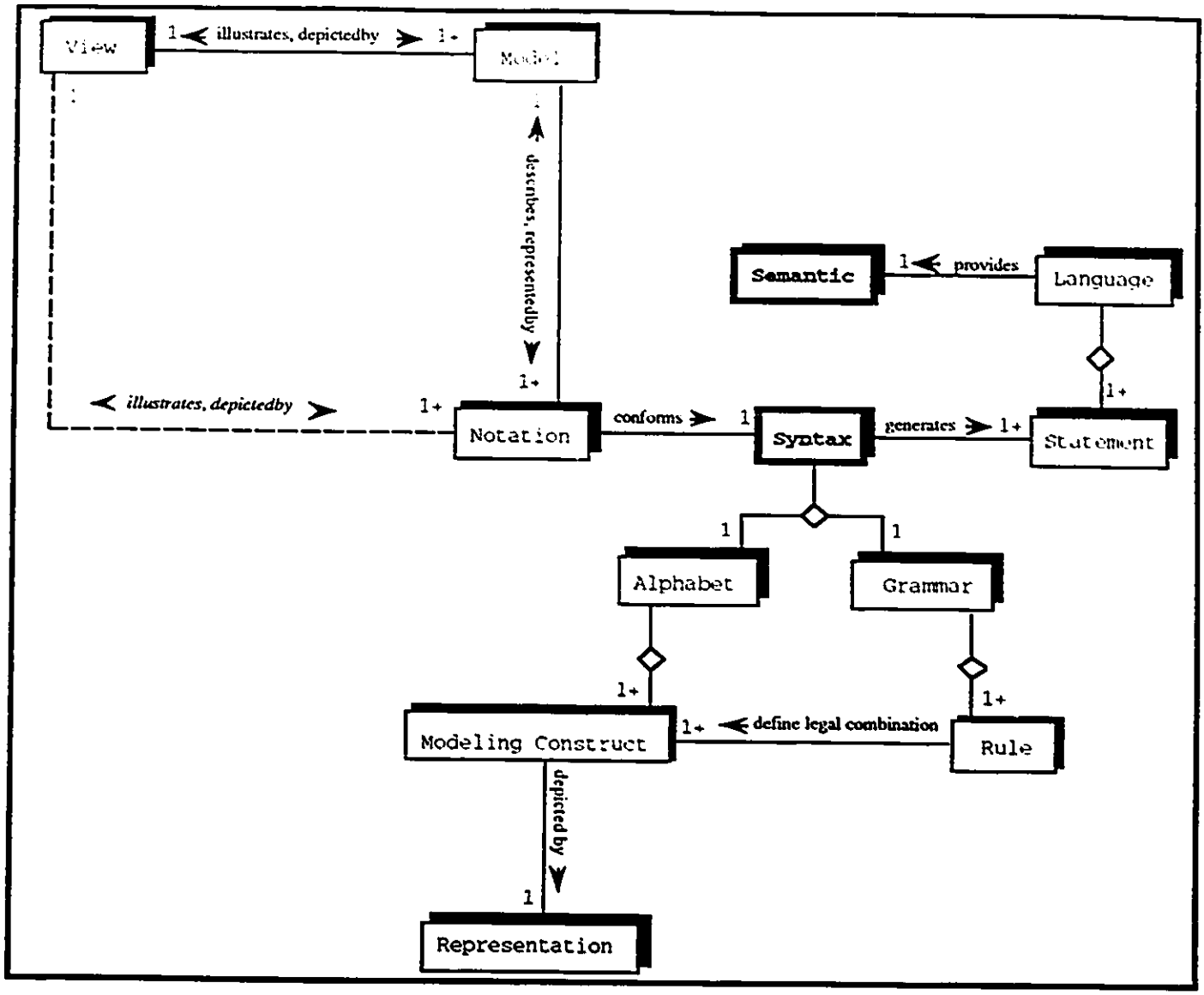


Figure 3.1 The Aspects of Notations

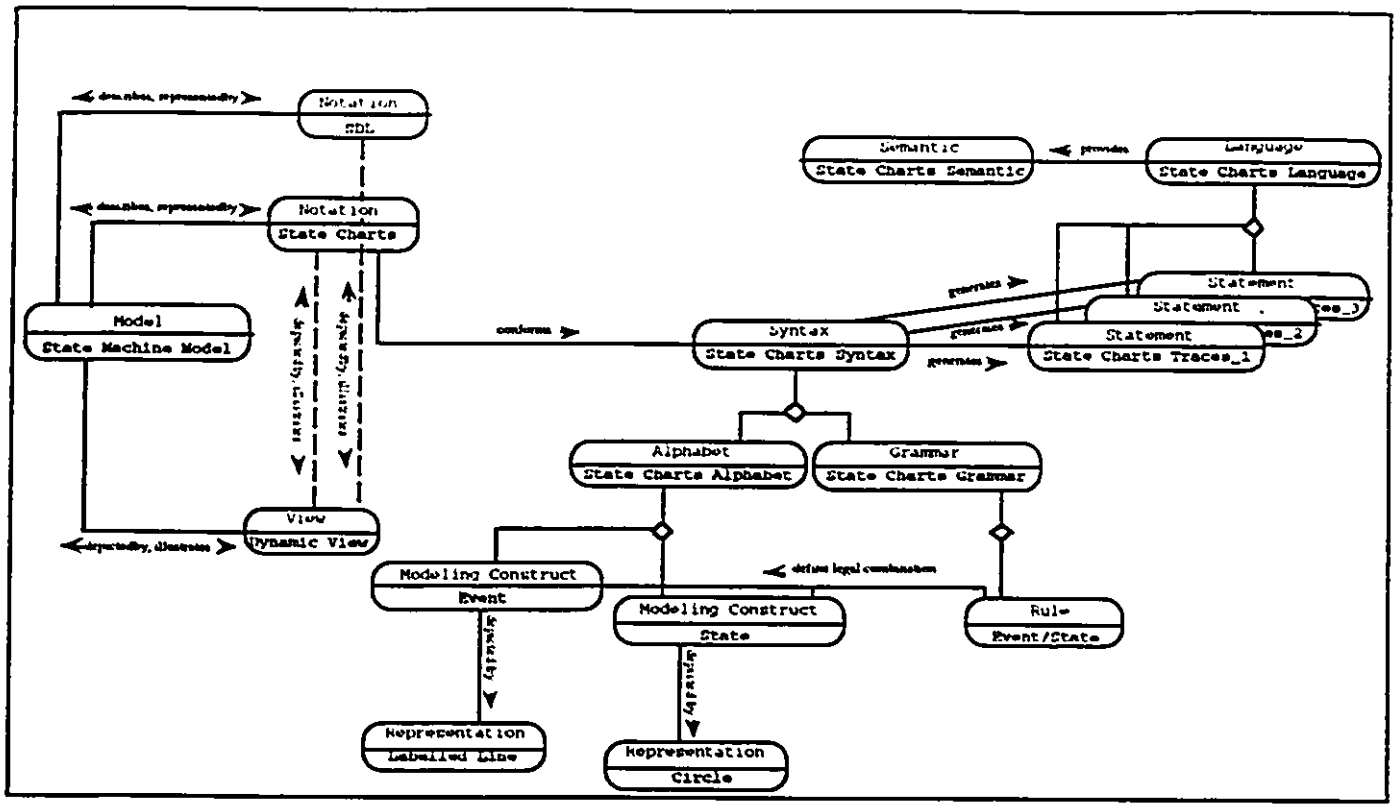


Figure 3.2 An Instance Configuration for the State Chart notation

The Attributes of notations

Completeness

Completeness is defined by the authors in [38] as the attribute exhibit by models when everything that the system is supposed to do is included in the software, the system responds to all classes of input data and there are no requirements marked to be determined.

In terms of notations, our definition states that "notation A is complete with respect to notation B, when the semantics of notations A, describes no more or less then what is required by the semantics of notation B".

Thus, we can talk of *semantic completeness*. *Semantic completeness* is necessary to achieve the concepts of Horizontal Traceability (section 3.2.2.3.1).

Correctness

Correctness is defined by the authors in [38] as the attribute exhibit by models when "every requirement represents something required by the system to be built and no error will affect the design". In terms of notations, our definition states that "notation A is correct when

(1) its statements conform to the syntax of the notation AND
(2) no errors occur in the semantics it provides."

Thus, we can talk of *semantic correctness* and *syntactic correctness*. *Semantic correctness* and *syntactic correctness* are necessary to achieve the concept of Horizontal Traceability (section 3.2.2.3.1).

Consistency

Consistency is defined by the authors in [38] as the attribute exhibit by models when "no set or requirements is in conflict with any other set".

In terms of notations, our definition states that "notation A is consistent with notation B when the semantics of notation A is not in conflict with the semantics notation B and vise-versa". More specifically,

(1) every modeling construct of a notation at phase n of the development process is present or rationalized (if deleted or changed) at phase $n+1$, and that

(2) every modeling construct at phase $n+1$ of the development process is a valid transformation of a modeling construct at phase n.

Thus, we can talk of *semantic consistency*. *Semantic consistency* is necessary to achieve the concept of Vertical Upwards Traceability and Vertical Downwards Traceability (section 3.2.2.3.1).

Incorporation

Incorporation is an attribute of notations proposed by our work to denote that the semantics of notations at phase n of iteration i are incorporated (i.e. taken into account) in the semantics of notations at phase n of iteration $i+1$ and vise-versa.

Thus, we can talk of *semantic incorporation*. *Semantic incorporation* is necessary to achieve the concept of Forward and Backward Iteration Traceability (section 3.2.2.3.1).

We have defined the following testability attributes of notations.

- semantic completeness
- semantic consistency
- semantic correctness
- semantic incorporation
- syntactic correctness

We say that notations that exhibit the above attributes provide high testability (i.e. the implementations derived from such notations are easier to test). Fig. 3.3 illustrates how the attributes of notations relate to traceability.

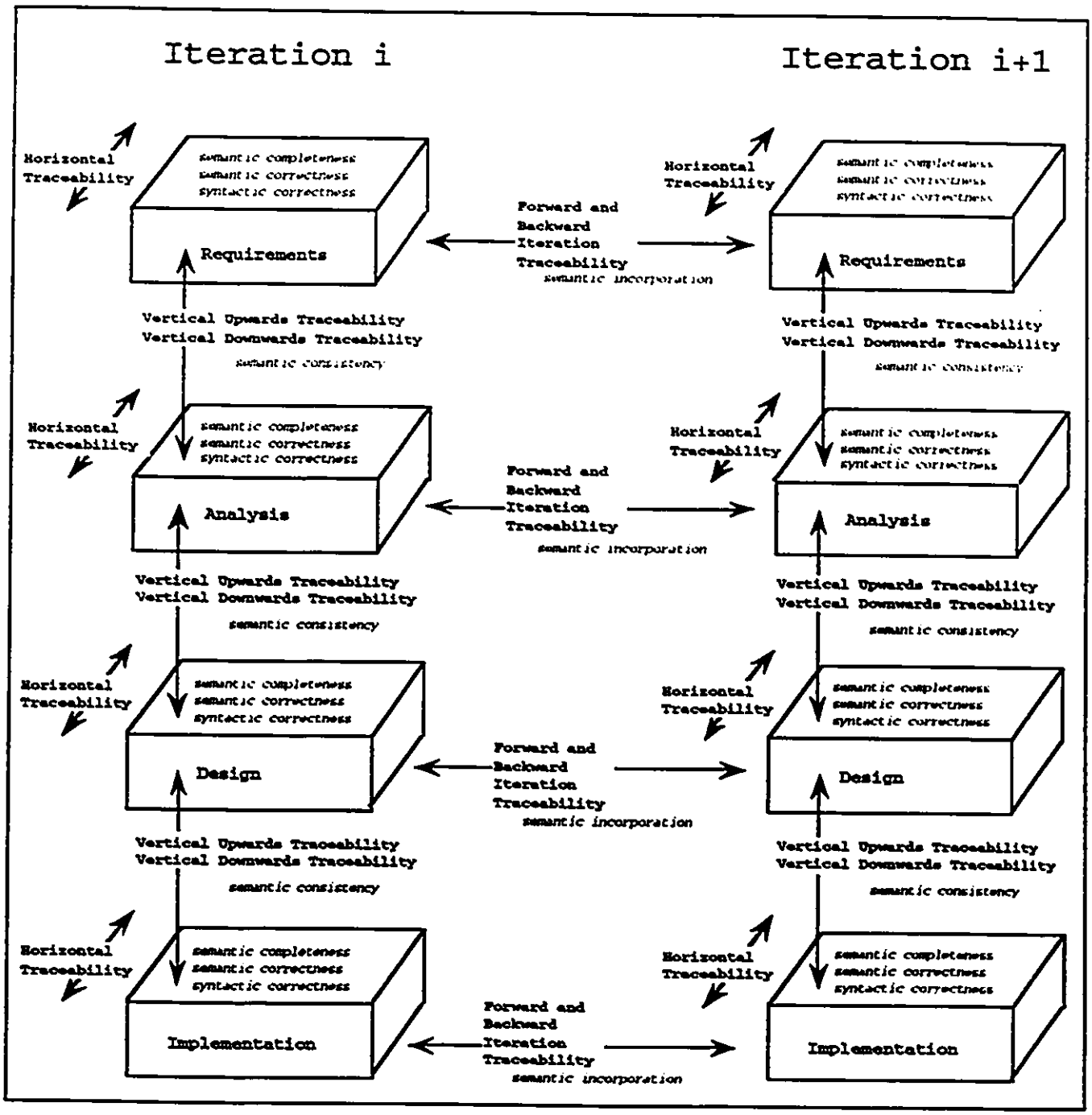


Figure 3.3 Attributes of notations and Traceability

Rational for the selected attributes of notations

As stated in section 3.2.2.3, no concrete definition of testability attributes of notations currently exists. Our selection was based on two constraints or requirements:

(1) we needed to take into account the iteration style of development of OO systems, and

(2) at a given phase of development no two same Views (i.e. group of notations describing the same semantics) can be provided.

We believe that the selected attributes and their definitions, best illustrate the above two requirements.

Completeness and *correctness* are attributes that are applicable to a given phase of the development process as illustrated in Fig. 3.3. Based on requirement (2) above, these attributes are judged against the different Views provided at given phase. Thus, we talk of *completeness* and *correctness* between different Views for a given phase.

Consistency, on the other hand, is an attribute applied to different phases of the development process as illustrated in Fig. 3.3. The attribute is judged against the mappings from one View at a given phase to the same View at the next (more detailed) phase. Thus, we can talk of *consistency* of a View through different phases.

Some important observations should be made at this point.

Why is consistency not a Horizontal Traceability attribute?

Consistency is an attribute that exists only in the context of the same View (i.e. we say that the Communication View at Analysis is *consistent* or in conflict with the Communication View at Design). Horizontal Traceability deals with Views provided at a given phase of the development process. At a given phase, no two same Views can be provided (e.g. we cannot have more than one Communication View at Design, therefore we cannot talk of consistency between the two Views).

Why are completeness and correctness not Vertical Traceability attributes?

Consistency deals with identifying conflicting semantics between two same Views at different phases of the development process. Our definition of consistency, as shown below, actually involves detecting completeness and correctness.

(1) every modeling construct of a notation at phase *n* of the development process is present or rationalized (if deleted or changed) at phase *n+1*, and that

(2) every modeling construct at phase *n+1* of the development process is a valid transformation of a modeling construct at phase *n*.

'every' deals with completeness and 'is present or rationalized' and 'is a valid transformation of' deal with correctness.

Why create the consistency attribute if consistency includes completeness and correctness?

The basic rational is to be able to differentiate the context in which *completeness* and *correctness* can be applied. The *consistency* attribute includes *completeness* and *correctness* but in a different context then in the original definitions provided above; while the context was: different Views at a given phase, the context in *consistency* is: the same View at different phases.

Also, the *completeness* and *correctness* in *consistency* are judged against the goals of the current iteration (this is described in Chapter 7). The original *completeness* and *correctness*, as described above, are judged against the semantics provided by a View.

Why create the incorporation attribute

Our definition of *incorporation* states that notations must take each other into account from iteration to iteration. The term "taken into account" can be open to different interpretations. This is purposely so.

When moving from one iteration to the next, the designer may be dealing with a totally new component (in the case that iterations where determined base on functional components of the application to be developed) or he/she may be dealing with more detail of the application to be developed (in the case that iterations where based on refinements).

Lets take an example of the first case:

- iteration *i* has developed component *fci*,
- iteration *i+1* developed component *fci+1*.

In this case, the *incorporation* at every phase (i.e. the interpretation of "taken into account") is well defined and measurable since it is based on the specification of the interactions between *fci* and *fci+1*. For example, the *incorporation* could be that an instance of a class of *fci* invokes an operation in an instance of a class of *fci+1*.

Lets take an example of the second case:

- iteration *i* has developed component *fci* with level of detail *di*
- iteration *i+1* continue development of component *fci*, with level of detail *di+1*

In this case, the *incorporation* at every phase (i.e. the interpretation of "taken into account") is not well defined and very difficult to measure since it is based on the refinement of *fci* . Here, the *incorporation* attribute has the flavor of *consistency* since we can say that:

(1) every modeling construct of a notation used to express the semantics of *fci* at phase *n* of iteration *i* is present or rationalized (if deleted or changed) at phase *n* of iteration *i+1*, and that

(2) every modeling construct of a notation used to express the semantics of *fci* at phase *n* of iteration *i+1* is a valid transformation of a modeling construct at phase *n* of iteration *i*.

The rational for using notations that conform to the basic testability requirements of verifiability, modifiability an traceability is that these notations can be assessed or tested for testability attributes (i.e. consistency, correctness, completeness and incorporation) through testing activities. This provides the ability of:

(1) uncovering errors early in the process where they are less costly to fix
(2) generating test cases that can be applied to the implementation
(3) guiding the testing of the implementation.
(4) facilitating the location of source(s) of error(s) for modification

We will discuss these separately in Chapters 5, 6 and 7.

3.2.2.4 The software process in which testing is conducted

The software process in which testing is conducted is one of the basic factors for achieving testability as described in section 3.2.2. The authors in [8,67] state that the process should closely integrate development and testing activities and ensure that sufficient resources (staff, time, funding) are allocated for these testing activities. This must be accompanied with creating a mind set in the organization of commitment to quality.

We observed that there are two major areas that are omitted in the process described in [8,67].

First, the process should enforce a consistent method of dealing with change (i.e. error correction, enhancements, new requirements, etc.). This is discussed in Chapter 4.

Second, the process should determine what are the testing activities that should take place at different phases of the OO software development. These are discussed in Chapters 5, 6 and 7.

Chapter 4 Notations, Models and Views

4.1 Introduction

There is a great deal of literature in the area of OO software development methodologies, [14,15,18,22,24,25,26,27,28,29,30,31,32], among others. "The existing research can be divided into three broad categories: (1) processes only (2) representations only and (3) processes and representations. The first category, processes, refers to procedural methods for performing OO analysis or design, or some particular aspect of analysis or design. These processes do not include OO analysis and design (OOAD) diagrams or notations. The second category, representations, refers to graphical notations or diagrams for depicting the output of OOAD. The focus is on visually representing a design not on how to derive a particular design. The third category, processes and representations, encompasses both processes for performing (some subset of) OOAD and representations for specifying results." [35].

Little effort has been spent in standardization mainly because different applications require different software development methodologies (i.e. Processes, Models and Notations). "Convergence of certain methodologies is more likely to happen as opposed to standardization" [33].

The authors in [18,34,35,36,37], provide excellent comparisons, categorizations and requirements of OO Processes, Models and Notations.

The intend of the work in this section is not to propose yet another methodology or to provide a critique of the existing methodologies. We gather different Processes, Models and Notations, combine them and construct a Generic Object-Oriented Development Methodology. This methodology is the basis for our proposed Generic Integrated Object-Oriented Development and Testing Methodology, described in Chapter 8. The focus of the methodology is to provide a way of handling change in a consistent manner and the integration of development and testing activities. This is one of the key factors in providing testability in OO software as described in Chapter 3. To arrive to generic testing activities (i.e. testing activity types), we classify notations used in the OO world into Models and Views. The rational for this categorization is purely from a testing perspective and becomes apparent in Chapters 6 and 7 where we illustrate that testing activity types test Models.

4.2 Definitions

The purpose of this section is to describe the methodology nomenclature that will be used throughout the rest this document. The discussion is based on Fig. 4.1 which is a part of Fig. 1.1 described in Chapter 1.

Object-Oriented Methodology: The `Generic Object-Oriented Development Methodology` is the set of artifacts (`Process`, `Models` and `Notations`) used in the development of OO software.

Process: An OO `Process` defines the development phases (e.g. Analysis, Design, etc.) and their ordering of occurrence. It also defines the type of development (i.e. Spiral versus Waterfall [10], Top-Down versus Bottom Up [20]). Finally, a `Process` should define how `Notations` used at a phase (e.g. Design) of an iteration of software development are derived from `Notations` of a previous phase (e.g. Analysis) and vice-versa. This is to ensure that `Notations` are correct, complete, consistent and incorporated in themselves and with respect to other `Notations` used in the software development process.

Model: A `Model` corresponds to the semantics described by a set of `Notations` (e.g. the State Machine Model is the semantic described by the SDL [12] and State Charts [13] notations, among others). Models are described in section 4.3.3.

View: A `View` is a higher level of abstraction then a `Model`. It corresponds to the semantics described by a set of `Models` (e.g. the Dynamic view of a system is the semantic described by the State Machine Model, among others). Views are described in section 4.3.4.

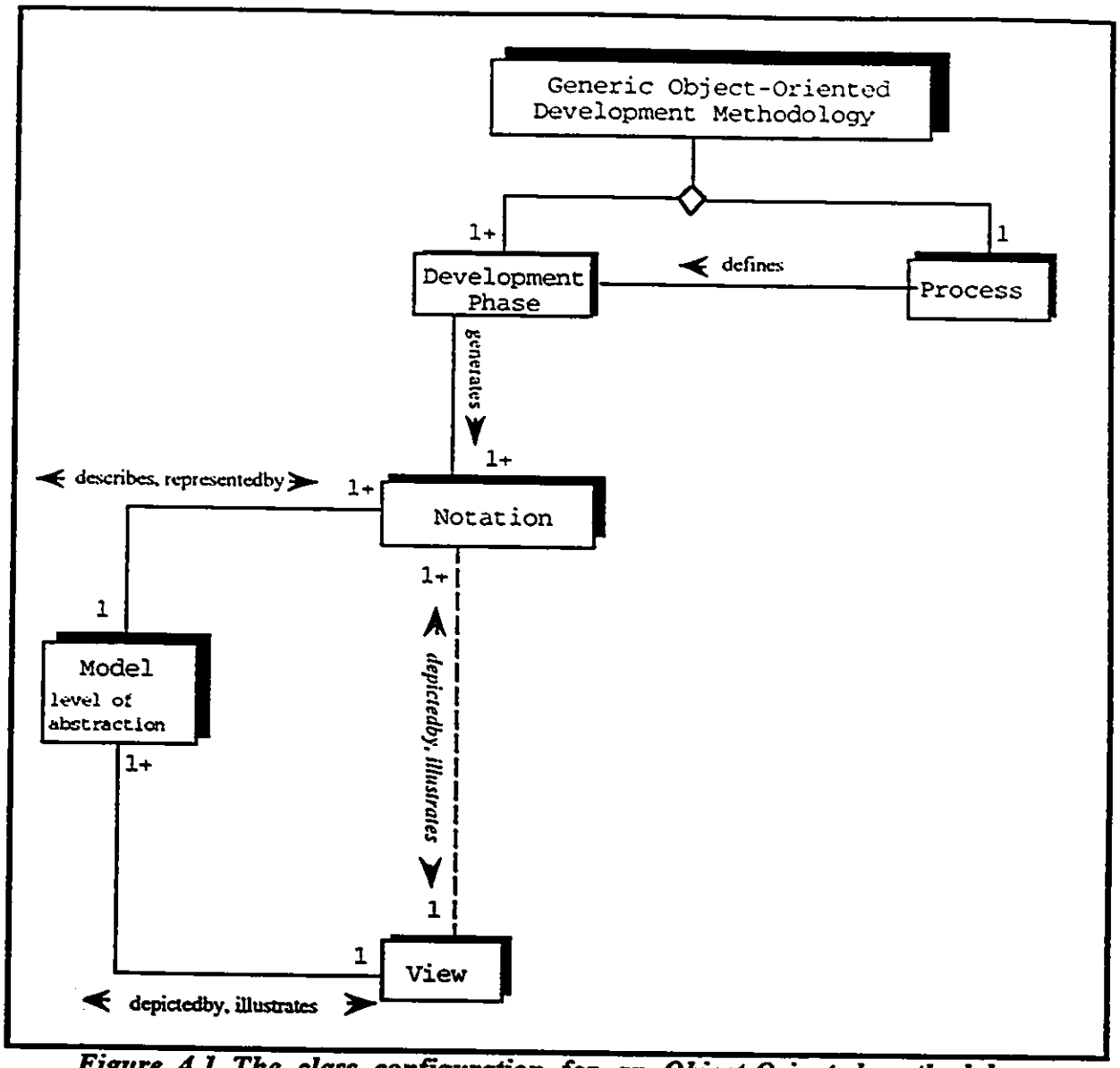


Figure 4.1 The class configuration for an Object-Oriented methodology

Notations: As described in section 3.2.2.3.2, a `Notation` conforms to a `Syntax`, which generates a set of `Statements` which in turn make up a `Language` for the `Notation`. It is this `Language` that provides the semantic for the `Model`. A `Syntax` is formed by an `Alphabet` and a `Grammar`. An `Alphabet` contains a set of `Modeling Constructs`, each of which has a unique `Representation`. The `Grammar` contains `Rules` that define how to legally combine `Modeling Constructs`. Notations are described in section 4.3.2.

4.3 The Generic Object-Oriented Development Methodology

We will now describe a Generic Object-Oriented Development Methodology based on [14,15,18,22,24,25,26,27,28,29,30,31,32]. The discussion first describes the process and then Notations, Models and Views.

4.3.1 The Process

Most OO processes define an Analysis, Design and Implementation phases of software development. Usually, Requirements Modeling is part of the Analysis phase as in the case in [18].

Three observations can be made about different OO methodologies; (1) the two ends of software development (i.e. Requirements Capture and Testing) are omitted most of the time; (2) "the boundaries between Analysis and Design are inconsistently defined" [35]; (3) the processes do not accommodate changes in requirements, analysis and design models, well. Note that the latter is one of the major advantages of a Spiral style software development [10], however the current methodologies do not provide consistent mechanisms for dealing with changes. The usual tendency in software engineering is to accommodated changes at the phase where they are discovered. For example, if a change to Requirements notations is identified at Design, the usual tendency is to modify the Design, update Requirements and then continue development from Design on. This results in *spaghetti development* (Design –> Requirements –> Design) which in turn causes the loss of traceability of notations for a given iteration and from one iteration to the next.

Observation (1) is addressed in Chapter 8. Observation (2) is addressed in section 4.3.1.1. Observation (3) is discussed in section 4.3.1.2.2.

4.3.1.1 The Development Phases

The development phases aim at describing different levels of abstraction of the application to be developed. As part of our work, we define the following levels of abstraction.

The *Requirements* level of abstraction describes the requirements of the application to be developed.

The *System* level of abstraction describes the domains, subsystems and problem domain classes/objects as well as their relationships of the application to be developed.

The *Class/Object* level of abstraction describes the external view of classes and/or objects (i.e. class structures, their interfaces and their communications).

The *Class/Object Internal* level of abstraction describes the internal behavior and data of objects.

We define the following development phases:

Requirements Capture

The Requirements Capture phase consists in gathering all requirements of the system to be developed. The requirements are usually collected in a specification document written in a natural language form. An alternative or complementary approach is to capture requirements in the form of Scenarios [14] or Use Case [18] specifying uses of the system. The Requirements Capture phase provides the *Requirements* level of abstraction of the application to be developed.

Requirements Modeling

The Requirements Modeling phase models the problem domain: "The Requirements Modeling phase aims at delimiting the system and defining what functionality the system should offer" [18]. The different activities that usually take place include:

(1) identification of different Domains[7], (e.g. "application, service, architectural, implementation", [24]) and the relationships between Domains (e.g. bridges in [24]);

(2) identification of Subsystems in Domains [14,18,22,24] and their relationships if necessary

(3) identification of Domain Classes/Objects in each Subsystem from the Uses Cases or Scenarios in the Requirements Capture phase

The Requirements Modeling phase provides the *System* level of abstraction of the application to be developed.

Analysis

The Analysis phase also models the problem domain [35]. The system is specified in terms of Semantic Classes/Objects ("things or concepts used in describing the problem, rather than the solution" [35]) and their relationships and interactions. These Semantic Classes/Objects are a refinement of Domain Classes/Objects of the Requirements Modeling phase.

The Semantic Classes/Objects produced at Analysis are described at two levels of abstraction: the *Class/Object* and *Class/Object Internal* level.

The Analysis models and notations should capture different Views of the system (e.g. Static, Communication, Dynamic and Processing, describe in section 4.3.4).

---

[7] "A Domain can be thought of as a separate world inhabited by its own conceptual entities, or objects. Hence, in an Automated Railroad Management System, a Railroad Operations Domain is concerned with trains, tracks and the like, while a User Interface Domain is involved with windows, displays, and icons" [24]. For more details, please refer to Appendix B.

Design

The design phase models the solution domain [35]. The Analysis models and notations are refined, elaborated and optimized (e.g. "semantic objects, may be extended as useful abstractions are discovered" [35], solution based objects are identified such as Interface, Application and Base/Utility classes [35], etc.).

The Solution Classes/Objects produced at Design are described at two levels of abstraction: the *Class/Object* and *Class/Object Internal* level.

Design should continue providing the different Views of the system as identified in Analysis.

Thus the boundaries between Analysis and Design are the domain that each phase deals with: Analysis deals with the problem domain in terms of Semantic objects; Design deals with the solution domain in terms of Solution objects.

Implementation

The Implementation phase corresponds to the translation of Design models and notations into programming languages for execution.

Testing

The Testing phase aims at verifying that the implementation fulfills the requirements of the original specification.

4.3.1.2 The Style of Development

Having define the development phases, we will now define the process (i.e. iteration style and ordering of development phases).

4.3.1.2.1 Background

Most methodologies propose an iterative incremental approach to OO software development based on the Spiral process [10].

As stated in [37], different authors propose different iteration styles. Fig. 4.2 illustrates the difference between OMT [14] and Jacobson [18], Shlaer and Mellor [24], and Booch [22].

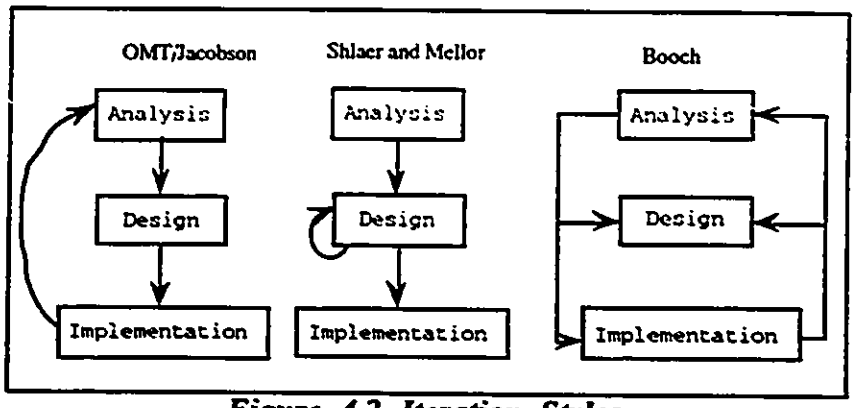


Figure 4.2 Iteration Styles

Other industry methodologies as the one used in the project described in [17], do not define the iteration style, it is up to the designer to determine "how much to do". As shown in Fig. 4.3, 'Iteration 1' includes extensive Analysis, some Design and no Implementation. 'Iteration 2' concludes Analysis, performs some Design and starts Implementation. 'Iteration 3' concludes Design and does mainly implementation.

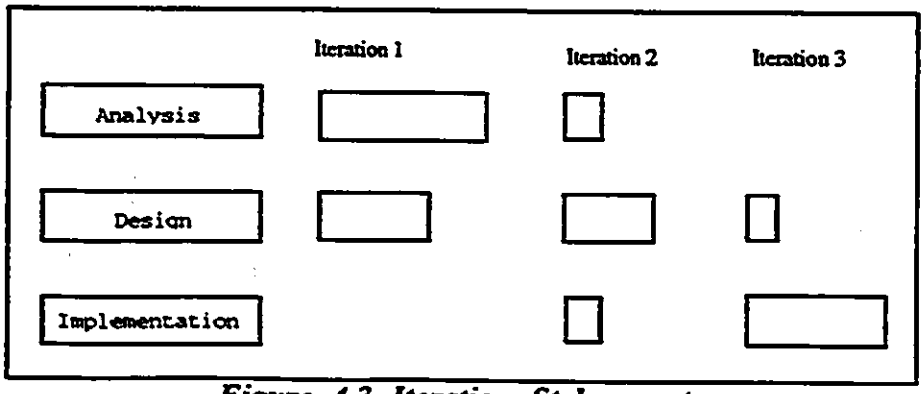


Figure 4.3 Iteration Styles cont.

4.3.1.2.2 How to handle change

None of the iteration styles described above accommodates changes (e.g. error corrections) in analysis notations, design notations, etc., such that traceability is not lost. We will illustrate our statement with examples and provide solutions. The solutions proposed can be easily implemented in the iteration styles.

Note that our proposed way of dealing with change does not address legacy code. Also, it assumes that the implementor is in close interaction with the analyst(s) and designer(s), or is the analyst and designer himself or herself. Note that this is often the case within a Micro-Iteration (define in section 4.3.1.3) but not within a Macro-Iteration.

Also it is not within the scope of this thesis to address management issues such as a code change not being accepted by a analyst and therefore delaying the project.

How to handle change in the OMT/Jacobson style of iteration

Lets take the iteration style of OMT/Jacobson in Fig. 4.2. We observe that there is no feedback from Design to Analysis and from Implementation to Design. However changes to Design notations will certainly cause changes to Analysis notations and in turn changes to Implementation will cause changes to Design notations. How can we ensure that these changes are addressed and that traceability of notations within an iteration and from one iteration to the next is always present? Fig. 4.4 illustrates our proposed solution.

The basic resource to provide a consistent and organized manner of dealing with change is a repository. In Fig. 4.4, there are two partitions to the repository.

Partition (1) stores changes to the current iteration Analysis notations discovered at the Implementation phase of the current iteration. This is shown by the line labeled as 2.

Partition (2) stores changes to previous iterations Analysis, Design and Implementation notations discovered at any of the three phases of the current iteration. This is shown by the lines labeled as 4, 5 and 6, respectively.

One iteration may be composed of one or more *cycle of changes* (one *cycle of change* corresponds to the path labeled as 8, 9, 10). The exit criterion for an iteration is when the goal of the iteration is fulfill and when partition (1) is empty. At this point the next iteration may begin if and only if partition (2) is empty. If this is not the case, then the changes in partition (2) must be addressed, as shown by the line labeled as 4. This will cause previous iterations to be revisited (every change in partition (2) should be moved to partition (1), which in turn will start a *cycle of change*). Thus, the process forces to address changes before moving forward, therefore avoiding to build on top of components that have errors.

At Analysis, the first activity is to address the changes in partition (1) as shown by the line labeled as 7. If the partition is empty then, if partition (2) is empty then we can start a new iteration, otherwise, the iteration in partition (2) requiring changes will become the next iteration. If partition (1) is not empty, then this means that the current iteration needs to go through a cycle of change. This will cause changes to the current Analysis notations. Once the changes in partition (1) have been addressed, the partition should be empty. The Design notations can now be updated based on the changes performed to Analysis, as shown by the line labeled as 8.

At Design, changes to Analysis notations may be discovered: they should be applied immediately as indicated by the line labeled as 1.

The Implementation phase can now be updated based on the changes performed to the Design notations as shown by the line labeled as 9. Changes to Analysis should be stored in partition (1), as indicated by the line labeled as 2. The same thing should occur for changes to Design that will affect Analysis. This is because, if they are applied to Design right away, nothing guarantees that later, the Analysis notations will be updated accordingly. In fact these changes to Analysis have a high probability of being forgotten. This will cause loss of traceability between Analysis and Design notations for the current iteration. Finally, changes to Design that will not affect Analysis can be directly applied to Design as indicated by the line labeled as 3. This may in turn cause changes to Implementation as indicated by the lines labeled as 9.

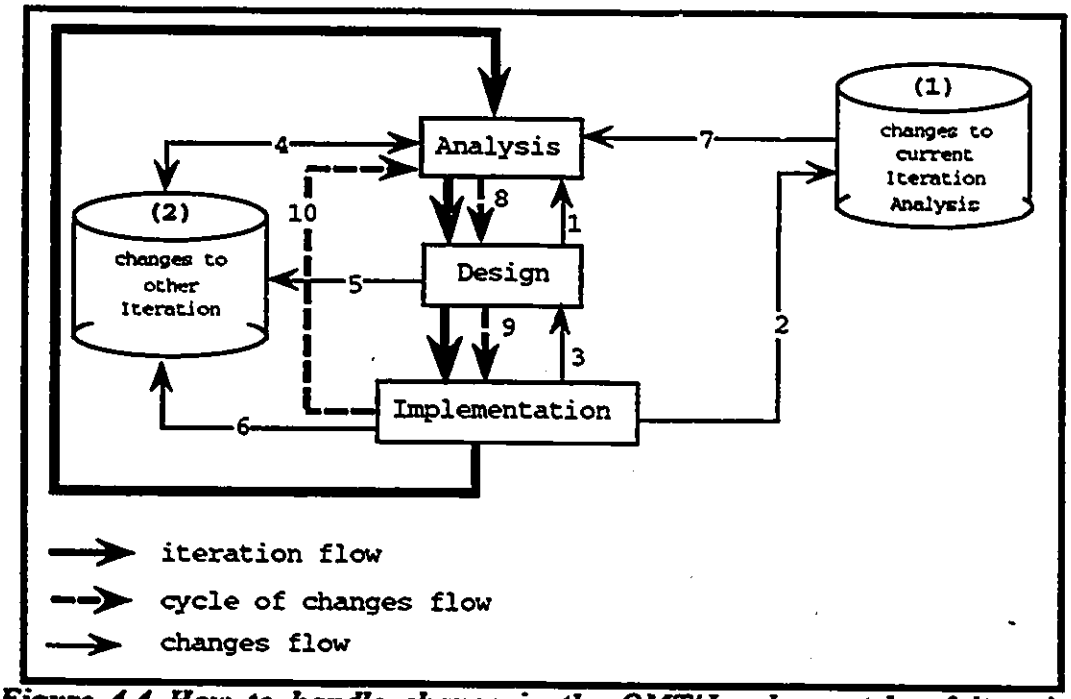


Figure 4.4 How to handle change in the OMT/Jacobson style of iteration

Within an iteration, we observe that changes bubble-up. For example, if at Implementation, it is determined that changes to both Design and Analysis notation will be required, then the changes will be store in partition `(1)` to be addressed as part of the next *cycle of change* .

Finally, note that forward changes (e.g. it is determined at Analysis that changes to Implementation will be required) do not need to be stored in a partition since they are handled by the current flow of cycle change (i.e. path labeled as 8, 9).

With the process described above, we can ensure Vertical Upwards and Vertical Downward traceability (described in section 3.2.2.3.1) within an iteration. This is because changes that involve consecutive phases are applied immediately to both phases. Changes that involve phases that are not consecutive are not applied directly but rather are moved up (bubble-up) to be addressed as part of a *cycle of change.*

We can ensure Forward and Backward Iteration traceability (described in section 3.2.2.3.1) in the following manner: before moving to the next iteration, the process forces to address partition `(2)`, thus not allowing to move forward until all errors have been corrected and notations updated.

How to handle change in the Shlaer and Mellor style of iteration

Lets take the iteration style of Shlaer and Mellor in Fig. 4.2. The iteration style is based on recursive Design until completed and error-free. We observe that the process does not include iteration of Analysis, therefore (a) no traceability can be achieved between notations (i.e. changes to Design and Implementation cannot be reflected in Analysis) and (b) all components (e.g. semantic classes, functionality, etc.) not identified in Analysis will lead to incomplete implementations (the implementations must be upgraded to include the new Analysis components through patching, causing expensive rework). Note that this is a common problem of the Waterfall style of iteration.

How can we ensure traceability of notations? Fig. 4.5 illustrates our proposed solution. The basic requirement of the iteration style is that to move to Implementation, Design must be completed and error-free. This means that Analysis and Design notations must have Vertical Upwards and Vertical Downwards traceability before moving to Implementation. This involves updating Analysis notations, while performing Design as illustrated by the line labeled as 1.

At Implementation, if changes to Analysis and/or Design notations come across, they should be stored in partitions (1) and (2) respectively as shown by lines labeled as 2 and 3.

Once Implementation is finished, if partitions (1) and (2) are empty then the process is finished (i.e. the application has been developed) and no *cycle of change* was required. Otherwise a *cycle of change* is started as indicated by the line labeled as 8.

The *cycle of change* starts with Analysis. The changes in partition (1) must be addressed as indicated by the line labeled as 5. This will cause changes to Design as indicated by the line labeled as 6. Once the changes in partition (1) have been addressed, the partition should be empty. Now, Design notations must be updated with the changes in partition (2) as indicated by the line labeled as 4. This may cause changes to Analysis as indicated by the line labeled as 1. Once the changes in partition (2) have been addressed, the partition should be empty. Now, the Implementation can be updated as indicated by the line labeled as 7. Again, this may cause changes to Analysis and Design notations. These should be stored in partitions (1) and (2) respectively as shown by lines labeled as 2 and 3, causing a new *cycle of change* to begin. This process is repeated until no changes are store in partitions (1) and (2) at the Implementation phase.

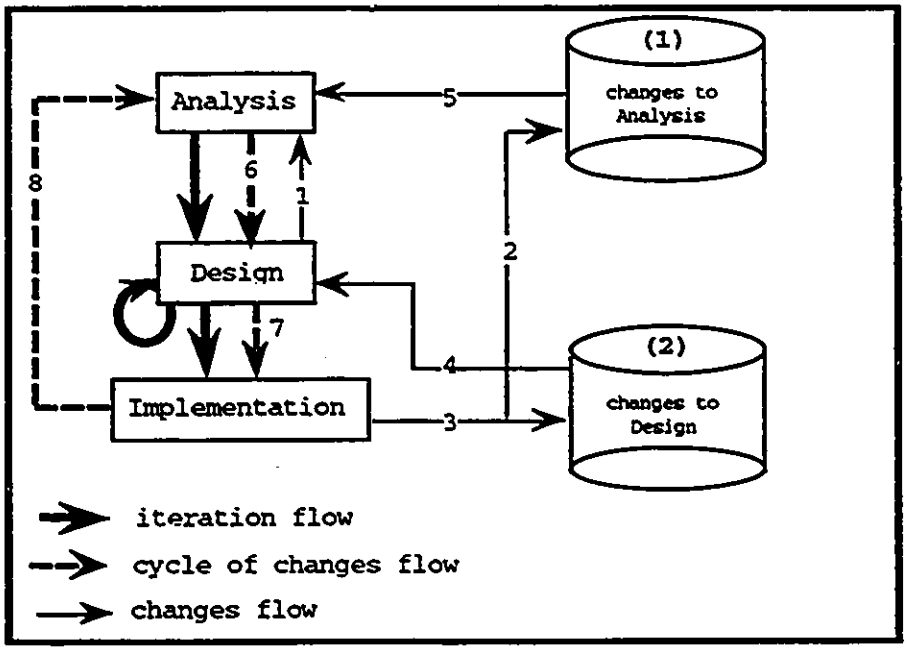


Figure 4.5 How to handle change in the Shlaer and Mellor style of iteration

In our process a *cycle of change* can start only after Implementation is completed. This is in line with the Shlaer and Mellor iteration style. The process allows to handle changes in a consistent manner, after implementation, to achieve Vertical Upwards and Vertical Downward traceability. This is achieved by storing changes in partition `(1)` and `(2)` and addressing them as part of a next *cycle of change*.

Vertical Upwards and Vertical Downward traceability between Analysis and Design notations while in the recursive Design stage is achieved by updating Analysis notations during recursion.

How to handle change in the Booch style of iteration

Lets take the iteration style of Booch in Fig. 4.2. In this iteration style, it is possible to go from Analysis to Implementation, thus jumping over the Design phase. This provides flexibility. However, Design will not be consistent with Analysis for a specific iteration, for both development and documentation, since an Implementation will have no Design that corresponds to it. In this case, the traceability flow is lost. How can we ensure traceability of notations, given that phases can be skipped? Fig. 4.6 illustrates our proposed solution.

At every phase, changes to notations of all other phases should be stored in partitions, if and only if, the next phase in the iteration cannot address the changes.

For example, at Analysis, changes to Design notations should be stored in partition `(2)` as indicated by the line labeled as `1`, if and only if, the next phase of development is Implementation. If the Design changes are not stored in partition `(2)`, nothing guarantees that these changes will be addressed in the near future. This is because, the iteration style allows to iterate from Analysis to Implementation, back to Analysis repetitively, thus increasing the probability of forgetting the Design changes.

Another example to illustrate our logic, is that at Implementation, changes to Analysis notations should be stored in partition `(1)` as indicated by the line labeled as `2`, if and only if, the next phase of development is Design. If the Analysis changes are not stored in partition `(1)`, nothing guarantees that these changes will be addressed in the near future. This is because, the iteration style allows to iterate from Implementation to Design, back to Implementation repetitively, thus increasing the probability of forgetting the Analysis changes.

Once at a phase, the first activity should be to address the changes in the partition corresponding to that phase. For example, at Analysis, changes in partition `(1)` should be addressed as indicated by the line labeled as 3. Once the changes in partition `(1)` have been addressed (i.e. Analysis notations have been updated), the partition should be empty. During this process, (a) Design changes may come across: they should be stored in partition `(2)` if the next phase of development is Implementation or (b) Implementation changes may be come across: they should be stored in partition `(3)` if the next phase of development is Design.

The exit criterion of the application to be developed is at Implementation, when the system has been developed and partitions `(1)`, `(2)` and `(3)` are empty.

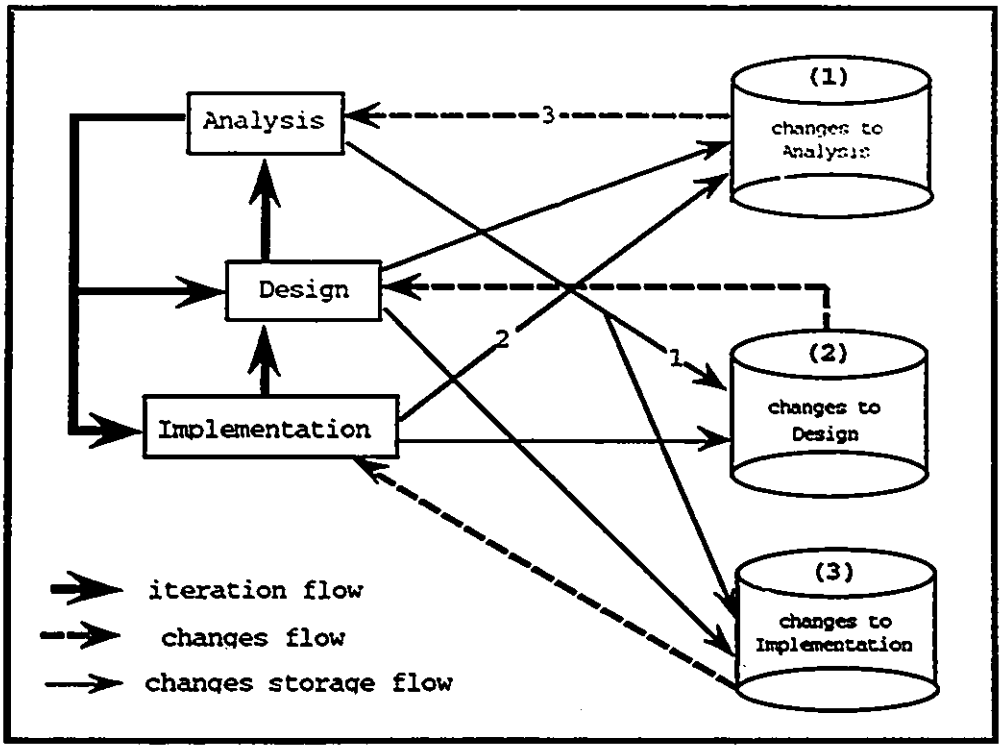


Figure 4.6 How to handle change in the Booch style of iteration

The process allows to handle changes in a consistent manner, regardless of the sequence of phases. This is in line with the Booch iteration style. At every phase, the process forces to address changes specific to the notations of that phase, thus not allowing to move forward (i.e. next phase) until all errors have been corrected and notations updated.

Vertical Upwards and Vertical Downward traceability between phases will be achieved at the exit criteria described above.

How to handle change in an industry iteration style

Lets take the iteration style in Fig. 4.3. This iteration style is probably the most flexible one, since it allows designers to set up their own iteration style. The problem however is similar to the one in the Booch's approach: one can easily loose traceability from one phase to another.

How can we ensure traceability of notations? Fig. 4.7 illustrates our proposed solution. Note that the figure illustrates one possible configuration, on the basis of 'Iteration 1' including extensive Analysis, some Design and no Implementation, 'Iteration 2' concluding Analysis, performing some Design and starting Implementation and 'Iteration 3' concluding Design and performing mainly implementation.

In 'Iteration 1', changes to Analysis notations, discovered at Design, should be stored in partition `(1)` as indicated by the line labeled as 1. At the next iteration, these changes should addressed at the corresponding phase (i.e. Analysis in our case, as indicated by the line labeled as 2). The partition should then be emptied.

In 'Iteration 2', the same process should be followed for changes to Analysis notations, discovered at Design as indicated by the line labeled as 3. At Implementation, changes to Analysis notations should be stored in partition `(1)` as indicated by the line labeled as 4; changes to Design notations should be stored in partition `(2)` as indicated by the line labeled as 5. These changes should be addressed in the next iteration.

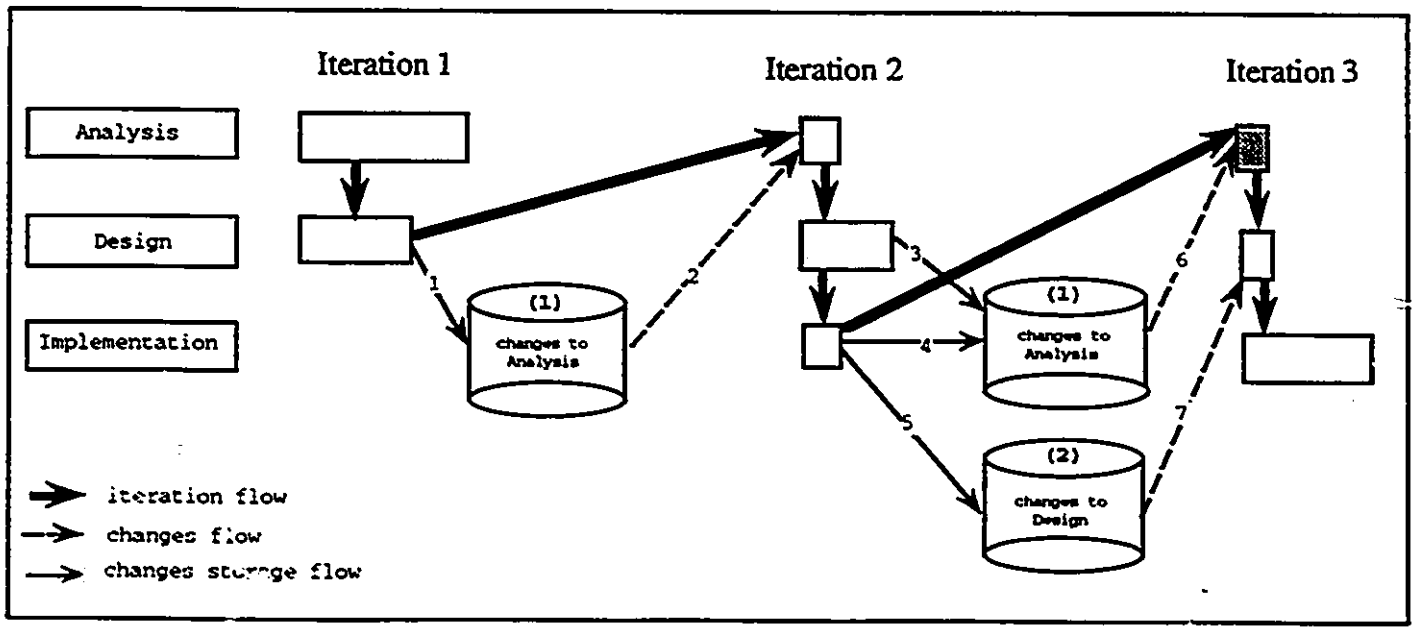


Figure 4.7 How to handle change in an industry style of iteration

In 'Iteration 3', the gray rectangle indicates that Analysis is not finished since partition `(1)` is not empty. These changes must be addressed, as indicated by the line labeled as 6, and Analysis must be revisited, even though the original plan scheduled Analysis to be finished at 'Iteration 2.' Once the Analysis notations have been updated, partition `(1)` should be emptied. At Design, changes in partition `(2)` should be addressed in the same manner.

The exit criterion of the application to be developed is at Implementation, when the system has been developed and partitions `(1)`, `(2)` and `(3)` are empty (i.e. no next iteration is required).

The process allows to handle changes in a consistent manner, regardless of the number of iterations. This is in line with the iteration style presented in Fig. 4.3. At every phase, the process forces to address changes specific to the notations of that phase, thus not allowing to move forward (i.e. next phase or next iteration) until all errors have been corrected and notations updated.

Vertical Upwards and Vertical Downward traceability between phases will be achieved at the exit criteria described above.

The previous section described iteration styles adopted in OO software development. We augmented every iteration style with a process to handle changes in a consistent manner to ensure that they will be addressed and will not brake traceability of notation through iterations and phases. This was one of the basic factors for achieving testability as described in section 3.2.2. We will now introduce the approach to software development used throughout this document.

4.3.1.3 The Generic Object-Oriented Development Process

Fig. 4.8a shows the approach to software development used throughout this document. This process will be referred to as the Generic Object-Oriented Development process.

The approach is based on the phases defined in section 4.3.1.1, on the iteration style of OMT/Jacobson described in section 4.3.1.2.1, and its corresponding way of handling change described in the same section. We have augmented the style of development to better suit large projects as the one described in case of [17].

Note that we could have selected any of the four iteration styles described in section 4.3.1.2.1, since for every one, we have proposed ways of handling change.

Our process allows iterative and incremental development. The phases of the development process are repeated (iterative) and the system is developed through successive refinements increasing in level of detail (incremental).

We define two types of iterations in our process: *Micro* and *Macro* iterations. These are similar to the Macroprocess and Microprocess steps in [39]. One Macro Iteration is composed of one or more Micro Iterations.

The first phase in our process is the Requirement Capture phase. Its intend is to capture the requirements of the system to be developed. Once a good understanding of the requirements is achieved, the Requirements Modeling phase may begin. Requirements modeling may be performed on the whole set of requirements or simply on a defined set (e.g. the set that will be used for the Macro-Iteration). This phase marks the beginning of a Macro-Iteration.

Macro-Iterations iterate over the requirements of the application to be developed. The goal of a Macro-Iteration is the selected set of Requirements Modeling that will be implemented during the Macro-Iteration. The partitioning of the system may be on a functionality basis (e.g. a domain, a subsystem), on a risk basis (e.g. items of higher risk are implemented first), on the basis of one or more scenarios from Requirements Capture, etc.

Micro-Iterations iterate over the subset of requirements selected in a specific Macro Iteration. The focus of Micro iterations should be the implementation of the Macro-Iteration goal (e.g. one subsystem, a risk item, etc.). Again, as in the case of Macro-Iterations, the Micro-Iteration could be over a particular interaction among subsystems (in the case that more then one subsystem was selected as the basis for the Macro-Iteration), over items of higher risk within the subset of requirements, etc.

The approach in Fig. 4.8a is both Top-Down and Bottom-Up.

Top-Down because of the refinements that occur at each phase: Domains, Subsystems and Domain Classes/Objects are derived from Scenarios; the latter are refined into Semantic objects, with semantic operations and relations between these Semantic objects; Semantic objects are refined into Solution objects, with their corresponding operations and relations.

Bottom-Up because each iteration (Macro and Micro) provides increasing functionality.

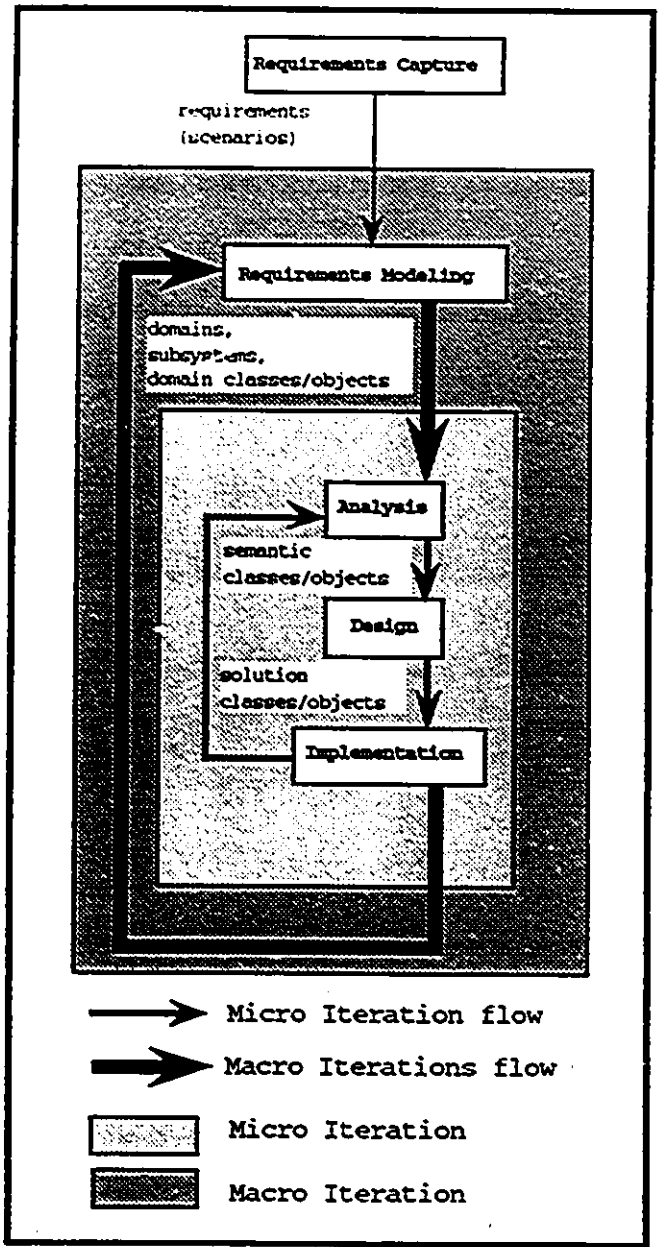


Figure 4.8a The Generic Object-Oriented Development Process

Fig. 4.8b illustrates our proposed way of handling change in the Generic Object-Oriented Development Process. Note that the figure is a more detailed version of Fig. 4.4.

The focus of the process is to provide a way of handling change in a consistent and organized manner. The intention is to provide traceability of notations:

(1) within a Macro/Micro iteration,
(2) from a Macro/Micro iteration to the next and
(3) from a Macro/Micro iteration to its previous iteration;

given that at any phase, changes to notations of any other phase may be identified.

Note that Fig. 4.8b does not illustrate the flow of development but rather the flow of changes and corrections to notations. The flow of development was illustrated in Fig. 4.8.a.

Fig. 4.8b appears to be very complex, but in reality, its semantics is quite simple. The process is based on the following:

(a) at any phase, changes to any other phase can be identified; these changes are stored in partitions of a database
(b) before starting a new phase, these partitions (i.e. changes) must be analyzed

The basic resource to provide a consistent and organized manner of dealing with change is a repository divided in 4 partitions depicted as `(1)` to `(4)`.

Partition `(1)`: stores changes to notations (i.e. specification document) of the Requirements Capture phase.
Partition `(2)`: stores changes to notations of the current Macro-Iteration.
Partition `(3)`: stores changes to notations of the current Micro-Iteration Analysis phase.
Partition `(4)`: stores changes to notations of previous Macro-Iterations.

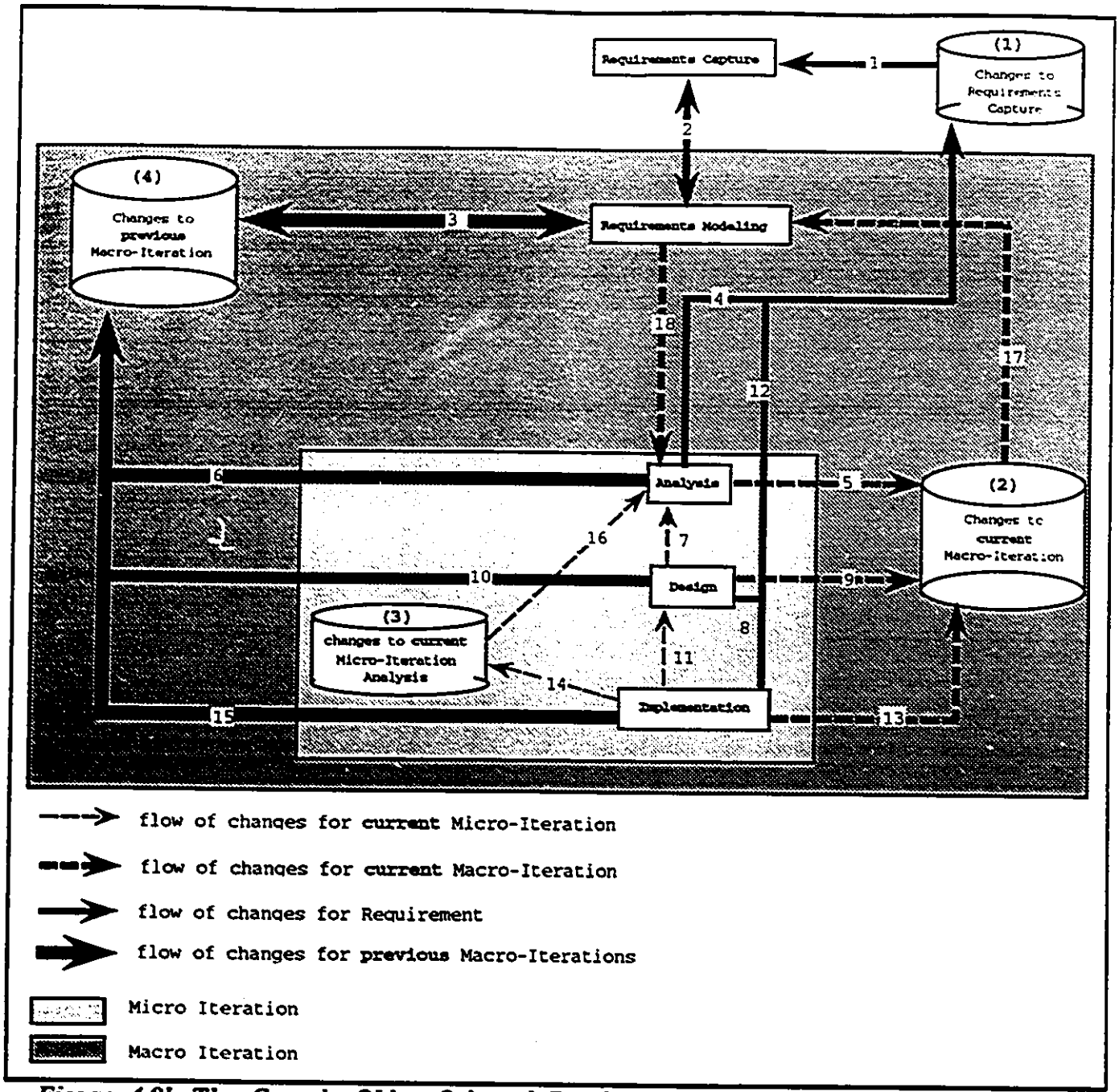


Figure 4.8b The Generic Object-Oriented Development Process to accommodate changes

We now provide a high level algorithm explaining Fig. 4.8b. In the discussion that follows, assume that Requirements Capture has occurred and that we are currently in the Requirements Modeling phase.

1. repeat until all partition are empty and no new Macro-Iteration remain

2. address changes in partition (1) *this is indicated by line 1*
   if empty
      - go to 3
   else
      repeat until empty
         apply changes to Requirements Modeling notations, in turn apply changes to Requirements Capture notations as shown by line 2.

3. address changes in partition (4) *this is indicated by line 3*
   if empty *next Macro-Iteration is a new Macro-Iteration*
      - select Micro-Iteration *next Micro-Iteration is a new Micro-Iteration*
      - perform Analysis, store changes as indicated by lines 4, 5, and 6.
      - perform Design, store changes as indicated by lines 7, 8, 9 and 10.
      - perform Implementation, store changes as indicated by lines 11, 12, 13, 14 and 15.
      - go to 4
   else
      repeat until (4) is empty *next Macro-Iteration is an existing Macro-Iteration*
         - select Macro-Iteration that requires change
         - move changes in (4) corresponding to the selected Macro-Iteration to partition (2)
         - remove these changes from (4)
         - go to 5

4. address changes in partition (3) *this is indicated by line 16*
   if empty
      - go to 5
   else
      repeat until (3) is empty
         - modify Analysis, store changes as indicated by lines 4, 5, and 6,
         - remove change from (3),
         - modify Design, store changes as indicated by lines 7, 8, 9, and 10.
         - modify Implementation, stores change as indicated by lines 11, 12, 13, 14 and 15
         - go to 5

5. address changes in partition (2) *this is indicated by line 17*
   if empty
      go to 1
   else
      repeat until (2) is empty
         - update Requirements Modeling notations *this is indicated by line 17*
         - select Micro-Iteration of current Macro-Iteration that requires change,
         - modify Analysis, store changes as indicated by lines 4, 5, and 6,
         - remove change from (2),
         - modify Design, store changes as indicated by lines 7, 8, 9, and 10.
         - modify Implementation, stores change as indicated by lines 11, 12, 13, 14 and 15
         - go to 4.

The following observations can be made:

1) one Micro-Iteration can be composed of one or more *cycles of change* (i.e. until partition `(3)` is empty),

2) one Macro-Iteration can be composed of one or more *cycles of change* (i.e. until partition `(2)` is empty),

3) the exit criteria for a Micro-Iteration is the goal of the Micro-Iteration and for partition `(3)` to be empty, at the Implementation phase.

4) the exit criteria for a Macro-Iteration is the goal of the Macro-Iteration and for partition `(2)` to be empty, at the Implementation phase,

5) do not move forward (i.e. address a new Micro-Iteration until partition `(3)` is empty,

6) do not move forward (i.e. address a new Macro-Iteration until partition `(4)` is empty,

7) changes in partition `(2)` will cause revisiting notations of previous Micro-Iterations for the current Macro-Iteration,

8) changes in partition `(4)`, will cause revisiting notations of previous Macro-Iteration as well as their Micro-Iterations

9) within a Micro-Iteration,
   - changes to consecutive phase can be applied directly if and only if, it is determined that the changes affect only the two phases in question (this is shown by the lines labeled as `7` and `11`.
   - changes to two non-consecutive phases (i.e. Analysis and Implementation) should be stored in partition `(3)` and be addressed as part of the next *cycle of change*, as explained in Fig. 4.4, these changes are said to *bubble-up*.

Lets take two examples to illustrate the discussion.

Example of change: 1

Assume that while in the Implementation phase, we detect changes in Design that will cause Analysis changes; if one performs those changes in the Design notations and implements them, then the Analysis notations will be inconsistent with the Design notations for that Micro-Iteration. This is because nothing guarantees that the Analysis notations will be updated later. Therefore, these changes should be stored in partition `(3)` to be addressed as part of the next *cycle of change* for the current Micro-Iteration. Note that, if it is determined that the change identified at Implementation does not affect any other phase but Design, then it can be applied immediately to Design notations.

Example of change : 2

Assume that while in the Analysis phase of a given Micro-Iteration, we detect changes that must be incorporated to the Implementation phase of a previous Micro-Iteration of the current Macro-Iteration; if one performs those changes in the Implementation, then we loose traceability within Micro-Iterations for the current Macro-Iteration. The changes should be stored in partition `(2)` (this is shown by the lines labeled as 5), and be addressed when the current Micro-Iteration is finished and we are in the process of moving to the next Micro-Iteration.

Example of change : 3

Assume that while in the Design phase of a given Micro-Iteration, we detect changes that must be incorporated to the Implementation phase of a previous Macro-Iteration; if one performs those changes in the Implementation, then we loose traceability within the current Micro-Iteration and the Macro-Iteration that requires the change. The changes should be stored in partition `(4)` (this is shown by the lines labeled as `10`) and be addressed when the current Macro-Iteration is finished and we are in the process of moving to the next Macro-Iteration.

The Generic Object-Oriented Development process provides all the traceability types described in section 3.2.2.3.1. As said earlier, the focus of the process is to provide a way of handling change in a consistent and organized manner. The intention is to provide traceability of notations:

(1) within a Macro/Micro iteration,
(2) from a Macro/Micro iteration to the next and
(3) from a Macro/Micro iteration to its previous iteration;

As described in section 4.2, the Generic Object-Oriented Development Methodology corresponds to a set of artifacts (Process, Notations and Models) used in the development of OO software. Having described the Process, our next activity is to describe Notations and Models.

4.3.2 The Notations

The purpose of this section is to provide a brief summary of commonly used notation (Table 4.1) in the OO world. Note that the list is not at all exhaustive. In some case notations or diagrams are omitted since they are language specific (e.g. Module Diagrams in [22]). The notations will then be classified into Models (in section 4.3.3) and Views (4.3.4).

The rational for our classification of Notations into Models and Views is from a testing perspective. Notations are used to describe the semantics of the application to be developed at different levels of abstraction (i.e. Requirements, System, Class/Object or Class/Object Internal); testing should be conducted at these different levels of abstraction to ensure that the semantics (Views) of the application are complete, consistent and correct. However, to avoid proliferation of testing activities, they should not be dependent on notations. They should be generic to a group of notations (i.e. View) and define (1) what needs to be tested, (2) coverage criteria, and (3) should assess for completeness, consistency and correctness of notations within the View. This corresponds to a *testing activity type*. A *testing activity* can be thought of as a specialization of a *testing activity type* for a specific notation in a View. This is explained in Chapter 5.

All notations accompanied by an * are described in more detail in Appendix B.

Notation Name	Notation Reference	Notation Description
* Use Case model	Jacobson [18]	Describes a specific way of using the system.
* Domain Object model	Jacobson [18]	Shows the relations (class and instance associations) between domain objects.
* Analysis model	Jacobson [18]	Shows the relations (class and instance associations) between analysis objects (Entity, Interface and Control objects). It also shows the distribution of objects into subsystems.
* Design model	Jacobson [18]	Defines the architecture of the system in terms of Blocks (analysis objects) and their relations (class and instance associations).
* Interaction Diagram	Jacobson [18]	Shows how objects (blocks) communicate in terms of stimulus (operations) sent and received.
State Transition Graph	Jacobson [18]	Describes an object's internal behavior in terms of states, events and actions.
SDL	Jacobson [18],	Describes an object's internal behavior in terms of states and events (processes, inputs and outputs).

Table 4.1 Overview of Notations used in the Object-Oriented world

Notation Name	Notation Reference	Notation Description
Class Diagram	Rumbaugh [14]	Schema, pattern or template for describing many possible instances of data (classes). It describes classes, their associations, cardinality of associations, inheritance, aggregation and attributes of classes.
Instance Diagram	Rumbaugh [14]	Describes how a particular set of objects (instances of classes) relate to each other.
Scenario	Rumbaugh [14]	Describes a sequence of events that occurs during one particular execution of the system. It is in the form of an enumeration of sequences of events.
Event Trace	Rumbaugh [14]	Shows how domain objects communicate in terms of events. Similar to an Interaction Diagram in [18], but does not depict the internals of an operation of an object.
State Diagram	Rumbaugh [14]	Describes an object's internal behavior in terms of states, events and actions. Similar to a State Transition Graph in [18].
* Object Interaction Diagram	Rumbaugh [39]	Describes the message flow between objects. It corresponds to an Instance Diagram with message flow between instances. Messages correspond to operations invoked by an object and data flows as message arguments and return values.
* Data Flow Diagram	Rumbaugh [14, 39]	Describes the effects of operations of objects. The diagram is extracted from the Object Interaction Diagram.
Class Diagram	Booch [22]	Shows the existence of classes and their relationships in the logical aspects of a system.
Object Diagram	Booch [22]	Shows the existence of objects and their relationships in the logical aspects of a system.
State Transition Diagram	Booch [22]	Describes an object's internal behavior in terms of states, events and actions. Similar to a State Transition Graph in [18].
Interaction Diagram	Booch [22]	Shows how objects communicate in terms of operations sent and received. Similar to an Interaction Diagram in [18].
* Domain Chart	Shlaer-Mellor [24]	Shows different domains of the application to be developed and usage among them.

Table 4.1 Overview of Notations used in the Object-Oriented world (cont.)

Notation Name	Notation Reference	Notation Description
* Information Model	Shlaer-Mellor [24]	Shows the information structure (i.e. objects[8], their attribute descriptions and relationships to other objects).
* Subsystem Relationship Model	Shlaer-Mellor [24]	Shows the relationships between subsystems in Domains.
State Transition Diagram	Shlaer-Mellor [24]	Describes an object[9]'s internal behavior in terms of states, events and actions. Similar to a State Transition Graph in [18].
State Transition Table	Shlaer-Mellor [24]	Another aspect of the State Transition Diagram but placing emphasis on transitions. A transition is an arrow from one state to another, labeled with the event that causes an object[10] to move from state to another. In a State Transition Table, columns represent *events*, rows represent *states* and cells represent the *next state* (or *event ignored* or *can't happen).* The State Transition Diagram and the State Transition Table form the State Model.
* Object Communication Model	Shlaer-Mellor [24]	Shows the asynchronous communication between objects[11] and external entities. Similar semantics to the Interaction Diagram notation in [18], but with no specific ordering of events.
Subsystem Communication Model	Shlaer-Mellor [24]	Shows the communication between subsystems. Similar to the Object Interaction Model, but instead of objects, subsystems are used.
* Object Access Model	Shlaer-Mellor [24]	Show the synchronous communication between objects[12].

Table 4.1 Overview of Notations used in the Object-Oriented world (cont.)

[8]In Shlaer-Mellor [24], at the level of Analysis, an *Object* corresponds to what we refer to as a class and an *instance of an Object* corresponds to what we refer to as an object. At Design the authors revert to our semantics (i.e. a Class corresponds to a class and an Object corresponds to an object).

[9]Same as footnote 8.

[10]Same as footnote 8.

[11]Same as footnote 8.

[12]Same as footnote 8.

Notation Name	Notation Reference	Notation Description
Subsystem Access Model	Shlaer-Mellor [24]	Show the synchronous communication between subsystems. Similar to the Object Access Model, but instead of objects, subsystems are used.
* Thread of Control Chart	Shlaer-Mellor [24]	Shows the sequence of actions and events that occurs in response to the arrival of an event when the system is in a particular state.
* Action Data Flow Diagram	Shlaer-Mellor [24]	Describes the units of processing within an action and the intercommunication between them.
Class Diagram	Shlaer-Mellor [24]	Describes the external aspects of a single class. Very detailed; it specifies class based and instance based operations with parameters types and values and logical components (data) with their type and meaning.
* Class Structure Chart	Shlaer-Mellor [24]	Describes the internal structure of a class as well as the flow of data and control within that class.
* Dependency Diagram	Shlaer-Mellor [24]	Describes the relationships between classes other then Inheritance.
* Inheritance Diagram	Shlaer-Mellor [24]	Describes the Inheritance relationships between classes.
* OAN Static	De Champeaux [15]	Capture the internal structure (e.g. attributes) and external structure (e.g. relationships) of objects.
* OAN Object Dynamic	De Champeaux [15]	Describes the internal behavior (states, events and transitions) of objects.
* OAN Object Interaction	De Champeaux [15]	Describes the interactions among objects.
* CRC Cards	Wirfs-Brock [31]	Describes the responsibilities and collaboration for a single class.
* Collaboration Graph	Wirfs-Brock [31]	Describes the collaboration between classes and subsystems.
Class Hierarchy	Wirfs-Brock [31]	Describes the Inheritance relationships between classes.
* ObjectCharts	Coleman [71] et all.	Describes the internal behavior of an object.
* Contracts	Helm and Holland [40] Wirfs-Brock [31]	Describes behavioral composition and the obligations on participating objects.

Table 4.1 Overview of Notations used in the Object-Oriented world (cont.)

4.3.3 The Models

We observe that different notations can be used to describe a specific level of abstraction (i.e. Requirements, System, Class/Object, Class/Object Internal, described in section 4.3.1.1) and within that level of abstraction, these different notations can be grouped based on the semantics they provide. This classification is called a Model. A Model is thus a classification of notations based on the level of abstraction and semantics they provide.

By examining Table 4.1, we define the Models as described in Table 4.2. The table shows the levels of abstraction addressed at different phase of the development process.

For each Model, we define its semantics and generic constructs common to the notations belonging to the Model, that are used to express the Model's semantics. Chapter 5 shows that these generic modeling constructs, based on Models and Views (described in the next section) are the artifacts to be tested by *testing activity types*.

4.3.3.1 Models describing the Requirements level of abstraction

The Requirements level of abstraction describes the requirements of the application to be developed in text format and/or by using notations in the Scenario Model. The Requirements Capture phase addresses this level of abstraction.

Scenario Model

Notations that are classified under the Scenario Model describe a specific use of the system by describing part of the system functionality.

Semantics provided by Model: Uses of the system.
Generic Constructs: - Actors (prospective users)
- Functions (of the system)
(e.g. `Customer generates daily report`
actor : `Customer`; function: `generate daily report`)
(e.g. `User lifts phone and gets dialtone`
actor : `User`; function: `lift phone, get dial tone`)
Notations: - `Use Case` notation in Jacobson [18]
- `Scenario` notation in Rumbaugh [14]

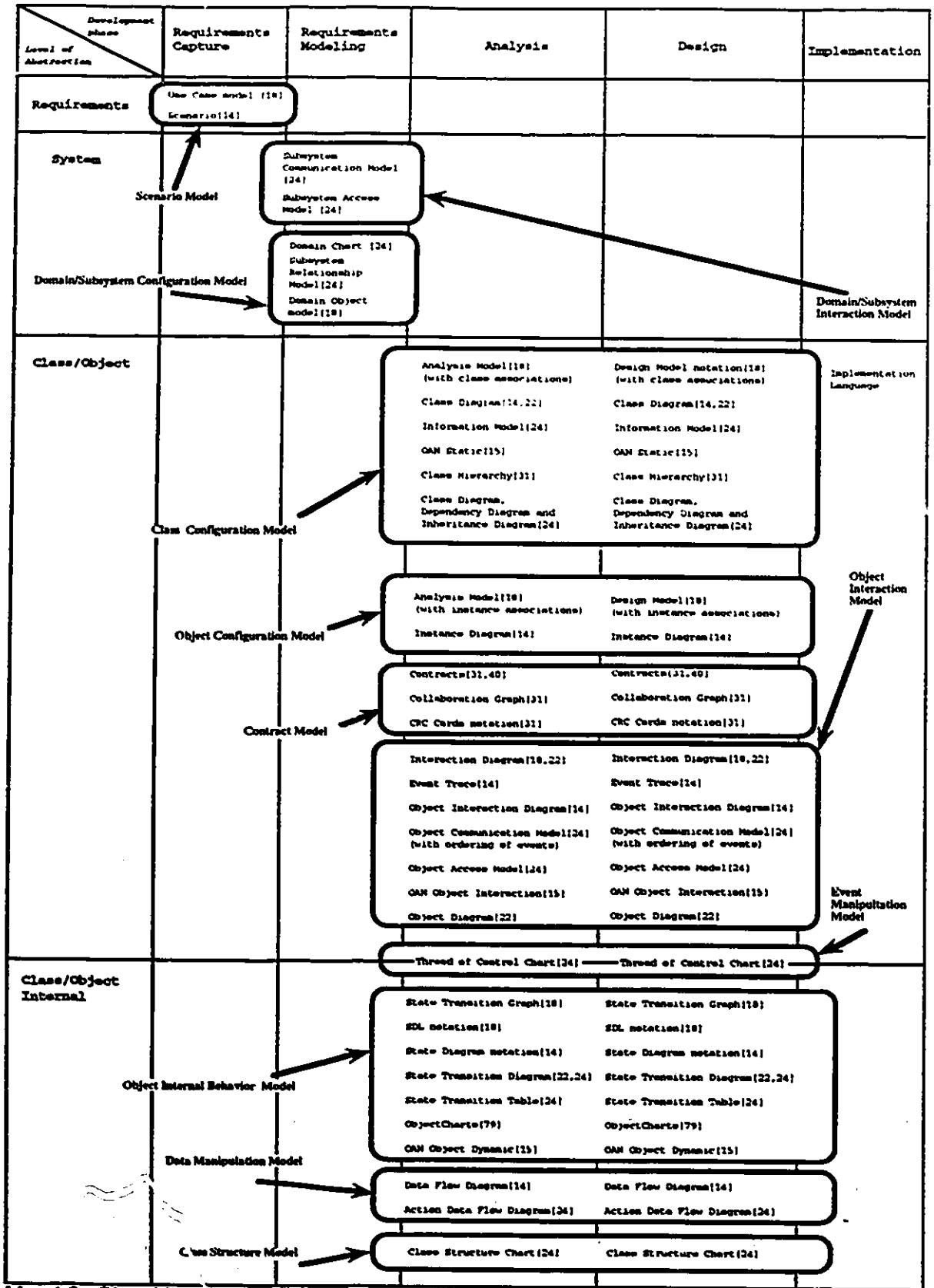


Table 4.2 Classification of Notations used in the Object-Oriented world into Models

4.3.3.2 Models describing the System level of abstraction

The System level of abstraction describes the Domains, Subsystems and Domain Classes/Objects of the application to be developed. Notations in the Domain/Subsystem Interaction Model describe the interactions between these entities and notations in the Domain/Subsystem Configuration Model describe their relationships. The Requirements Modeling phase addresses this level of abstraction.

Domain/Subsystem Interaction Model

Notations that are classified under the Domain/Subsystem Interaction Model describe the communication between Domains, Subsystems and Domain Classes/Objects.

Semantics provided by Model: Describes how Domains, Subsystems and Domain Classes/Objects communicate in terms of messages sent to and received from other Domains, Subsystems and Domain Classes/Objects. The description includes the message flow (including their ordering) between the entities, but does not depict their internal behavior. The message flow corresponds to event(s) that will eventually translate into operation(s) call(s) of object inside the Domains and/or Subsystems and/or Domain Classes/Objects.

Generic Constructs:
- Entities (Domains, Subsystems, Domain Classes/Objects).
- Messages (events between entities)
- Ordering of messages

Notations:
- `Subsystem Communication Model` notation in Shlaer-Mellor [24]
- `Subsystem Access Model` notation in Shlaer-Mellor [24]

Domain/Subsystem Configuration Model

Notations that are classified under the Domain/Subsystem Configuration Model describe the static configuration of Domains, Subsystems and Domain Classes/Objects.

Semantics provided by Model: Describes the relationships between and properties of Domains, Subsystems and Domain Classes/Objects.

Generic Constructs:
- Entities (Domains, Subsystems, Domain Classes/Objects).
- Relationships between entities
- Properties of entities (e.g. number of Subsystems in a Domain, an attribute of a Domain Class)
- Operations of Entities

Notations:
- `Domain Chart` notation in Shlaer-Mellor [24]
- `Subsystem Relationship Model` notation in Shlaer-Mellor [24]
- `Domain Object Model` notation in Jacobson [18]

4.3.3.3 Models describing the Class/Object level of abstraction

The Class/Object level of abstraction describes the external view of classes and/or objects (i.e. class structures, their interfaces and their communications).

The Analysis phase describes this level of abstraction for Semantic Classes and Objects of the application to be developed.

The Design phase also describes this level of abstraction for Solution Classes and Objects of the application to be developed.

Notations in the Class Configuration Model, Object Configuration Model, Contract Model and Object Interaction Model are used to describe this level of abstraction.

Class Configuration Model

Notations that are classified under the Class Configuration Model describe the static configuration of a set of classes.

Semantics provided by Model: Describes the relationships between and properties of Semantic Classes in the case of Analysis and Solution Classes in the case of Design.

Generic Constructs:
- Entities (Classes).
- Relationships between entities
- Properties of entities (e.g. an attribute of a Class)
- Operations of Entities

Notations:
- `Analysis Model` notation in Jacobson [18] (with class associations)
- `Design Model` notation in Jacobson [18] (with class associations)
- `Class Diagram` notation in Rumbaugh [14] and in Booch [22]
- `Information Model` in Shlaer-Mellor [24]
- `OAN Static` notation in De Champeaux [15]
- `Class Hierarchy` notation in Wirfs-Brock [31]
- `Class Diagram` and `Dependency Diagram` and `Inheritance Diagram` notations in Shlaer-Mellor [24]

Object Configuration Model

Notations that are classified under the Object Configuration Model describe the static configuration of a set of objects (instances of classes) at a given time. An Object Configuration Model corresponds to an instance of a Class Configuration Model.

Semantics provided by Model: Describes the relationships between and properties of Semantic Objects, in the case of Analysis and Solution Objects, in the case of Design at a time.

Generic Constructs:
- Entities (Objects).
- Relationships between entities
- Properties of entities (e.g. an attribute of an Object)
- Operation of Entities

Notations:
- `Analysis Model` notation in Jacobson [18] (with instance associations)
- `Design Model` notation in Jacobson [18] (with instance associations)
- `Instance Diagram` notation in Rumbaugh [14]

Contract Model

Notations that are classified under the Contract Model describe the expected collaboration between classes.

Semantics provided by Model: Describes the collaborations between Semantic Classes in the case of Analysis and Solution Classes in the case of Design.

Generic Constructs: - Entities (Classes).
- Contract

Notations: - `Contract` notation in Helm and Holland [40]
- `Collaboration Graph` notation in Wirfs-Brock [31]
- `CRC Card` notation in Wirfs-Brock [31]

Object Interaction Model

Notations that are classified under the Object Interaction Model describe the communication between objects.

Semantics provided by Model: Describes how objects (i.e. Semantic Objects in the case of Analysis and Solution Objects in the case of Design) communicate in terms of messages (operation invocations) sent to and received from other objects. The description includes the message flow (including their ordering) between objects but does not depict the internal behavior of an object.

Generic Constructs: - Entities (objects).
- Message or Event (corresponds to an operation invocation of an object)
- Ordering of messages

Notations: - `Interaction Diagram` notation in Jacobson [18] and in Booch [22]
- `Event Trace` notation in Rumbaugh [14]
- `Object Interaction Diagram` notation in Rumbaugh [14]
- `Object Communication Model` notation in Shlaer-Mellor [24] (with ordering of messages)
- `Object Access Model` notation in Shlaer-Mellor [24]
- `OAN Object Interaction` notation in De Champeaux [15]
- `Object Diagram` notation in Booch [22]

4.3.3.4 Models describing the Class/Object Internal level of abstraction

The Class/Object Internal level of abstraction describes the internal behavior of objects.

The Analysis phase describes this level of abstraction for Semantic Classes and Objects of the application to be developed.

The Design phase also describes this level of abstraction for Solution Classes and Objects in the application to be developed.

Notations that belong to the Object Internal Behavior Model, Data Manipulation Model and the Class Structure Model are used to described this level of abstraction.

Object Internal Behavior Model

Notations that are classified under the Object Internal Behavior Model describe the internal behavior of objects.

Semantics provided by Model: Describes the internal behavior of objects (i.e. Semantic Objects in the case of Analysis and Solution Objects in the case of Design) in terms of states and events.

Generic Constructs:
- States
- Events (correspond to an operation invocation of an object)
- Guards (a condition that must be true for the event to be received)
- Actions (set of steps that an object must perform after the reception of an event, for example the invocation of an operation)

Notations:
- `State Transition Graph` notation in Jacobson [18]
- SDL notation in Jacobson [18]
- `State Diagram` notation in Rumbaugh [14]
- `State Transition Diagram` notation in Booch [22] and in Shlaer-Mellor [24]
- `State Transition Table` notation in Shlaer-Mellor [24]
- `ObjectCharts` notation in Coleman et all. [71]
- `OAN Object Dynamic` notation in De Champeaux [15]

Data Manipulation Model

Notations that are classified under the Data Manipulation Model describe the internal processing of data.

Semantics provided by Model: Describes the effects on data of an operation (i.e. event) in a group of objects (i.e. Semantic Objects in the case of Analysis and Solution Objects in the case of Design).

Generic Constructs:
- Process (corresponds to the execution of an operation)
- Data
- Data flows

Notations:
- `Data Flow Diagram` notation in Rumbaugh [14]
- `Action Data Flow Diagram` notation in Shlaer-Mellor [24]

Class Structure Model

Notations that are classified under the Class Structure Model describe the internal structure of a class .

Semantics provided by Model: Describes the internal structure of a class as well as its flow of data and control (i.e. Semantic Classes in the case of Analysis and Solution Classes in the case of Design).

Generic Constructs:
- Module (operation of a class)
- Data flows
- Control flows

Notations:
- `Class Structure Chart` notation in Shlaer-Mellor [24]

4.3.3.5 Models describing both the Class/Object and Class/Object Internal level of abstraction

The following model provides both the Class/Object and Class/Object Internal levels of abstraction for Semantic Objects at the Analysis phase and for Solution Objects at the Design phase.

Event Manipulation Model

Notations that are classified under the Event Manipulation Model describe the communication between objects as well as their internal behavior.

Semantics provided by Model: Describes the sequence of events and states for a group of objects (i.e. Semantic Objects in the case of Analysis and Solution Objects in the case of Design).

Generic Constructs:
- States
- Events
- Entities

Notations:
- `Thread of Control Chart` notation in Shlaer-Mellor [24]

4.3.4 The Views

A View is a higher level of abstraction then a Model. It corresponds to the semantics described by a set of Models. The Models described in the previous section provide semantics (Views) of the system as shown in Table 4.3 at different levels of abstraction.

While a Model corresponds to a grouping of notation based on semantics and level of abstraction, a View corresponds to a grouping of Models based on semantics only. Thus a View can provide different levels of abstraction as indicated by its Models. For example, in Table 4.3, the Static View is offered at the Requirements level of abstraction (as denoted by notations in the Domain/Subsystem Configuration Model) and at the Class/Object level of abstraction (as denoted by notations in the Class Configuration Model and Object Configuration Model).

Note that in Table 4.3, we only focus on analysis and design activities. Testing is distributed between each phase, but is not shown here.

Since a View is a grouping of Models with same semantics, a View can be expressed by the set of generic modeling constructs as defined for Models.

Static View

The Static View of a system corresponds to its non-behavioral view. It does not necessarily describe the actual structure or architecture of the system; it simply captures all static information. We have identified three Models, described in the previous section, that provide the Static View of a system:

- *Domain/Subsystem Configuration Model*
- *Class Configuration Model*
- *Objects Configuration Model*

Generic Modeling Constructs: Entities,
Relationships (i.e. Aggregation and Association),
Properties (i.e. Attributes)
Operations

The Static View of the application to be developed is described both at the System and Class/Object level of abstraction for the Requirements Modeling, Analysis and Design phases of development.

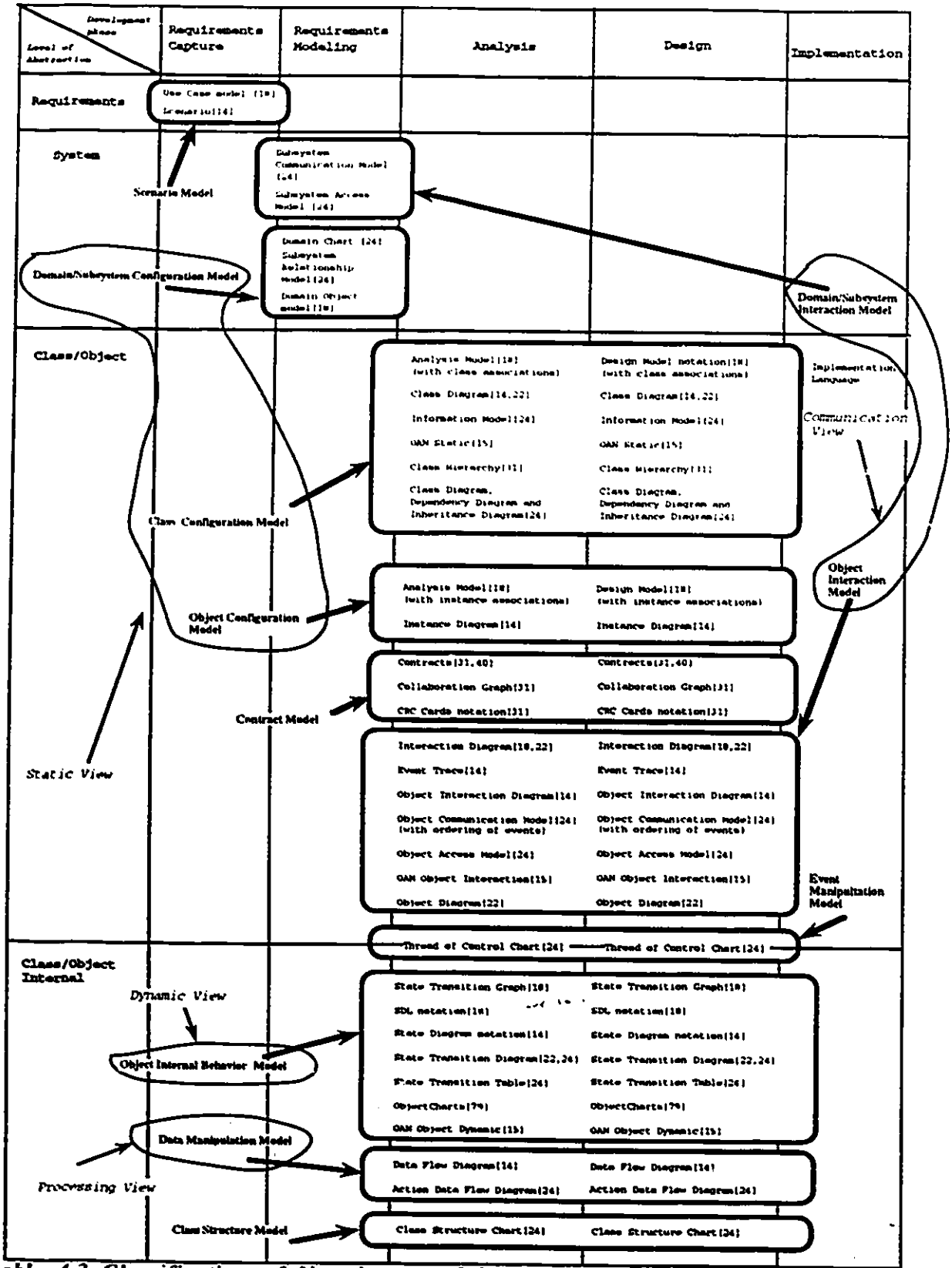


Table 4.3 Classification of Notations used in the Object-Oriented world into Models and Views

Communication View

The Communication View of a system describes the communication between entities without considering the internals of these entities.

We have identified two Models, described in the previous section, that provide the Communication View of a system:

- *Domain/Subsystem Interaction Model*
- *Object Interaction Model*

Generic Modeling Constructs: Entities, Messages (i.e. operation invocations of Entities) and Ordering of messages.

The Communication View of the application to be developed is described both at the System and Class/Object level of abstraction for the Requirements Modeling, Analysis and Design phases of development.

Dynamic View

The Dynamic View of a system describes the internal behavior of the entities in the Communication View.

We have identified one Model, described in the previous section, that provide the Dynamic View of a system:

- *Object Internal Behavior Model*

Generic Modeling Constructs: States,
Events (i.e. operations of an Entity invoked by other Entities),
Guards (i.e. conditions that must be true for Events to be received),
Actions (i.e. set of steps that an Entity must perform after the reception of an Event, for example the invocation of an operation).

The Dynamic View of the application to be developed is described at the Class/Object Internal level of abstraction for the Analysis and Design phases of development.

Processing View

The Processing View of a system describes the effects on data of operations or events of the system.

We have identified one Model, described in the previous section, that provide the Processing View of a system:

- *Data Manipulation Model*

Generic Modeling Constructs: Process, Data, and Data Flow

The Processing View of the application to be developed is described at the Class/Object Internal level of abstraction for the Analysis and Design phases of development.

Note that, there are four Models that do not fit in the Views defined above: the *Scenario Model*, the *Contract Model*, the *Event Manipulation Model* and the *Class Structure Model*.

These Models either provide more than one View or an aspect of a View as shown below. They are used for testing other Models. This is discussed in Chapters 6 and 7.

The *Scenario Model* provides aspects of the Communication View, since every use of the system can be viewed as communication between the User and the System. However, it does not provide messages exchanged between the User and the System.

The *Contract Model* glues the Communication View and the Static View. The Model describes the collaboration between Classes in term of operations implemented by each class (Static View) and their ordering (Communication View).

The *Event Manipulation Model* glues the Communication View and the Dynamic View, since it describes the interaction between object (Communication View) and their internal behavior (Dynamic View).

The *Class Structure Model* provides aspects of the Processing View, since it describes operations, control flow and data flow, but does not describe the processing of data.

4.3.5 The integration of Process, Notations, Models and Views

The purpose of this chapter was to introduce a Generic Object-Oriented Development Methodology (Process, Notation, Models and Views) whose focus is to provide a way of handling change in a consistent manner. We provided specific definitions used in the methodology. We also classified notations used in the OO world into Models and Views defined by our methodology for testing purposes.

We conclude this Chapter by integrating development phases, levels of abstraction and Views.

Phase: Requirements Capture

Goal: Describes the **Requirements** level of abstraction, in text format or Scenario format. This forms a specification document. Text can capture different aspects of the application to be developed (e.g. performance, data, architecture, behavior, etc.). Notations that fall under the Scenario model, capture uses of the system. From these scenarios, a structure can be derived (i.e. Static View) as described in [18].

Phase: Requirements Modeling

Goal: Describes the **Static** and **Communication** views of the application to be developed at a **System** level of abstraction, in terms of Domains, Subsystems and Domain Classes/Objects.

Phase: Analysis

Goal: - describes the **Static** and **Communication** views of the application to be developed at a **Class/Object** level of abstraction, in terms of Semantic Classes/Objects,

- describes the **Dynamic** and **Processing** views of the application to be developed at a **Class/Object Internal** level of abstraction, in terms of Semantic Classes/Objects.

Phase: Design

Goal: - describes the **Static** and **Communication** views of the application to be developed at a **Class/Object** level of abstraction, in terms of Solution Classes/Objects.

- describes the **Dynamic** and **Processing** views of the application to be developed at a **Class/Object Internal** level of abstraction, in terms of Solution Classes/Objects.

Phase: Implementation

Goal: - describes the **Static** and **Communication** views of the application to be developed at a **Class/Object** level of abstraction, in terms of coded classes/objects (i.e. implementation language).

- describes the **Dynamic** and **Processing** views of the application to be developed at a **Class/Object Internal** level of abstraction, in terms of coded classes/objects (i.e. implementation language).

Chapter 5 Testing Activity Principles and Types

5.1 Introduction

The purpose of this chapter is to introduce *testing activity types* that should take place at different phases (i.e. Requirements Capture, Requirements Modeling, Analysis and Design) of the Generic Object-Oriented Development Methodology described in Chapter 4.

We start by providing the benefits of introducing testing activities early in the development process. We then describe the concepts of Verification, Validation, Certification and Incorporation. These will be referred to as *testing activity principles.*

We will expand these concepts into *testing activity types* of the Generic Object-Oriented Development Methodology. This is based on the semantics (i.e. different Views: Static, Dynamic, Communication and Processing) of Models as described by Notations and the syntax of Notations. These are presented in details in Chapters 6 and 7 with examples.

5.2 Benefits of introducing testing activities early in the development process

As stated in Chapter 3, the rational for using notations that exhibit the basic testability requirements (i.e. verifiability, modifiability and traceability) is that these notations can be tested for testability attributes (i.e. consistency, correctness, completeness and incorporation) through testing activities, thus achieving traceability. This provides the ability of:

(1) uncovering errors early in the process where they are less costly to fix
(2) generating test cases that can be applied to the implementation
(3) guiding the testing of the implementation.
(4) facilitating the location of source(s) of error(s) for modification

We will discuss these separately.

5.2.1 Uncovering errors before implementation

Uncovering errors before implementation can be achieved by introducing testing activities early in the software development process.

The advantage (less costly to find and correct) of finding errors before coding is well recognized in software engineering. In [7] a number of validation techniques that allow finding high level errors before coding are discussed. These are divided into Specification Based and Design Based.

Specification Based testing techniques aim at detecting incomplete, inconsistent and misleading requirements. An example is given in [51]. Also Formal Specification Languages like Ina Jo [50], allow specifications to be executed and thus tested.

Design Based testing techniques aim at verifying mainly two things: (1) that Analysis and Design notations reflect the requirements in the specification and (2) that Analysis and Design notations are complete, consistent, correct and integrated in themselves and with respect to each other.

In terms of testaoility (a measure of how easy the testing of source code is made), Specification Based and Design Based testing techniques are useful if refinements (i.e. mappings and transformations[13]) exist throughout the software development process. For example, if the source code of a system is not a refinement of its Design that was tested using Design Based testing techniques, nothing prevents the design errors from reoccurring in the source code, thus the effort spent testing the design may be lost. Thus Analysis notations should be viewed as refinements of Requirements Modeling notations, Design notations should be viewed as refinements of Analysis notations and source code should be viewed as a refinement of Design notations.

In the OO paradigm, as observed by the authors in [8], the refinements are less radical then in other development methods: "Analysis models, Design models are implementation models all use classes and objects as the primary vehicles for representation. Even though a model shows some particular view that is different from another model, all of the models are illustrating aspects of classes and objects." [8].

[13]By mappings or transformations, we understand the valid refinements that notations can undergo when moving from one phase of development to the next. This is discussed in more details throughout this chapter.

CASE (Computer Aided Software Engineering) tools, such as Statemate[14] [52], SDT[15] [53, 54] and Argos[16] [55] provide mechanism to validate and verify models used and automate mappings and transformations.

We cannot however completely rely on the source code generated by CASE tools. The generated code is typically of prototype quality and is useful for simulation applications and not as efficient and as fine tuned as production code.

Therefore, we can say that there are two options for uncovering errors early in the OO software development process: (1) apply Specification Based and Design Based testing techniques early in the process and ensure correct mappings and transformations from one phase to the next; (2) rely on CASE tools that automate the testing of Analysis and Design models and generate source code from the tested models. Throughout the rest of this document we look at option Design Based testing techniques in point (1) above.

Testing activity types for OO notations are described in section 5.4. Here, we provide some background information.

There are four levels of testing in the Generic Object-Oriented Development Process introduced in Chapter 4. These four levels correspond to the four levels of abstraction of the application to be developed: Requirements, System, Class/Object and Class/Object Internal. The four levels of abstraction in turn, correspond to phases of the process, as shown in Fig. 5.1. As presented in Chapter 4, at every level of abstraction, different Views of the application to be developed are provided. The four basic Views were:

---

[14]Statemate by I-logix Inc., is intended for code specification, analysis, design and documentation of large and complex reactive systems, such as real-time embedded systems, control and communication systems and interactive software. Statemate can generate Ada code and provides several tools for testing and querying the design. Thus, if a design is expressed using the Statemate case tools and has been analyzed for deadlock, loop and liveness properties, via automatic Reachability Analysis, the generated Ada code that implements this design was made easier to test, since it should not contain deadlocks, loops, etc.

[15]SDT (SDL Design Tool) can generate C code from design expressed in SDL (System Description Language). The different components of SDT: Graphical Editor, Analyzer, Simulator, Report Generator, Validator etc. can create, edit SDL specifications, perform syntactic and semantic analysis of the SDL system, report errors, debug, perform State Space Exploration etc. on the SDL system. The C code generated is made easier to test since errors such as deadlock, can be discovered at the design stage.

[16]Argos by Miramar Technology, is a graphical object-oriented development environment that generates Smalltalk code from class structures. The code is thus always traceable to the class structures. The tool verifies that coherent diagrams are drawn (e.g. no self Inheritance). Since the code is generated from the diagrams, these incoherence are eliminated.

- the **Static View** of a system corresponds to its non-behavioral view (i.e. structure or architecture of the system) and captures all static information

- the **Communication View** of a system describes the communication between entities without considering the internals of these entities.

- the **Dynamic View** of a system describes the internal behavior of the entities in the Communication View.

- the **Processing View** of a system describes the effects on data of operations or events of the system.

As shown in Fig. 5.1, the level of abstraction is coupled to the View (i.e. the System and Class/Object levels of abstraction address the Static and Communication View; the Class/Object Internal level of abstraction addresses the Dynamic and Processing Views).

Throughout this chapter, we will show that testing activity types should take place at different phases of development based on the Views provided at each phase. Furthermore, we will show that these testing activity types do not depend on the phase of the Generic Object-Oriented Development Process. Thus, a testing activity type used to test the Static View, is not dependent on whether the artifact being tested is a Domain, Subsystem, Domain Class of the Requirements Modeling phase or whether it is a Semantic Class of the Analysis phase (i.e. the same testing activity can be used for all artifacts and phases).

Also, we will show that testing activity types are generic in the sense that they do not dependent on the Notation used by Views and that they are refinable (adaptable to different levels of abstraction) in order to be uniformly applied at different phases of development.

Finally, note that testing the Requirements Capture level of abstraction is beyond the scope of our work, we therefore introduce it only very briefly in the discussions that follows.

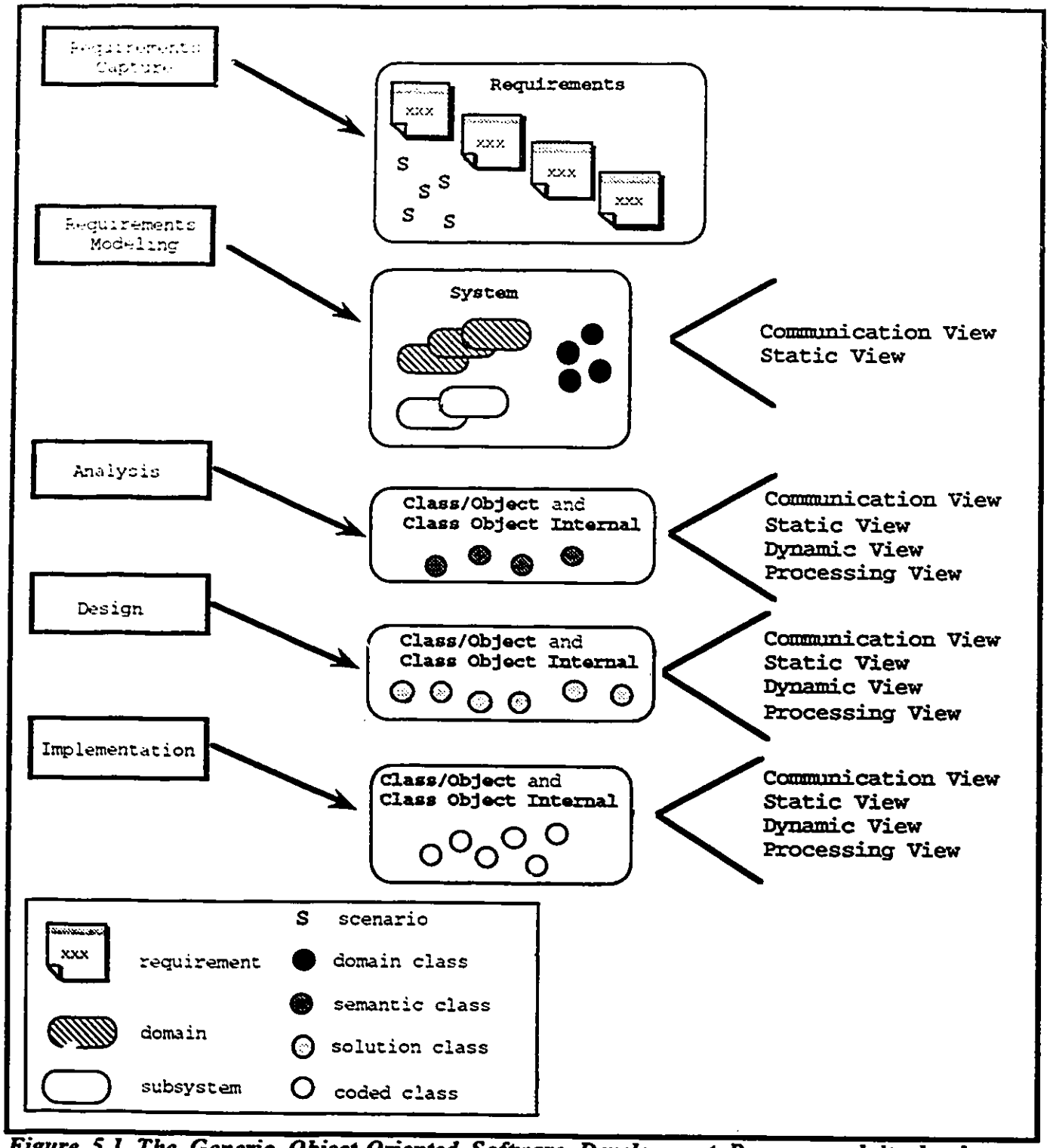


Figure 5.1 The Generic Object-Oriented Software Development Process and its level of development abstraction

In Fig. 5.1, at the Requirements Capture phase, the application is viewed as a set of Requirements or Scenarios: these requirements are the first artifact to be tested.

At the Requirements Modeling phase, the application is viewed as a set of Domains, Subsystems and Domain Classes/Objects derived from the Scenarios of the Requirements Capture phase: these become the next artifact to be tested. The latter should also be tested against the requirements in the Requirements Capture phase.

At the Analysis phase, the application is viewed as a set of Semantic Classes/Objects: these become the third artifact to be tested. As in the previous case, these should be tested against the Domains, Subsystems and Domain Classes/Objects of the Requirements Modeling phase.

At the Design phase, the application is viewed as a set of Solution Classes/Objects: these become the next artifact to be tested. These must be tested against the Semantic Classes/Objects of the Analysis phase.

At the Implementation phase, the application is viewed as a set of Coded Classes/Objects: these become the fifth artifact to be tested. The latter must be tested against the Solution Classes/Objects of the Design phase.

As we go from one phase to the next, the notations used at the previous phase have been tested, thus decreasing the probability of creating notations at the next phase based on notations that contains errors.

Note that, the granularity of the artifact to be tested increases as we move from one phase to the next. We will show that testing activities must be adapted to the appropriate level of abstraction.

5.2.2 Generating test cases from notations

The idea of generating test cases[17] from notations is well established. The basic idea is to derive test cases from notations and apply these test cases to the implementation. We study the derivation of test cases from notations in more detail.

As described by the authors in [56], testing of OO systems must be viewed from different levels abstraction. *Unit* testing corresponds to testing the interaction of operations and data that are encapsulated within a class. *Cluster* or *Subsystem* testing considers the interactions of a group of collaborating classes. *System* testing is the testing of the complete application (i.e. all code from all classes).

Generating Unit, Cluster and System test cases directly from the implementation is time consuming, error-prone and tedious. Rather, test cases should be generated from the different notations used in the Generic Object-Oriented Development Process and be constructed (i.e. refined) as we go from one phase to the next. Fig. 5.2 illustrates this process.

Requirement's Test Cases are generated from notations used at the Requirements level of abstraction (usually text or scenarios), as shown by the directed line labeled as `(1)`. These test cases are used to construct System level test cases, as shown by the directed line labeled as `(2)`. That is, the Requirements Test Cases should be refined to be expressed in terms of Domains, Subsystems and Domain Classes/Objects.

The System Test Cases generated from the Requirements Test Cases are augmented with test cases generated from notations used at the System level of abstraction, as shown by the directed line labeled as `(3)`.

The System Test Cases are used to construct Cluster Test Cases (i.e. System Test cases should be refined to be expressed in terms of Semantic objects of the Analysis), as shown by the directed line labeled as `(4)`. Lets take a brief example to illustrate the discussion. Assume that the association between Subsystem `Teller` and `Bank`, is `reports`, as shown below.

A System test case would ensure that there are abstract operations (i.e. operations that require further refinements as objects or as other operations) in both `Teller` and `Bank` that satisfy the association. At Analysis, the Cluster test case derived from this System test case would ensure that there are Semantic objects in both subsystems that still satisfy the relationship.

---

[17]By a Test Case, we understand: input and expected output.

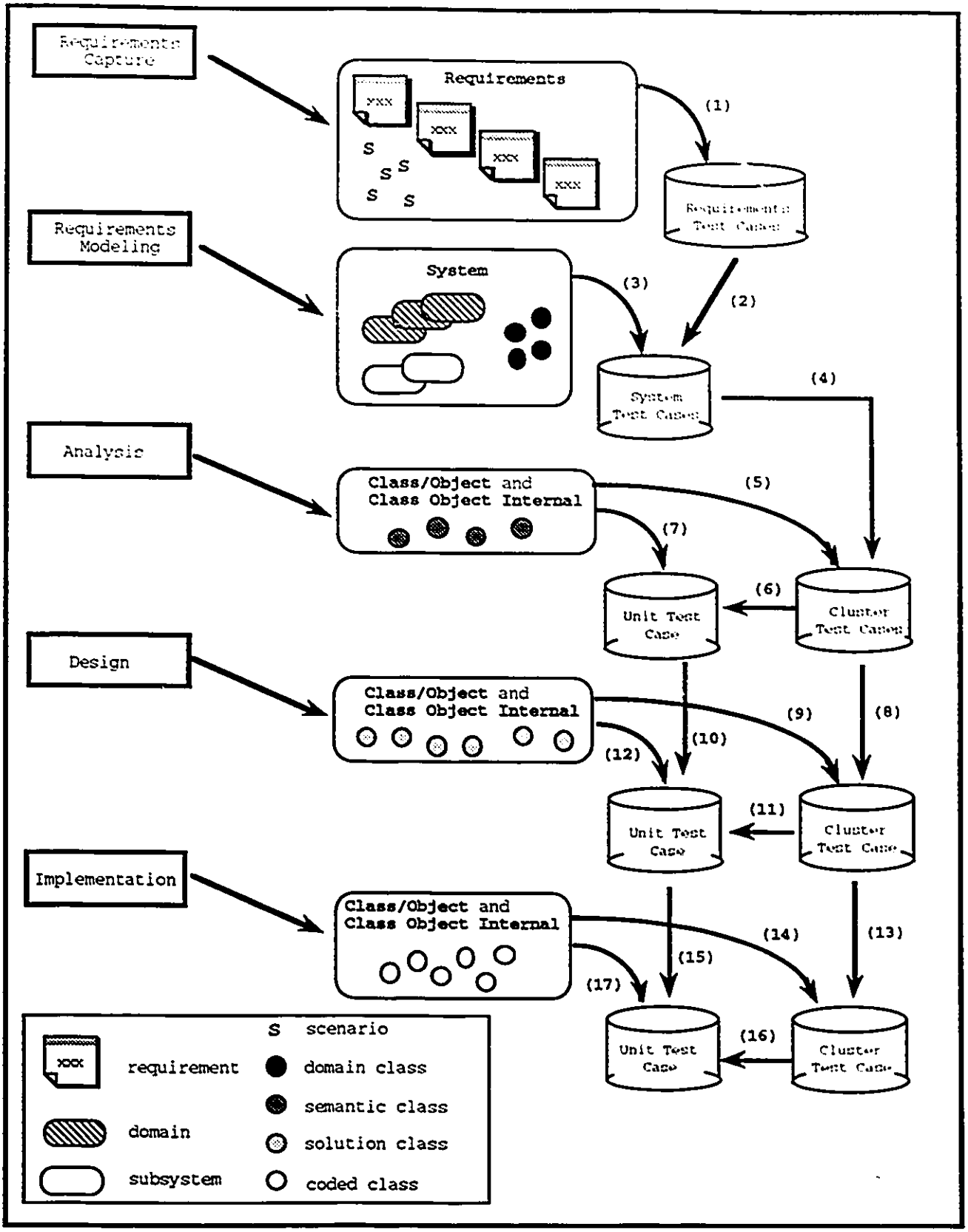
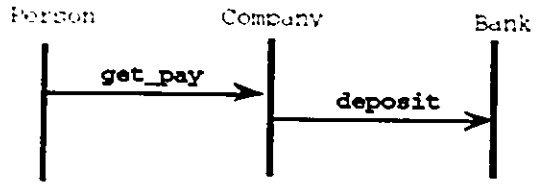


Figure 5.2 Process of reusing Test cases from notations

Cluster Test Cases are augmented with test cases generated from notations used to express the Class/Object level of abstraction in terms of Semantic Classes/Objects, as shown by the directed line labeled as (5).

Cluster Test Cases are in turn used to generate Unit Test Cases for Semantic objects, as shown by the directed line labeled as (6). Here, we can view Cluster Test Cases as identifying the *input* and *expected output* of an object. Lets take a simple example. Assume that a Cluster Test Case determines that when the Semantic Class Company receives the message **get_pay** (i.e. operation **get_pay** is invoked by Person), it must send the message **deposit** to Bank.



When performing Unit testing of Company, we know that upon invocation of operation **get_pay** (i.e. *input)*, the behavior of class Company should generate message (i.e. invoke operation) **deposit**.

In Fig. 5.2, Unit Test Cases are augmented with test cases generated from notations used to express the Class/Object Internal level of abstraction in terms of Semantic Classes/Objects, as shown by the directed line labeled as (7).

In this manner, the same process is repeated at Design for Solution Classes/Objects and at Implementation for Coded Classes/Objects. Thus, the final set of test cases (i.e. test suite) that will be applied to the implementation has been constructed through a step-wise refinement process all the way from Requirements.

Note that the basic idea is that test cases should follow the same principles of traceability described in section 3.2.2.3.1, (e.g. a Unit Test Case at the Implementation phase should be traceable to its Cluster test case at Implementation, its Unit Test Case at Design, its Cluster Test Case at Design, its Unit Test Case at Analysis, its Cluster Test Case at Analysis, its System Test Case and finally its Requirements Test Case). Note that this is specifically true for Use Case directed design.

5.2.3 Directing the testing of the implementation

Testing early in the OO software development process allows to guide the testing of the implementation.

Two of the fundamental differences between testing implementations that follow the function/data paradigm versus the object-oriented paradigm are the Randomness of Operations problem [56] and the Localization problem [9].

The Randomness of Operation problem

The objects of the OO model are composed of operations and possibly data-structures that contain the state of the object. No ordering of the invocation of those operations is specified explicitly once the object has been created, although there may be some implied order. There is thus no sequential input, process, output model into which testing methods can be adapted. Therefore, the testing process becomes a searching problem since programs (objects) are not executed sequentially. This is because operations in a class can be combined in arbitrary order. The testing thus involves searching for the order in which the operations can be executed with various parameters that yield errors.

Consider an example of the Randomness of Operations problem from [56]. A class represents a clock object and provides operations to set and retrieve the time, increment and decrement the time by one second, and produce a display of the current time in a graphical form. There is no stipulation in the ordering that these operations should be invoke. The time may be retrieved a number of times in a timing exercise or set a number of times, if the user wishes to examine the effect of different time zones. Operations simply provide a service. Given that an error may occur after a particular sequence of operation invocations of an object, the problem in terms of testing can be seen as a search for the correct order to reproduce a specific error.

Testing early in the OO development process can greatly alleviate the Randomness of Operation problem. The basic idea is that the ordering of invocation of operations should not be identified directly from the implementations, but rather from notations used at the Analysis and Design phases, like Class Diagrams [14]. Example 1 in Chapter 6 illustrates an example of the latter.

The Localization problem

The author in [9] discusses that one reason why testing of object-oriented programs is different from the testing of more conventional software is *localization*. Localization is the process of placing items in close physical proximity to each other. He observes that in function/data paradigm (software created using functional decomposition), the localization is based on functionality and thus there is a high degree of correlation between "testing a function" and a "unit of software". In the OO paradigm, the localization of information is around classes or instances of these classes: objects. Thus a class or an object is the "unit of software". The correlation between "testing a function" and a "unit of software" is not one-to-one as in the function/data paradigm. A class or an object may provide many functions and one function may involve several classes or objects.

Testing early in OO development process can greatly alleviate the Localization problem. The basic idea is that the identification of "functions" should not be left to the implementation phase, rather, these functions and the object(s) that participate in the functions can be determined from notations (e.g. Class Diagrams [14], Interaction Diagrams [22]) used at the Analysis and Design phase. Example 1 in Chapter 6 illustrates an example of the latter.

5.2.4 Location of source of error

In section 3.2.2.3.1, we defined the 5 types of traceability that notations should exhibit (horizontal, vertical upwards, vertical downwards, forward iteration and backwards iteration).

This can be achieved by annotating or tagging components of the system development (e.g. a requirement, an interaction between objects as described in an Interaction Diagram [18], a class as described in a Class Diagram, a path in SDL [18] describing some aspect of the internal behavior of an object, a Coded Class, etc.).

If the development of a system (i.e. components) is tagged in such a way as to provide the 5 types of traceability, the location of the source of an error is greatly facilitated. The software designer will be able to answer, in which iteration, which phase, which notation and/or in which component of that notation the error occurred. Also he/she would be able to determined other components in other phases and iterations could be affected by the error.

To achieve this ideal situation, tools like RTM (Requirements Tracing and Management) [68] and DOORS [70] capture some traceability information. However, the level of granularity of the component to be tagged is currently not sufficient. They stop at the object level and one must manually tag the internals of an object (i.e. its behavior and data).

5.3 Testing Activity Principles

We define four basic testing activity principles (i.e. verification, validation, certification and incorporation) from which *testing activity types* will be derived. We will show that these were selected in the context of an iteration style for software development.

Fig. 5.3 shows a high level view of the testing activity principles. *Verification* aims at ensuring Horizontal traceability of notations within a phase of a given iteration. *Validation* aims at ensuring Vertical Upward and Vertical Downward traceability of notations of consecutive phases within a given iteration. *Certification* aims at ensuring conformance to requirements of notations at different phases for a given iteration. *Incorporation* aims at ensuring that two consecutive iterations that have undergone Verification, Validation and Certification have Forward Iteration and Backward Iteration traceability. We will discuss these separately.

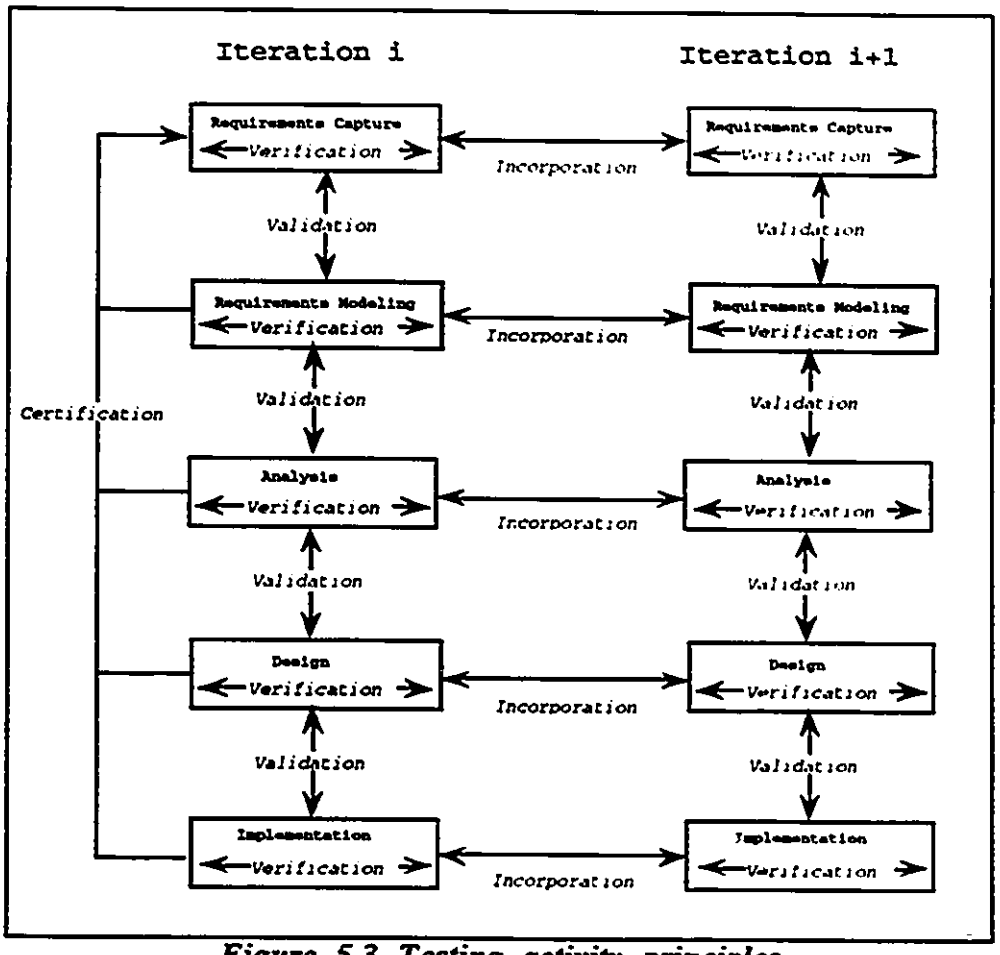


Figure 5.3 Testing activity principles

5.3.1 Verification

"Verification is the process of checking whether the system is being built right." [69]. This involves determining whether the system contains errors. We define Verification as the set of activities that allows to detect errors in the semantics and syntax provided by Notations used at a **specific** phase of the Generic Object-Oriented Development Methodology. We can therefore talk of

- `Requirements Capture Verification`,
- `Requirements Modeling Verification`,
- `Analysis Verification`,
- `Design Verification`,
- `Implementation Verification` (usually referred to as Testing).

Thus, Verification aims at determining whether Notations used at phase n of a given iteration are complete and correct with respect to other Notations at phase n of the same iteration and whether these Notations are correct in themselves. Verification aims at providing Horizontal Traceability described in 3.2.2.3.1.

Fig. 5.4 shows when Verification should take place with respect to the development activities described in Chapter 4.

The figure shows that after each development activity, Verification should take place. Moving to the next development activity (e.g. Design) is contingent on the verification performed on Notations used at the previous development activity (e.g. Analysis). The flow from a verification activity to a development activity is a set of verified Notations. For example, the output of the `Requirement Modeling Verification` phase is a set of verified (i.e. complete and correct) Notations expressing Domains, Subsystems and Domain Classes/Objects.

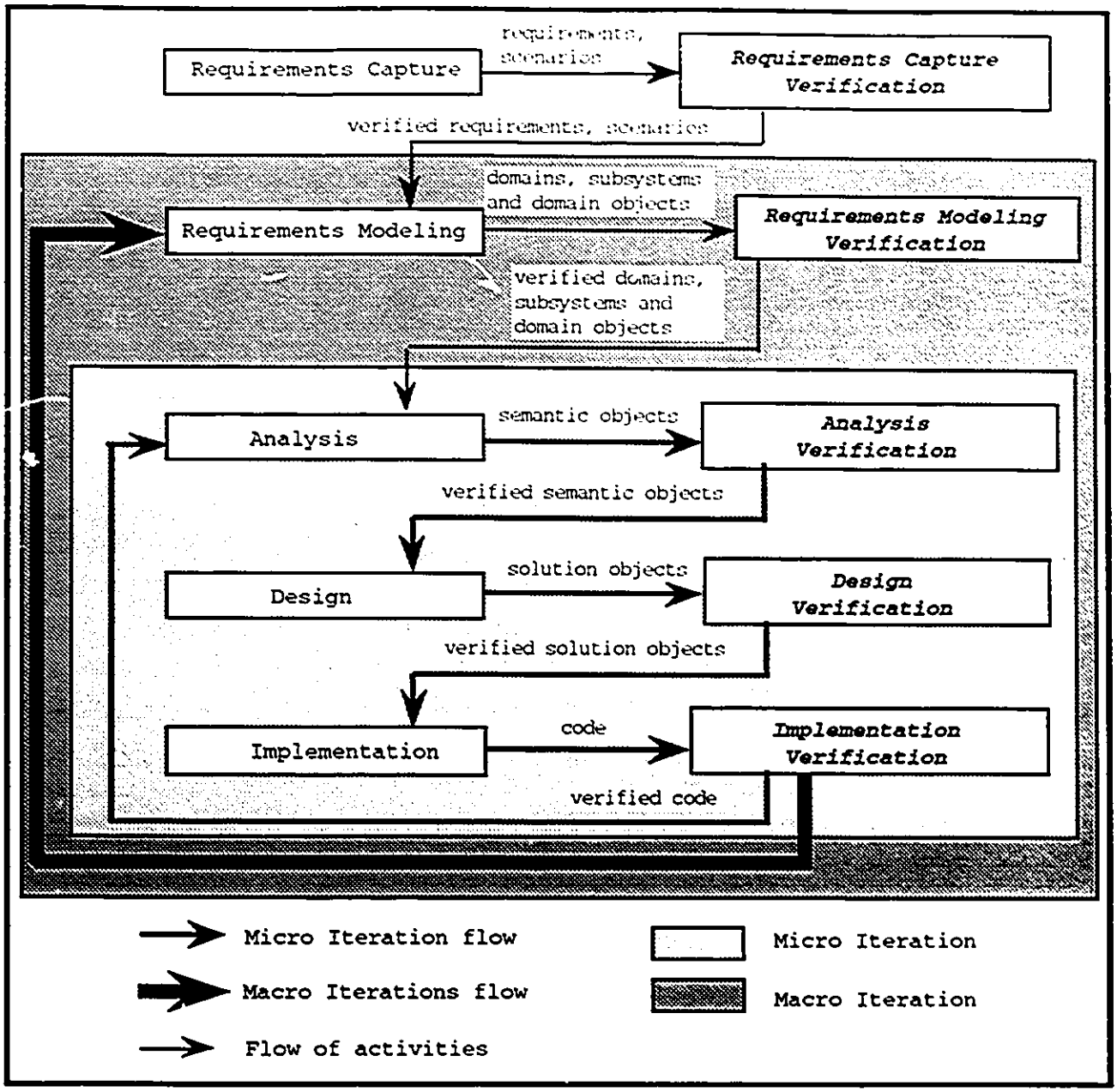


Figure 5.4 Integration of Development activities and Verification testing activities

5.3.2 Validation

"Validation is the process of checking whether the right system is being built." [69]. This involves determining whether what is being implemented or modeled conforms to requirements. Thus, Validation requires a reference point. We define Validation as the set of activities, that allow to determine whether Notations used at phase *n* (e.g. Analysis) are consistent with respect to Notations used at phase *n+1* for a given iteration (e.g. Design) and whether Notations used at phase *n+1* (e.g. Design) are consistent with respect to Notations used at phase *n* for the same iteration. We can therefore talk of

- `Requirements Capture vs. Requirements Modeling Validation,`
- `Requirements Modeling vs. Analysis Validation,`
- `Analysis vs. Design Validation,`
- `Design vs. Implementation Validation.`

Validation aims at providing Vertical Upwards and Vertical Downwards Traceability described in 3.2.2.3.1.

Fig. 5.5 shows when Validation should take place with respect to the development activities described in Chapter 4.

The figure shows that after each development activity, Validation should take place. Moving to the next development activity (e.g. Design) is contingent on the validation performed on Notations used at the two previous development activities (e.g. Requirements Modeling and Analysis). The flow from a validation activity to a development activity is a set of validated Notations. For example, the output of the `Requirements Modeling vs. Analysis Validation` phase is a set of validated Notations indicating that Notations used at the Requirements Modeling phase are consistent with respect to Notations used at the Analysis phase and vice-versa.

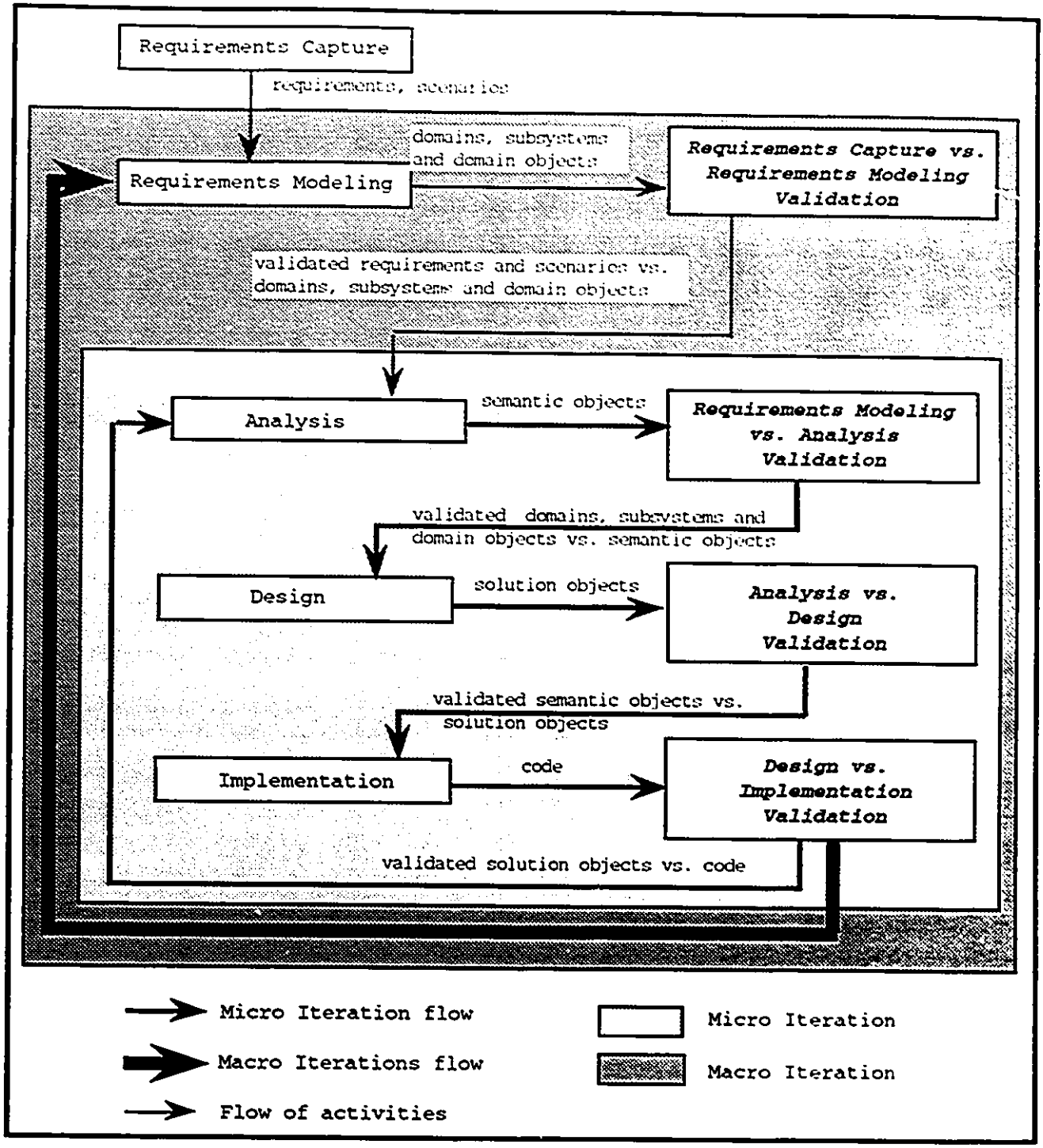


Figure 5.5 Integration of Development activities and Validation testing activities

5.3.3 Certification

We define Certification as the process of determining whether at different phases of software development Notations used are complete, consistent and correct with respect to the original requirements (described in the Requirements Capture phase). Thus we can talk of

- Requirements Modeling Certification,
- Analysis Certification,
- Design Certification,
- Implementation Certification.

As said earlier, Verification provides Horizontal Traceability and Validation provides Vertical Upwards and Vertical Downwards Traceability. Therefore, Certification is a result of Verification and Validation and is not required to be a separate testing activity. In this manner we say that:

Requirements Modeling Certification
is provided by (equals)
Requirements Modeling vs. Requirements Capture Validation.

As an example, we can also say that

Design Certification (i.e. whether the Design implements the original requirements), is a result of having performed the following testing activities:

Requirements Capture Verification followed by
Requirements Modeling Verification followed by
Requirements Modeling vs. Requirements Capture Validation followed by
Analysis Verification followed by
Requirements Modeling vs. Analysis Validation followed by
Design Verification followed by
Analysis vs. Design Validation.

5.3.4 Incorporation

We define Incorporation as the process of determining whether Notations used at different iterations are incorporated to one another. More specifically, Incorporation refers to the set of activities that allow to determine if Notations at phase n of iteration i are incorporated in notations at phase n of iterations $i+1$ and whether Notations at phase n of iteration $i+1$ are incorporated in notations at phase n of iterations i. Thus, we can talk of

- `Requirements Capture Incorporation`
- `Requirements Modeling Incorporation`
- `Analysis Incorporation`
- `Design Incorporation`
- `Implementation Incorporation`

Incorporation provides Forward Iteration traceability and Backward Iteration traceability.

Fig. 5.6 shows when Incorporation should take place with respect to the development activities described in Chapter 4.

The figure shows that after each Macro-Iteration that has been verified, validated and certified, Incorporation should take place. Moving to the next Macro-Iteration is contingent on the incorporation performed on Notations.

The flow from an incorporation activity is a set of integrated Notations. For example, the output of the `Requirements Modeling Incorporation` phase of iteration i and $i+1$ is a set of integrated Domains, Subsystems and Domain Classes/Objects. These actually become the constraint (i.e. part of a requirement) under which the next iteration should be built. This is because the next iteration must take into account the incorporated components (i.e. Domains, Subsystems and Domain Classes/Object). This is in terms of how the new or refined component of the next iteration fit in with the incorporated components.

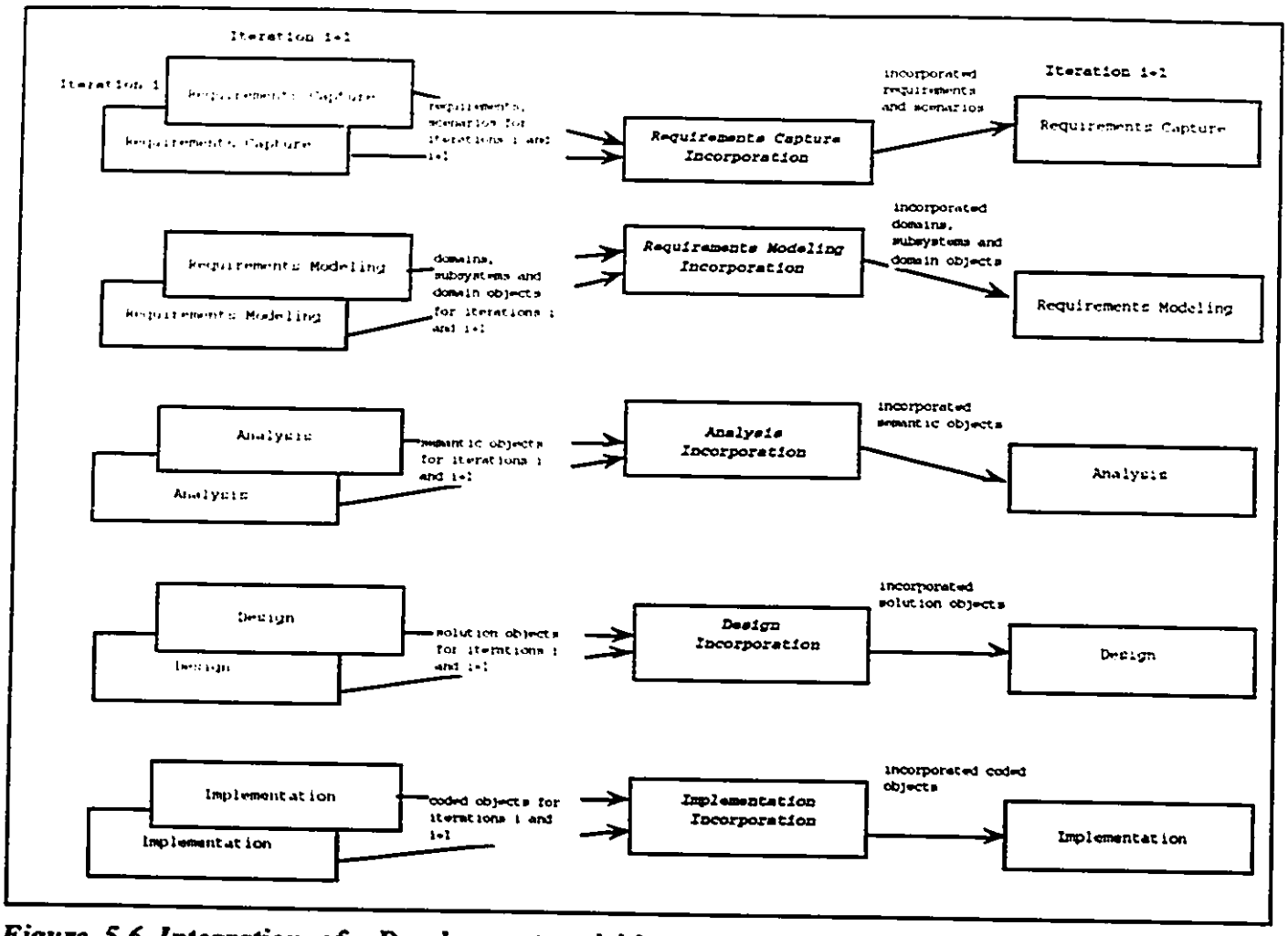


Figure 5.6 Integration of Development activities and Incorporation testing activities

5.4 Testing Activities Types

In Chapter 3, we determined that there should exist activities to test for the following testability attributes of notations.

- semantic completeness
- semantic correctness
- syntactic correctness
- semantic consistency
- semantic incorporation

In section 5.3, we defined the testing principles of Verification, Validation, Certification and Incorporation necessary to achieve traceability.

Table 5.1 describes the *testing activity types* introduced by our work and how they relate to testability attributes and to traceability

Testing Activity Types	What to test: testability attributes	Traceability type provided
Syntactic Verification[18]	syntactic correctness	Horizontal
Semantic Verification[19]	semantic completeness semantic correctness	
Semantic Validation [20]	semantic consistency	Vertical Upward Vertical Downward
Semantic Incorporation[21]	semantic Incorporation	Forward Iteration Backward Iteration

Table 5.1 Testing Activity Types

[18]Discussed in section 5.4.1
[19]Discussed in section 5.4.2
[20]Discussed in section 5.4.3
[21]Discussed in section 5.4.4

Fig. 5.7 illustrates the reusability of testing activities. The left part of the figure illustrates an abstract view of the reuse, the refined model is illustrated on the right.

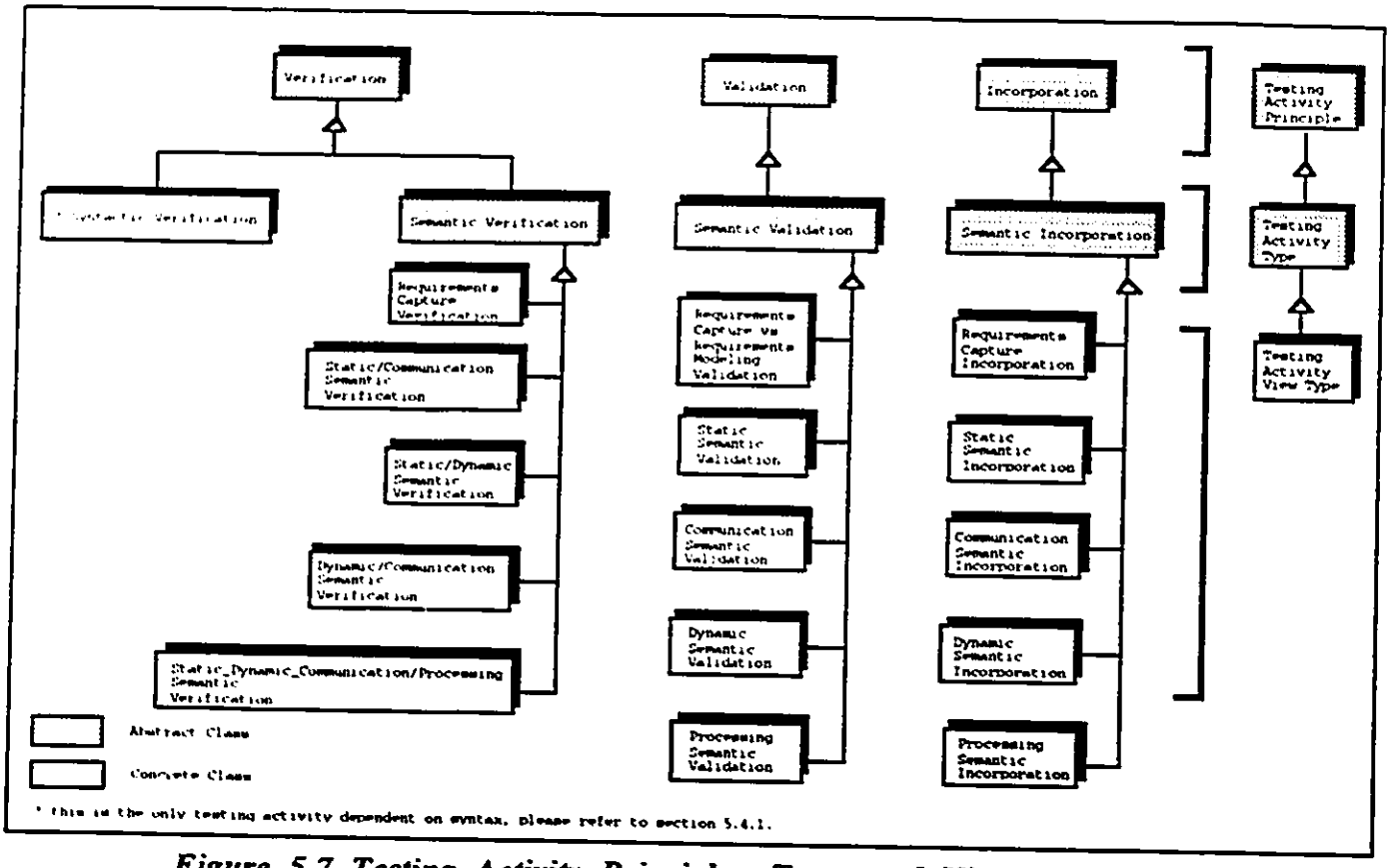


Figure 5.7 Testing Activity Principles, Types and View Types

Every `Testing Activity Type` (e.g. `Semantic Validation`) has specializations based on Views (e.g. `Static Semantic Validation`, `Communication Semantic Validation`, etc.). The latter are referred to as `Testing Activity View Types`. We will discuss these separately throughout the rest of this chapter.

Testing Activity View Types are generic in the sense that they are

- independent of development phase
- independent of notation
- refinable (i.e. adaptable to different levels of abstraction)

They depend only on the View provided at a development phase.

The basic idea is that notations can be tested by applying one or more of the sixteen Testing Activity View Types, illustrated in Fig. 5.7, to the View that the notation provides. This is based on the fact that modeling constructs of notations can be translated to modeling constructs of the View they provide. This was illustrated in Chapter 4. It is these modeling constructs that are tested by Testing Activity View Types. The testing involves determining the coverage criteria as well as assessment of the testability attributes (i.e. correctness, completeness, consistency and incorporation) of the View and thus of the notation. This is illustrated in Fig. 5.8.

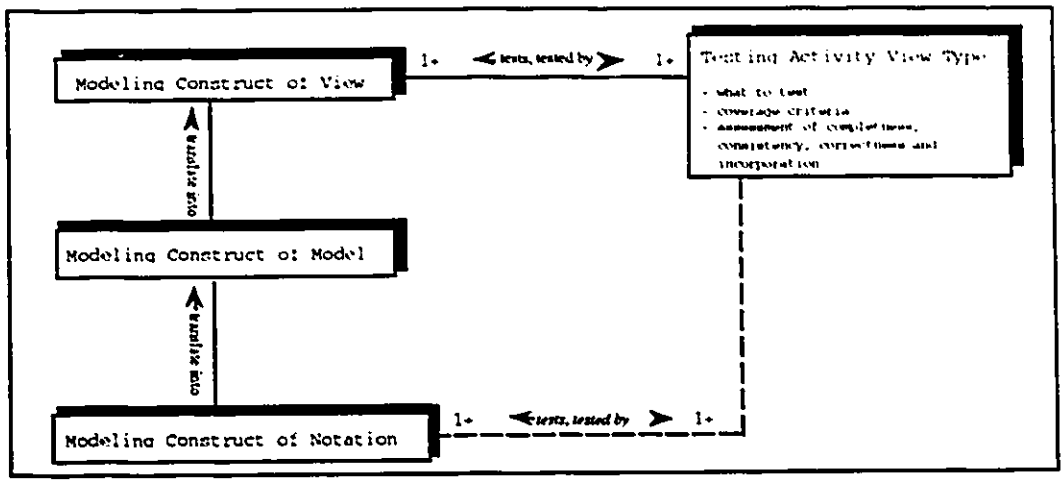


Figure 5.8 Testing of notations via translations of modeling constructs

Lets consider an example to clarify our discussion. As presented in Chapter 4, a notation is classified into a Model based on the level of abstraction and semantics the notation provides. The Model in turn is classified into a View based on the semantic it provides. Thus a notation provides a View; the modeling constructs of the notation will eventually translate into the modeling constructs for the View it provides.

Consider the Domain Chart [24] notation and the Class Diagram [14] notation. As discussed in Chapter 4, the Domain Chart notation was classified as belonging to the Domain/Subsystem Configuration Model and the Class Diagram notation was classified as belonging to the Class Configuration Model. Both of these Models provide the Static View.

In Fig. 5.9, `Static Semantic Validation` tests the modeling constructs of the Static View. The testing activity view type indirectly tests the modeling constructs of the notation Class Diagram and Domain Chart as illustrated by the derived association in italics.

The modeling constructs of the notation Domain Chart are translated into the modeling constructs of the Domain/Subsystem Configuration Model and then to the modeling constructs of the Static View.

The modeling constructs of the notation Class Diagram are translated into the modeling constructs of the Class Configuration Model and then to the modeling constructs of the Static View.

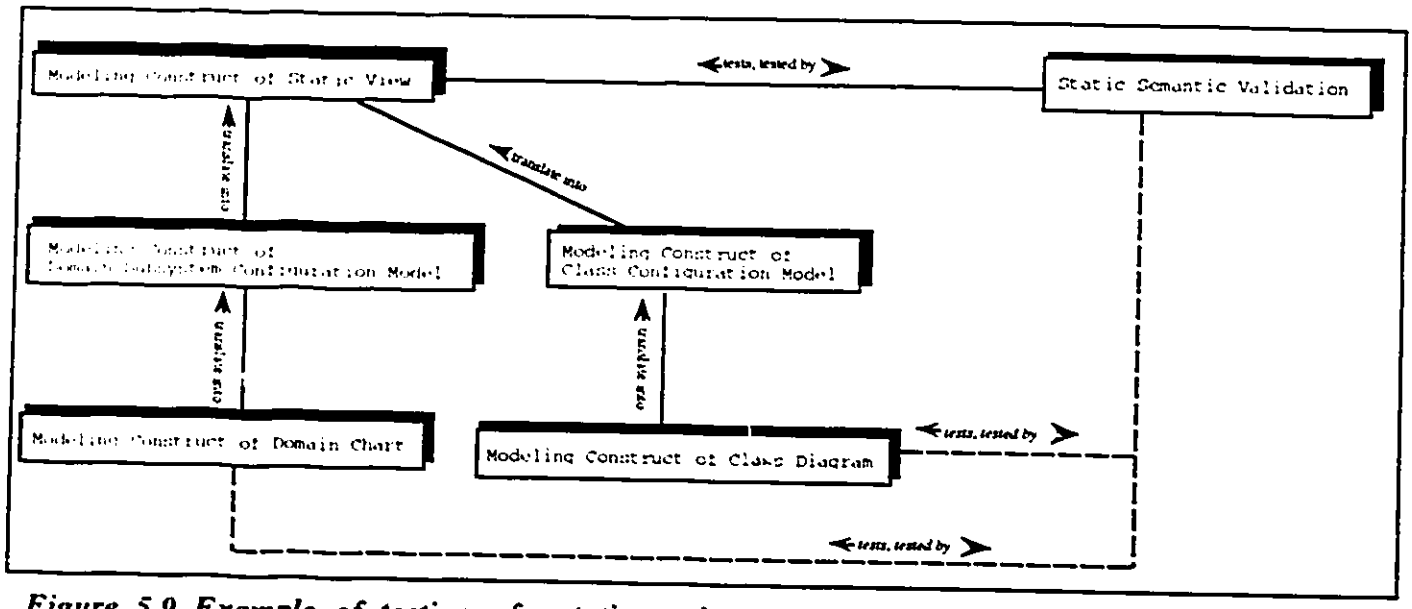


Figure 5.9 Example of testing of notations via translations of modeling constructs

Fig. 5.10 illustrates the instance configuration for Fig. 5.9. Fig. 5.10 is divided into 4 sub-figures for clarity purposes. The figure below illustrates the instances of the modeling constructs for the Static View and the instance of the `Static Semantic Validation` testing activity view type.

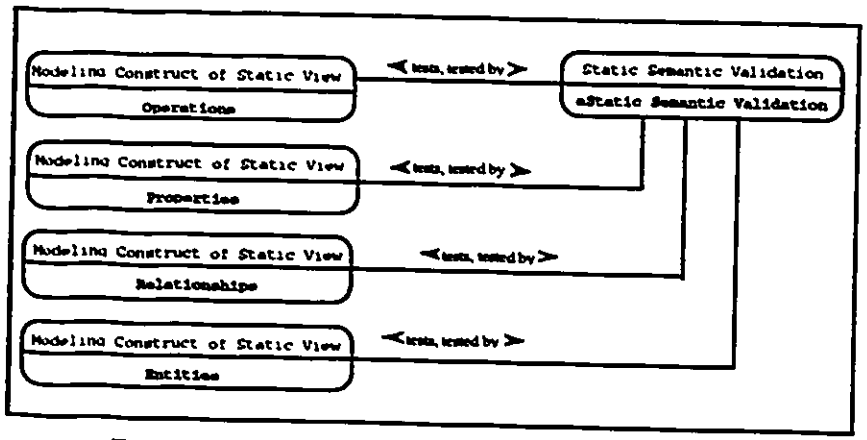


Figure 5.10 Instance Diagram for Fig. 5.9

The figure below illustrates the translation of the constructs of the Domain/Subclass Configuration Model and the Class Configuration Model into the constructs of the Static View.

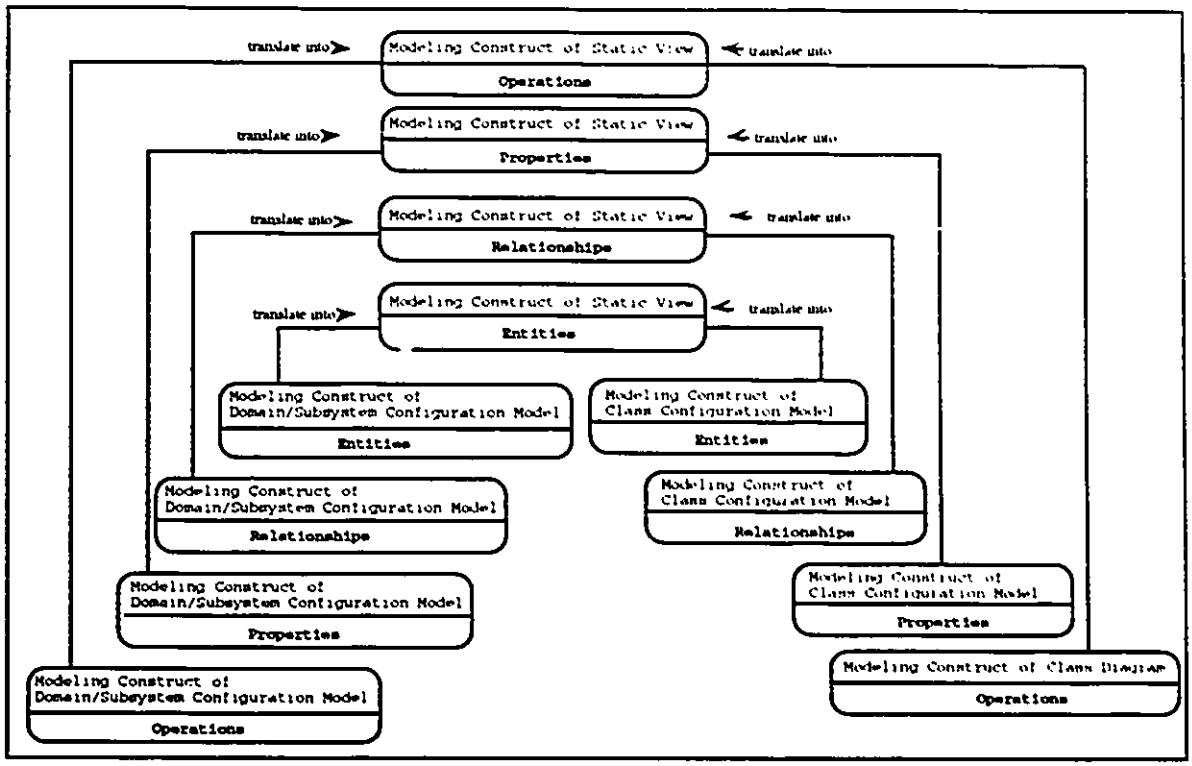


Figure 5.10 Instance Diagram for Fig. 5.9 (cont.)

The figure in the next page illustrates the translation of the constructs of the Domain Chart notation into the constructs of the Domain/Subclass Configuration Model as well as the indirect testing of the latter constructs by the testing activity view type: `Static Semantic Validation`. Note that the Domain Chart notation does not provide the semantics for an operation and a property, thus these are not translated.

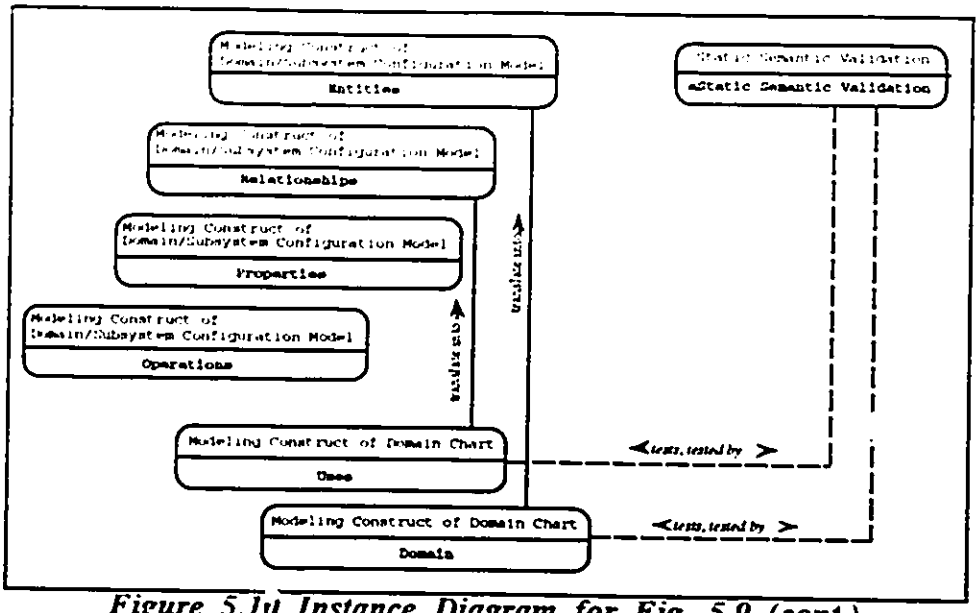


Figure 5.10 Instance Diagram for Fig. 5.9 (cont.)

The figure below illustrates the translation of the constructs of the Class Diagram notation into the constructs of the Class Configuration Model as well as the indirect testing of the latter constructs by the testing activity view type: `Static Semantic Validation`.

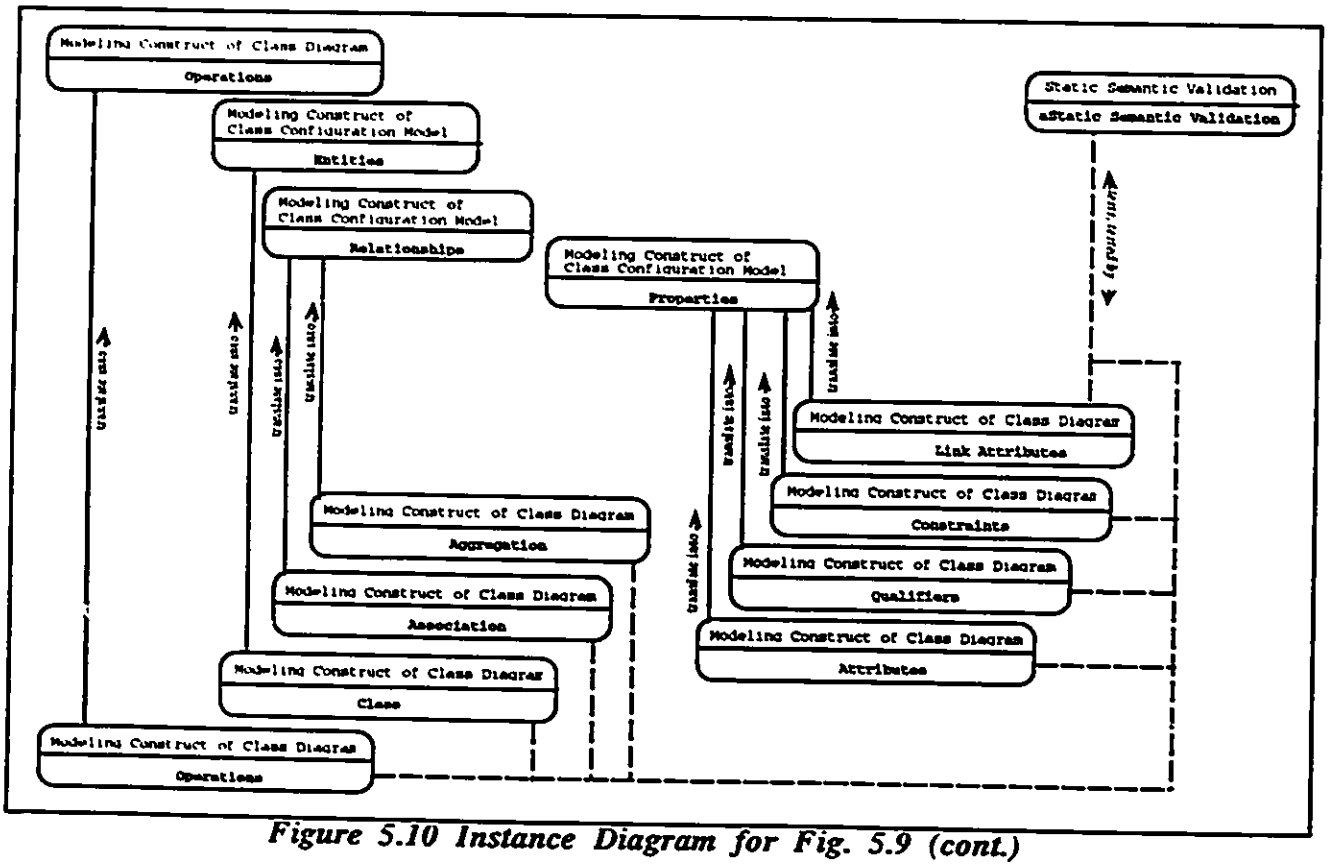


Figure 5.10 Instance Diagram for Fig. 5.9 (cont.)

Chapter 6 Verification Testing Activity Types

6.1 Introduction

As discussed in section 5.3.1 Verification aims at determining whether Notations used at phase *n* of a given iteration are (1) syntactically correct and (2) semantically complete and correct with respect to other Notations at phase *n* of the same iteration and whether these Notations are correct in themselves. Verification aims at providing Horizontal Traceability described in 3.2.2.3.1. We will discuss syntactic and semantic verification.

Verification testing activities should occur after each development phase, before moving to the next development phase.

6.2 Syntactic Verification: testing for syntactic correctness

Syntactic Verification aims at testing that the statement generated by a notation conform to its syntax. It is the only testing activity type that depends on the syntax of the notation being tested.

Referring to Fig. 3.1, in Chapter 3, we say that Syntactic Verification aims at verifying that Modeling Constructs of notations used to express the semantics of models

(1) have the correct Representation AND
(2) are legally combined according to Rules

Whether captured formally (e.g. BNF form) or not, all notations have a syntax.

Tools that allow for the graphical representation of notations inherently test for syntactic correctness by enforcing specific representations for modeling constructs and not allowing certain sequences of representations. This is done at entry-time.

For example, the graphical editor of the Case tool SDT [54] is used to create and edit SDL [12] system specifications and descriptions in the graphical form of SDL, thus enforcing specific representations for its modeling constructs (e.g. input message, output message, state, etc.). Also, the representation for an input message cannot be directly connected to the representation of an output message, thus enforcing a rule for legally combining modeling constructs. The editor does local syntax checks at entry time.

Another example of syntactic correctness in CASE tools is the Argos CASE tool [55]. Argos enforces specific representations for its modeling constructs at entry time (e.g. classes are drawn as rectangles, inheritance is depicted as a directed double line from subclass to superclass, derived associations are represented in italics, class operations and operations follow a strict Smalltalk like syntax). The tool also verifies at entry time that modeling constructs are legally combined, for example, a class cannot inherit itself.

Thus the process of Syntactic Verification can be automatic (i.e. at entry time as in the case CASE tools) or manual, if no tool is available. The latter must be performed to communicate the right semantics.

6.3 Semantic Verification: testing for semantic completeness and semantic correctness

We can further divide this section into Requirements Verification and Development Verification.

6.3.1 Requirements Capture Verification

As described in section 5.2.1, the verification process for the Requirements Capture phase is not the focus of the work presented, therefore we briefly introduce it. This activity fall under Specification Based testing techniques discussed in that section.

Text and notations that fall under the Scenario Model (discussed in section 5.3.3.1) are usually used to capture requirements. This forms a specification document.

Text can capture different aspects of the application to be developed (e.g. performance, data, architecture, behavior, etc.). Notations that fall under the Scenario model, provide aspects of the Communication View, since every use of the system can be viewed as communication between the User and the System.

Testing semantic completeness and correctness is highly coupled to the type of specification used to capture requirements. Specification Based testing techniques, aim at detecting incomplete, inconsistent and misleading requirements. An example is illustrated in [51]. When requirements are capture as notations in the Scenario Model, we must ensure that Uses Cases [18] or Scenarios [14] do not invalidate (correctness) each other they are complete with respect to the system or application to be modeled.

Overall, assessing Use Cases and Scenarios for their completeness and correctness, still remains a topic of research.

6.3.2 Development[22] Verification

As described in Chapter 4, models used at different phases of the Generic Object-Oriented Software Development Methodology provide different semantic (views) of the application to be developed. The four basic views are: Static, Communication, Dynamic and Processing.

These Views are provided at different levels of abstraction (i.e. System, Class/Object and Class/Object Internal) as described in Fig. 5.1.

Semantic Verification aims at verifying that the semantics provided by Views is complete and correct with respect to the semantics of **other** Views used at the **same** phase of the Generic Object-Oriented Development Methodology. The semantics of Views is described in terms of generic modeling constructs as described in Chapter 4. Semantic Verification will be defined in terms of these generic modeling constructs. These were defined in Chapter 4.

Static View: Entities, Relationships, Properties, Operations
Communication View: Entities, Messages, Ordering of messages
Dynamic View: States, Events, Actions, Guards
Processing View: Process, Data, Control flow and Data Flow

Fig. 6.1 illustrates the aim of Semantic Verification. We will discuss each link between Views separately.

Testing activity view types (i.e. links between Views in Fig. 6.1) should take place at every phase of development where the Views are offered. For example, at the Requirements Modeling phase the Communication and Static Views are provided as illustrated in Fig. 5.1, therefore, only `Static/Communication Semantic Verification` should take place.

[22]Note that here Development includes the Requirements Modeling phase.

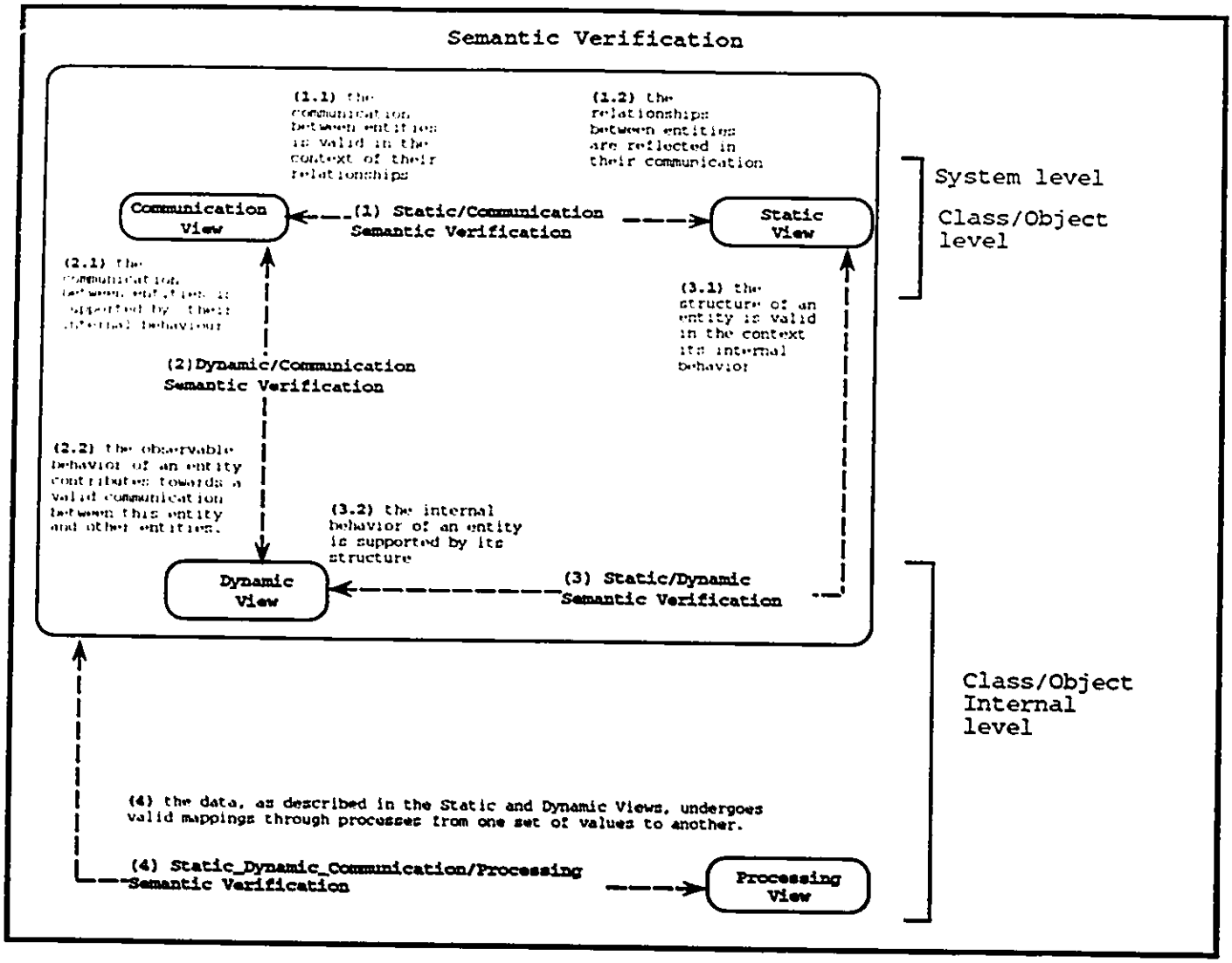


Figure 6.1 Semantic Verification

6.3.2.1 Static/Communication Semantic Verification

Static/Communication Semantic Verification should occur at the levels of abstraction in software development where these Views are offered (i.e. System and Class/Object level).

The purpose of Static/Communication Semantic Verification is to ensure the following[23]:

- **1- the Communication View is *correct* with respect to the Static View**

This is labeled as (1.1) in Fig. 6.1. The aim is to verify that the communication (i.e. Messages exchanged) between Entities is valid in the context of their Relationships. By "valid", we understand that:

> **For every Message sent from Entity X to Entity Y (i.e. Operation invocation of Y by X), Entity X and Entity Y must exit in the Static View and there must be at least one Relationship (direct or indirect[24]) between these two Entities (i.e. Entity X must have visibility over Entity Y via a direct or indirect Relationship).**

The latter corresponds to the coverage criteria for the Communication View in Static/Communication Semantic Verification.

-2- the Static View is *correct* with respect to the Communication View.

This is labeled as (1.2) in Fig. 6.1. The aim is to verify that direct and indirect Relationships between Entities are reflected in their communication. By "reflected", we understand that:

> **For every two Entities, there is a set of Operations that participate in a valid Message exchange to satisfy the Relationships between these two Entities.**

---

[23]In the discussion that follows, the generic modeling constructs of Views start with a capital letter in order to identify them better.

[24]Two Entities can have indirect visibility (i.e. Relationship) over one another; for example, class A can obtain a dynamic reference to class B as part of one of the parameters of a method in class A; another example, is when an Entity has an inherited Relationship.

The latter corresponds to the coverage criteria for the Static View in Static/Communication Semantic Verification.

The coverage criteria of the Static and Communication Views described above ensure *completeness* of these Views in Static/Communication Semantic Verification.

The basic idea is that a Relationship is described (implemented) as a set of Operations on the participating Entities. The order between these Operations should be captured in the Communication View as Message exchanges and Ordering of these Messages.

Examples of incorrect Communication View in Static/Communication Semantic Verification

(1) When a Message is sent from Entity X to Entity Y, and X has no visibility over Y. In this case, either the Communication View should remove this specific communication or the visibility between the two Entities must be added to the Static View.

(2) When a Message is not sent from Entity X to Entity Y, but X has visibility over Y. In this case, either the Communication View should add this specific communication or the visibility between the two Entities should be removed from the Static View).

Examples of incorrect Static View in Static/Communication Semantic Verification

(1) When there are Operations in Entities X and Y that satisfy a Relationship between the two Entities, but the Communication View does not capture the Message exchange and Ordering to satisfy the Relationship. In this case, either the communication to satisfy the Relationship should be added to the Communication View or the Relationship and its Operations should be removed from the Static View.

(2) When there are not sufficient Operations in Entities X and Y to satisfy a Relationship between the two Entities, but the Communication View captures the Message exchange and Ordering to satisfy the Relationship. In this case, either the communication to satisfy the Relationship should be removed from the Communication View or Operations should be added to Entities X and Y in the Static View to satisfy the Relationship.

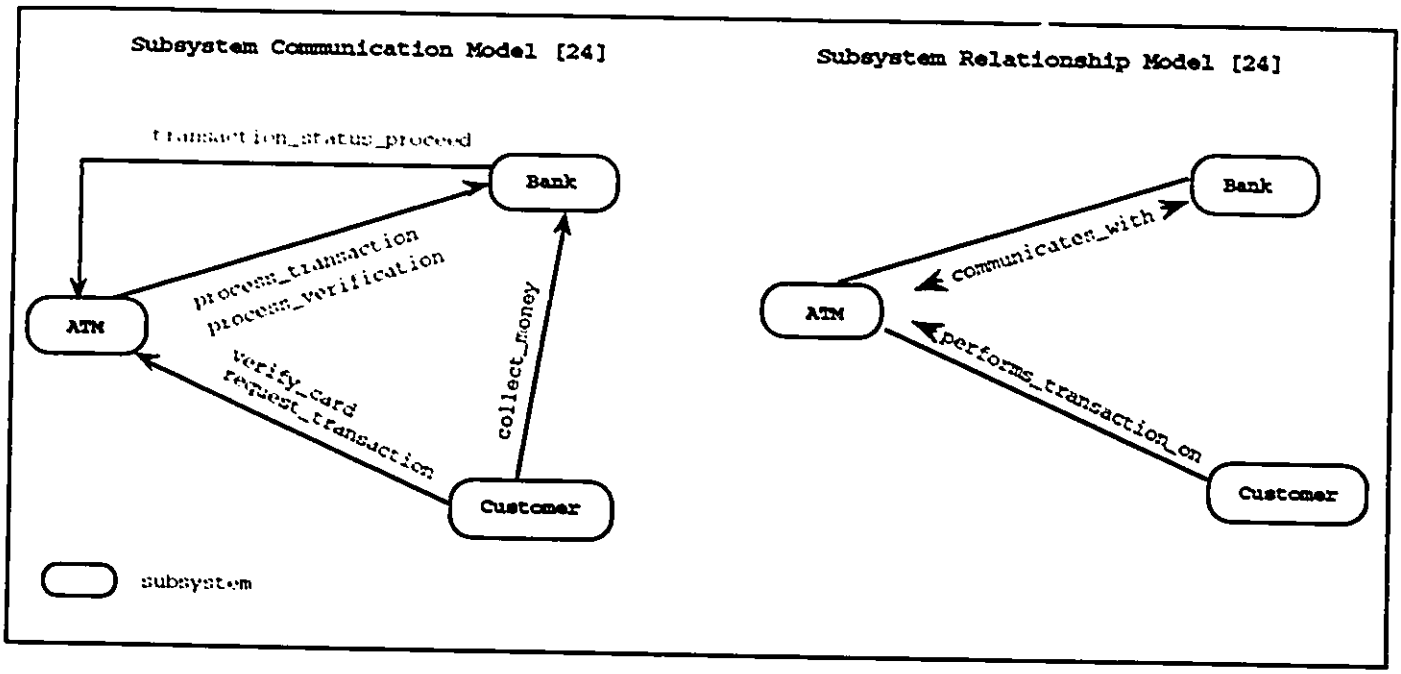
Notations that fall under the Contract Model as described in Chapter 4, provide a framework for identifying completeness and correctness of the Static and Communication Views, since the Model glues these Views. The Model describes the collaborations between Entities in terms of operations implemented by each Entity and descriptions of their interactions. Static/Communication Semantic Verification consists in ensuring that the Contract can be supported by both Views. More specifically, every collaboration between two Entities in a Contract specification must be supported by the Static and Communications Views (i.e. the description of each collaboration should be translatable to valid Message exchanges of the Communication View and expressed in terms of existing and valid Operations of the Static View).

Example 1, illustrates an example of Static/Communication Semantic Verification The example illustrates the System level of abstraction with the use of a Contract Model notation; the same technique is applicable for the Class/Object level of abstraction but instead of Subsystems, Semantic and Solution classes are used.

Example 1: Static/Communication Semantic Verification in the System level of abstraction

In Chapter 4, we identified notations (e.g. Subsystem Communication Model [24]) used to express the Communication View at a System level of abstraction. We also identified notations (e.g. Subsystem Relationship Model [24]) used to express the Static View at the System level of abstraction.

Consider, the simple example below. The Subsystem Relationship Model on the right shows the inter subsystem relationships for a given Domain. The Subsystem Communication Model on the left shows the messages between Subsystems for that same Domain.



Static/Communication Semantic Verification in the example above should verify the following:

- 1- the communication between entities is valid in the context of their relationships

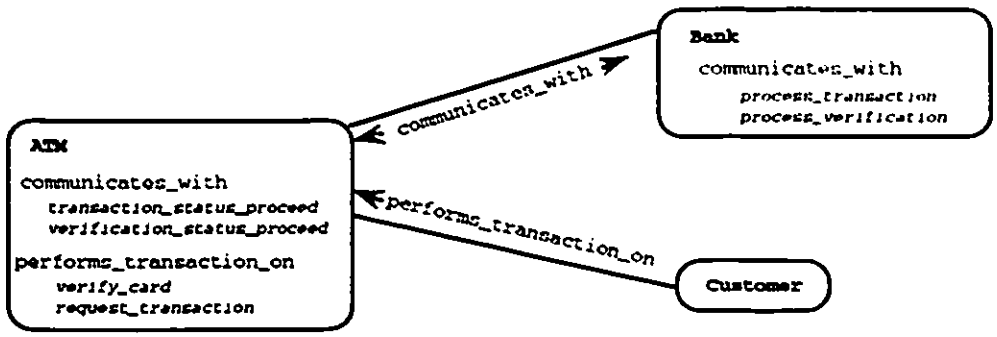
If there is a message exchanged between two subsystems, there must be a relationship between these two subsystems (i.e. one subsystem must have direct or indirect visibility over the other via a relationship). For example, the `process_transaction` message between `ATM` and `Bank`, is valid since the latter two subsystems are connected through the `communicates_with` relationship. On the other hand the `collect_money` message is not valid in the context of our example since subsystem `Customer` does not have visibility over subsystem `Bank` as indicated in the Subsystem Relationship Model on the right. If is determined that message `collect_money` should exist between the two subsystems, then either a direct relationship should be added between subsystems `Bank` and `Customer` in the Subsystem Relationship Model on the right or an indirect relationship should be indicated between the Subsystems. The latter would be the case if `Customer` obtains a dynamic reference to `Bank` from `ATM` as part of a parameter of one of `Customer`'s abstract operations[25]. Otherwise, message `collect_money` should be removed from the Subsystem Communication Model on the left. For the sake of this example assume that the latter is the action to take.

In the example above we have addressed the correctness attribute of the Communication View.

-2- the relationships between entities are reflected in their communication

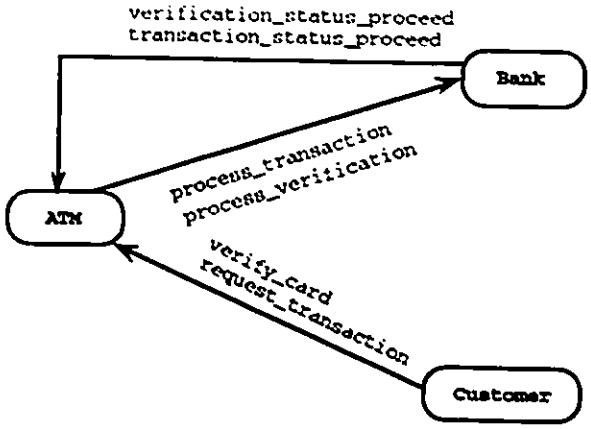
With the current Subsystem Relationship Model notation, we cannot verify this important activity. The notation should be augmented with abstract operations that satisfy every relationship. This is illustrated below.

augmented Subsystem Relationship Model notation



[25] An abstract operation is an operation of a Subsystem. The abstract operation when refined, may be transformed to one or more operations of a class(es) in that subsystem, one or more classes etc.

Given the 'augmented Subsystem Relationships Model', we can now verify that the relationships between entities are reflected in their communication (e.g. the relationship `communicates_with` between subsystems `ATM` and `Bank` requires a certain communication between these subsystems; the abstract operations involved in the relationship are `transaction_status_proceed` and `verification_status_proceed` for `ATM` and `process_transaction` and `process_verification` for `Bank`). The latter operations should be reflected in the Communication View. We notice that they are not. The abstract operation `verification_status_proceed` is not reflected in the Subsystem Communication Model. Thus we found an incorrectness in the Static View. If it is determined, that the abstract operation is not part of the message exchange between `ATM` and `Bank` to satisfy the relation `communicates_with`, then it should be removed from subsystem `ATM` in the 'augmented Subsystem Relationship Model'. Otherwise, it should be added to the Subsystem Communication Model notation as shown below. For the sake of this example assume that the latter is the action to take. The Subsystem Communication Model



So far, we can say the Static and Communication Views are complete and correct with respect to each other.

We will briefly illustrate the concept of a function. The `perform_transaction_on` relationship between subsystems `ATM` and `Customer`, identifies `Customer` as a Client and `ATM` as a Server. This is because of the direction of the relationship. Thus, the relationship becomes the function that a Server must provide to its Client. The process is thus to validate this function in terms of the operations exchanged between the Client and Server in order to implement the relationship.

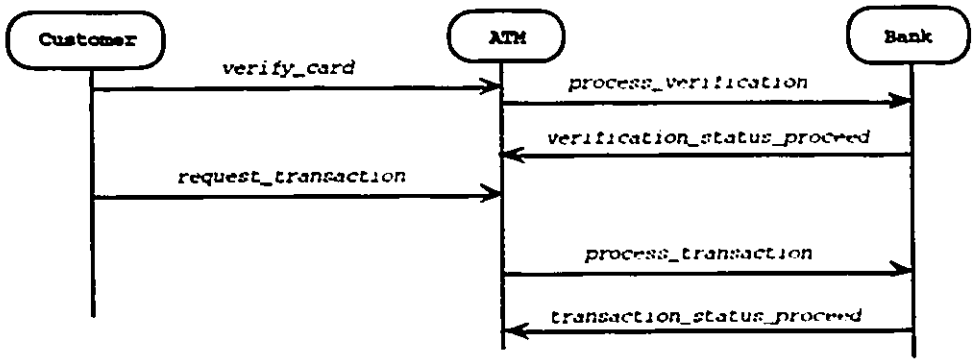
Notations that fall under the Contract Model as described in Chapter 4, also provide a framework for identifying correctness for the two Views.

In our example, subsystem Bank has two responsibilities: to verify a card and to process a transaction. We can define this, as part of a contract as illustrated below.

```
Contract 1:    Process a transaction based on card
               verification
Server:        Bank
Clients:       ATM
Description:   This contract specifies that a transaction
               can occur only when the customer card has
               been verified.
               In Bank:
               process_verification returns verification_status_proceed,
               process_transaction returns transaction_status_proceed.
```

Now Static/Communication Semantic Verification consists in ensuring that the contract can be supported by both views. For this, we need to augment the Subsystem Communication Model notation. What is missing, is the ordering of messages (responsibilities) required to satisfy a specific contract. We can do this by augmenting the notation as an Even Trace [14] or an Interaction Diagram [18,22]. This is shown below.

augmented Subsystem Communication Model notation



Therefore, from the 'augmented Subsystem Communication Model', we can detect whether the message exchange is complete and correct with the respect to the contract (i.e. whether the description section of the contract is translatable to message exchanges of the 'augmented Subsystem Communication Model').

From the 'augmented Subsystem Relationship Model', we can detect whether the complete and correct set of operations are offered to satisfy the Contract (i.e. whether the Contract is expressed in terms of existing and valid operations of the 'augmented Subsystem Relationship Model').

6.3.2.2 Dynamic/Communication Semantic Verification

Dynamic/Communication Semantic Verification should occur at the levels of abstraction in software development where these Views are offered (i.e. Class Object level and Class/Object internal level).

The purpose of this testing activity is to ensure the following:

- 1- the Communication View is *correct* with respect to the Dynamic View

This is labeled as (2.1) in Fig. 6.1. The aim is to verify that the communication between Entities is supported by their internal behavior. By "supported", we understand that:

> **For every communication between two Entities (i.e. Message exchanges), the internal behavior of the receiver Entity, must eventually generate a Message (i.e. contain an Action that will cause an operation invocation of another Entity) as described by the communication.**

The latter corresponds to the coverage criteria for the Communication View in Dynamic/Communication Semantic Verification

Note that we see the internal behavior of an Entity only in terms of its Events (Operations invoked by other Entities) and Actions (Operations invoked by the Entity).

-2- the Dynamic View is *correct* with respect to the Communication View

This is labeled as (2.2) in Fig. 6.1. The aim is to verify that the observable behavior of an entity contributes towards a valid communication between itself and other entities. By "contributes", we understand that:

> **All internal behavior that can be observed from the outside of the Entity (Actions: Operations invoked by the Entity and Event: Operations of the Entity invoked by other Entities) must be present in the Communication View.**

The latter corresponds to the coverage criteria for Dynamic View in Dynamic/Communication Semantic Verification.

The coverage criteria of the Dynamic and Communication Views described above ensure *completeness* of these Views in Dynamic/Communication Semantic Verification.

Examples of incorrect Communication View in Dynamic/Communication Semantic Verification

(1) When a Message is sent from Entity X to Entity Y and Y is expected to respond by sending message M to Entity Z, but the internal behavior of Y does not support the sending (i.e. invocation) of message M as part of one of its Actions. In this case, either the Communication View should remove this specific communication (i.e. the sending message M to Entity Z) or an Action should be added to account for sending message M in the internal behavior of Y.

(2) When a Message M is not sent from Entity X to Entity Y but an Action in the internal behavior of X supports the invocation of M. In this case, either the Communication View should add this specific communication (i.e. sending message M to Entity Y) or the Action causing the invocation of M should be removed from the internal behavior of X.

Examples of incorrect Dynamic View in Dynamic/Communication Semantic Verification

(1) When an Event and/or Action of an Entity is not present in a communication. In this case, either the Event (and necessary Guards) and/or Action should be removed from the internal behavior of the Entity in the Dynamic View or the communication should be added to the Communication View.

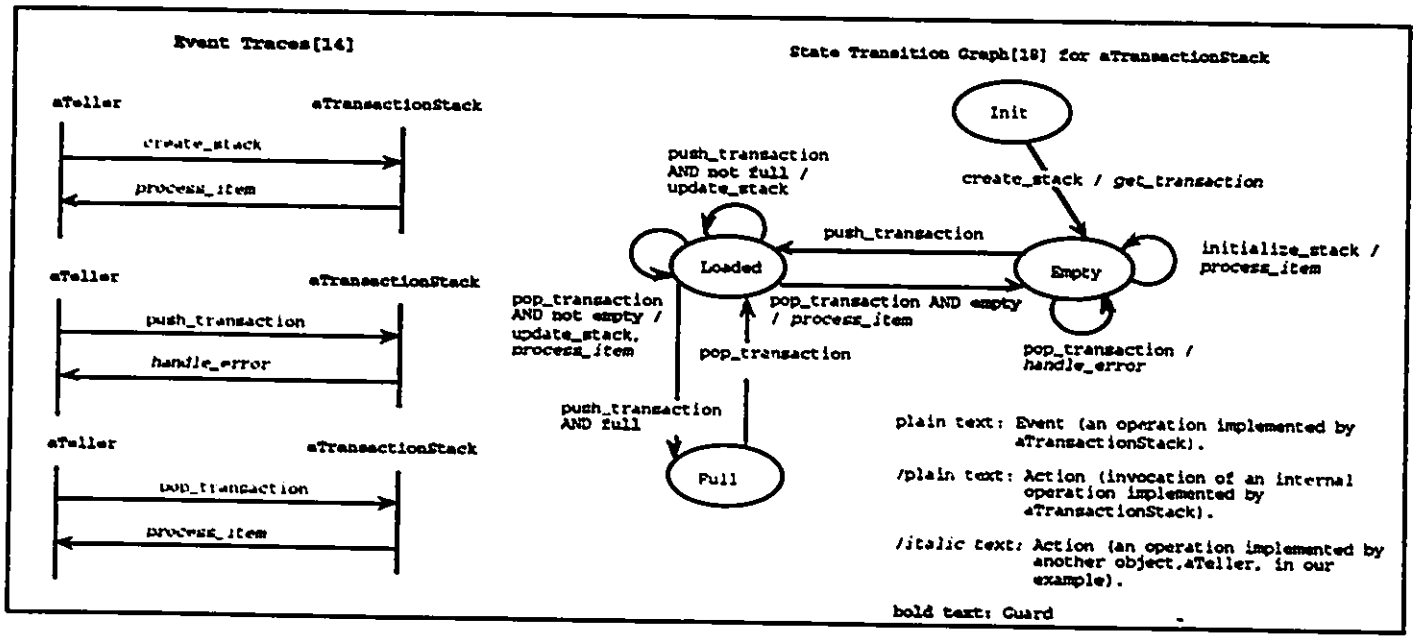
(2) When an Event and/or Action of an Entity is present in a communication as Message exchanges, but the internal behavior of an Entity does not account for this. In this case, either the Event (and corresponding Guards) and/or Action should be added to the internal behavior or the Entity in the Dynamic View or the communication should be removed from the Communication View.

Example 2 illustrates an example of Dynamic/Communication Semantic Verification.

Notations that fall under the Event Manipulation Model as described in Chapter 4, provide a framework for identifying completeness and correctness of the Dynamic and Communication Views, since the Model glues these Views. The Model describes both the internal behavior of an Entity (Dynamic View) and the interactions between Entities as message exchanges (Communication View). In the Event Manipulation Model, Dynamic/ Communication Semantic Verification consists in ensuring that every path in the Model can be decomposed into communications that are present in the Communication View and that each object in the communication can be further decomposed into Events and Actions of its internal behavior as described in the Dynamic View.

Example 2: Dynamic/Communication Semantic Verification

Assume that `aTeller` and `aTransactionStack` are instances of classes `Teller` and `TransactionStack` respectively. The Event Traces[14] on the left depicts the Communication View and the State Transition Graph[18] on the right depicts the Dynamic View for the object `aTransaction Stack`.



Dynamic/Communication Semantic Verification in the example above should verify the following:

- **I-** the communication between Entities is supported by their internal behavior

The internal behavior of the receiver of a message in the Communication View, must eventually generate a message as described by the Communication View. To verify this, we need to analyze different event/state/action (i.e. input/state/output) combinations in the State Transition Diagram. Well established testing techniques for Finite State Machine as described in [11], can be used.

In our example, the `create_stack` event (i.e. operation) can only be invoked in state `Init` which generates action `get_transaction` and not `process_item` as indicated in the first Event Trace. If it is determined that

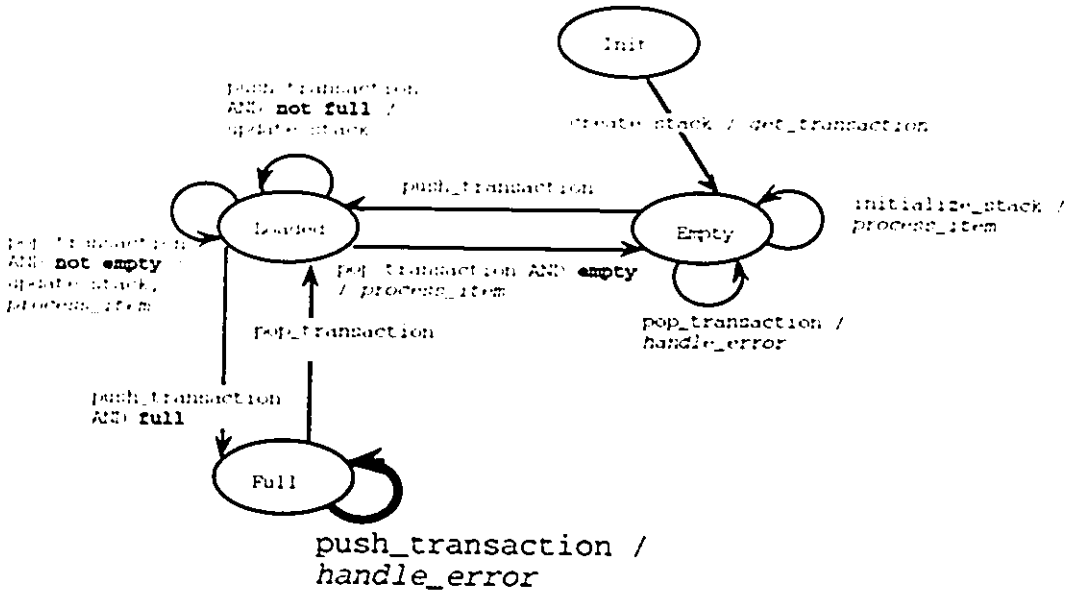
(1) the action of `aTransactionStack` upon reception of the `create_stack` event, should be `get_transaction`, then the first Event Trace should be updated with the latter event; or

(2) the action of `aTransactionStack`, upon reception of the `create_stack` event, should be `process_item`, then State Transition Graph should be updated accordingly; or

(3) the action of `aTransactionStack` should be `process_item`, but the event that generates this action is not be `create_stack`, then the first Event Trace should be updated with events: `initialize_stack` or `pop_transaction` as indicated in the State Transition Graph.

For the sake of this example assume that the first case is the action to take.

Continuing with our example, we observe that the second Event Trace is not supported by the State Transition Graph. If it is determine that this Event Trace is valid, then the State Transition Diagram should be updated as indicated below in bold:



The third event trace is supported by the State Transition Graph in state Loaded.

Note that at this point, we can say that the Event Traces (Communication View) are correct with respect to the State Transition Diagram (Dynamic View).

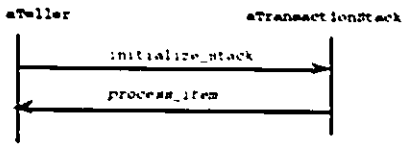
- 2- the observable internal behavior of an Entity contributes towards a valid communication

All internal behavior that can be observed from outside (i.e. events and actions that correspond to the invocation of an operation in another object) must be present in the Communication View. As said in -1-, well established testing techniques for Finite State Machine as described in [11], can be used to analyze different input/state/output combinations in the State Transition Graph.

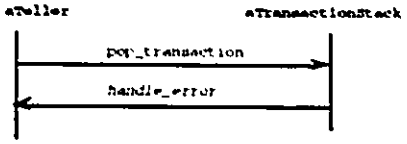
The intend is that every state in the State Transition Diagram should be analyzed for its event/state/action and that Event Traces should be updated as required by these combinations.

As said earlier, at this point, the State Transition Graph is seen only in terms of its externally observable behavior, therefore internal operation calls such as `update_stack`, cannot be observed and therefore should not be part of the Communication View.

In our example, we observe that the `initialize_stack` event causes the `process_item` action in state `Empty`; this is not reflected in the event traces an therefore should be added as follows:



Finally, we observe that the `pop_transaction` event causes the `handle_error` action in state `Empty`; this is not reflected in the event traces an therefore should be added as follows:



Note that at this point, we say that the State Transition Graph is correct with respect to the Event Traces.

Furthermore, since every Event Trace can be traced to a path in the State Transition Diagram and every observable event/action pair in the State Transition Diagram is depicted in the Event Traces; we can say the two views are complete with respect to each other.

6.3.2.3 Static/Dynamic Semantic Verification

Static/Dynamic Semantic Verification should occur at the levels of abstraction in software development where these Views are offered (i.e. Class Object and Class/Object internal levels).

In the discussion that follows, we make no assumption about the type of state machine notation that provides the Dynamic View (i.e. Actions could be placed in a transition and/or in a state).

The purpose of this testing activity is to ensure the following:

- **1- The Static View is *correct* with respect to the Dynamic View**

This is labeled as `(3.1)` in Fig. 6.1. The aim is to verify that the structure of an Entity is valid in the context of its internal behavior. By valid, we understand that:

(1) Every Operation of an Entity X should be present in its Dynamic View, either as

- **an Event in a transition (in the case of operations of X that can be invoked by other Entities or in the case of internal operations[26] of X that can be invoked by only X) and/or**
- **an Action in a transition or in a State (in the case of internal operations)**

(2) Every Property (i.e. attribute[27]) of an Entity X should be accounted for in its Dynamic View, either as part of :

- **a State and/or**
- **an Action in a transition or in a State (as in the case of setting data values in a transition) and/or**
- **a Guard**

---

[26]Operations that other Entities cannot have access to.

[27]By attribute, we mean either a data value or an attribute that corresponds to a pointer to a class.

(3) Every Relationship is accounted for in the Dynamic View, as part of the Events of X that satisfy the X side of the Relationship.

The latter corresponds to the coverage criteria for the Static View in Static/Dynamic Semantic Verification.

-2- **The Dynamic View is *correct* with respect to the Static View**

This is labeled as (3.2) in Fig. 6.1. The aim is to verify that the internal behavior of an entity is supported by its structure. By supported, we understand that:

(1) Every Event of Entity X should be present as an Operation in the Static View.

(2) Every State of Entity X must be expressed in terms of one or more Properties (i.e. attributes[28]) of the Static View.

(3) Every Guard (i.e. Boolean expression) of Entity X is expressed in terms of Properties in the Static View.

(4) Every Action of Entity X in a transition or in a State, that does not involve invoking an operation of another Entity, should be accounted for in its Static View, either as

- a Property (as in the case of setting data values) or
- as an internal Operation

The latter corresponds to the Coverage criteria for the Dynamic View in Static Dynamic Semantic Verification

The coverage criteria of the Static and Dynamic Views described above ensure *completeness* of these Views in Static/Dynamic Semantic Verification.

Example 3, illustrates an example of Static/Dynamic Semantic Verification. The reader is recommended to read this example to clarify the presentation of the testing activity type.

---

[28]By attribute, we mean either a data value or an attribute that corresponds to a pointer to a class.

Examples of incorrect Static View in Static/Dynamic Semantic Verification

(1) When an Event is the Dynamic View cannot be related to an Operation of the Static View. In this case, either the Dynamic View should remove the Event and affected States, Guards and Actions or the Static View should create a new Operation supporting the Event.

(2) When an Operation in the Static View cannot be related to an Event is the Dynamic View. In this case, either the Dynamic View should add the Event and corresponding States, Guards and Actions or the Static View should remove the Operation.

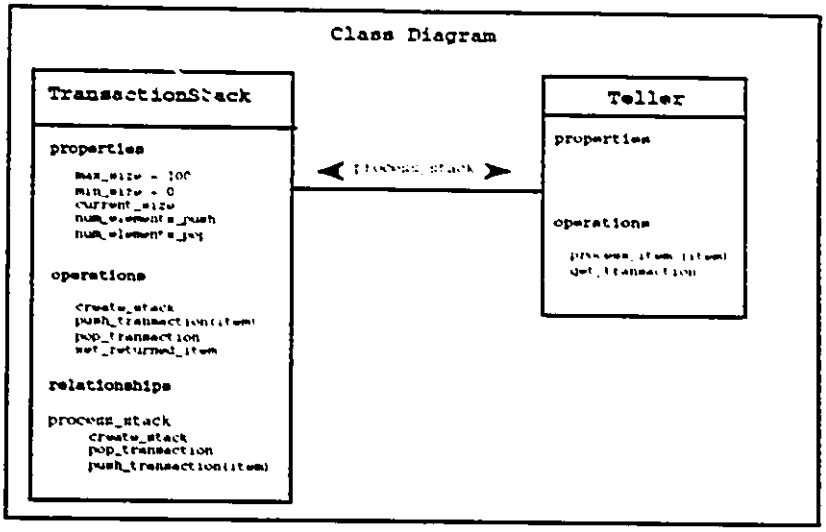
Examples of incorrect Dynamic View in Static/Dynamic Semantic Verification

(1) When a State in the Dynamic View cannot be related to Properties of the Static View. In this case, either the Dynamic View should remove the State and affected Events, Guards and Actions or the Static View should provide the Properties to arrive to the State.

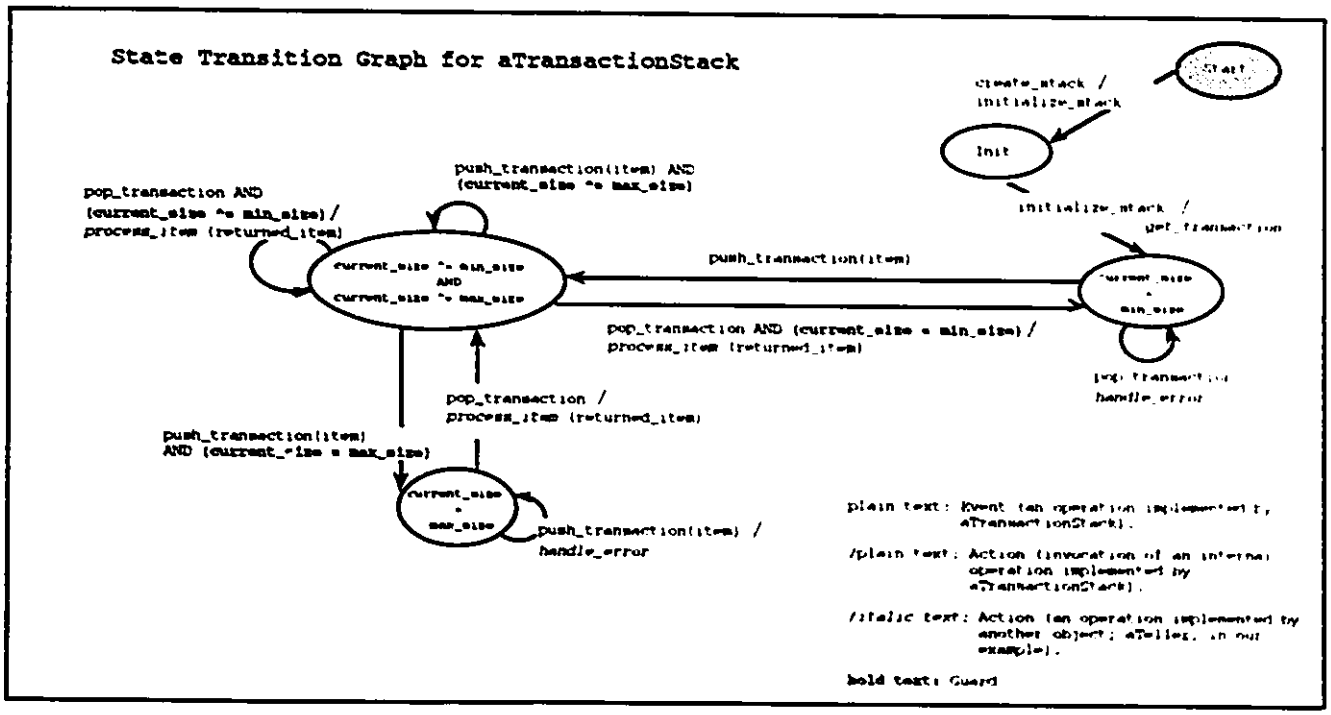
(2) When a Guard in the Dynamic View is not expressed in terms of Properties of the Static View. In this case, either the Dynamic View should remove the Guard and affected Events, States and Actions or the Static View should provide the Properties to express the Guard.

Example 3: Static/Dynamic Semantic Verification

The Class Diagram[14] below illustrates the Static View for classes `Teller` and `TransactionStack`; these classes are related via the `process_stack` relationship, which is satisfied by the operations indicated under `relationships` in both classes.



The Dynamic View of an instance of class `TransactionStack` is shown by the State Transition Graph[18] described below.



Static/Dynamic Semantic Verification in the example above should verify the following:

- I- the structure of an Entity is valid in the context of its internal behavior.

By valid we mean that

(1) **Every Operation of an Entity X should be present in its Dynamic View, either as**

- **an Event in a transition (in the case of operations of X that can be invoked by other Entities or in the case of internal operations of X that can be invoked by only X) and/or**
- **an Action in a transition or in a State (in the case of internal operations).**

We observe that operations `create_stack`, `push_transaction` and `pop_transation` are present in transitions of the State Transition Diagram but that operation `set_returned_item`, is not present. The operation should either be removed from the Class Diagram or accounted for (i.e. added) to the State Transition Diagram. Assume that the action to take is the latter and that `set_returned_item` is an internal operation. An updated State Transition Diagram is shown at the end of the example.

(2) **Every Property (i.e. attribute) of an Entity X should be accounted for in its Dynamic View, either as part of :**

- **a State and/or**
- **an Action in a transition or in a State (as in the case of setting data values in a transition) and/or**
- **a Guard**

We observe that properties `max_size`, `min_size` and `current_size` are present in states and in guards. However, properties `num_elements_push`, `num_elements_pop` are not present in the State Machine. The properties should either be removed from the Class Diagram or accounted for (i.e. added) to the State Transition Graph. Assume that the action to take is the latter and that the properties are accounted for as Actions by setting date values for these properties. An updated State Transition Diagram is shown at the end of the example.

(3) **Every Relationship is accounted for in the Dynamic View, as part of the Events of X that satisfy the X side of the Relationship.**

We observe that the three operations of `TransactionStack` that satisfy the `process_stack` relationship are indeed invoked as shown by events in the State Transition Graph.

-2- the internal behavior of an entity is supported by its structure

By supported, we mean that

(1) **Every Event of Entity X should be present as an Operation in the Static View.**

We observe that the event `initialize_stack` from state `Init` is not present in the Class Diagram. The event should either be removed from the State Transition Graph or accounted for (i.e. added) to the Class Diagram. Assume that the action to take is the latter and that the operation is an internal operation. An updated Class Diagram is shown at the end of the example.

(2) **Every State of Entity X must be expressed in terms of one or more Properties (i.e. attributes[29]) of the Static View.**

We observe that state `Init` is not expressed in terms of the properties the Class Diagram. The state should either be removed from the State Transition Graph or accounted for (i.e. expressed in terms of properties) in the Class Diagram. Assume that the action to take is the latter and that the `Init` state can be expressed as (`current_size = min_size; num_elements_push = 0; num_elements_pop = 0`). An updated Class Diagram is shown at the end of the example.

(3) **Every Guard (i.e. Boolean expression) of Entity X is expressed in terms of Properties in the Static View.**

We observe that all guards in the State Transition Graph are expressed in terms of the properties the Class Diagram.

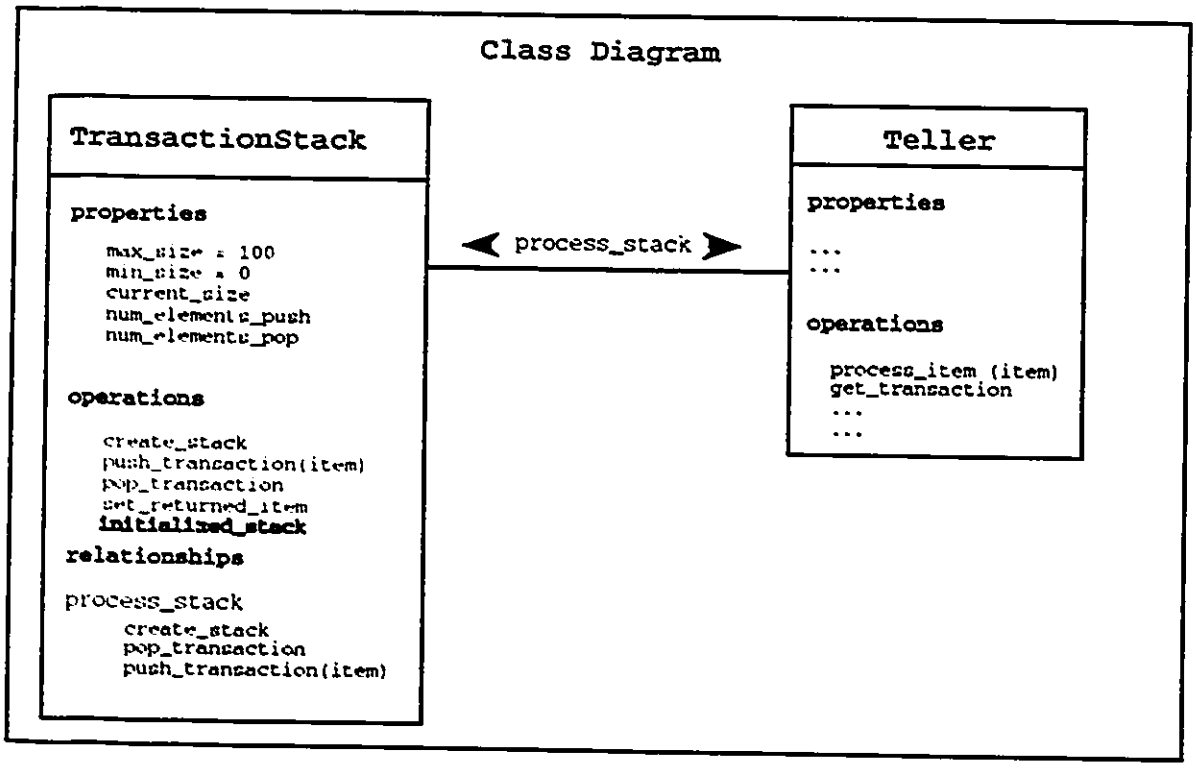
[29]By attribute, we mean either a data value or an attribute that corresponds to a pointer to a class.

(4) **Every Action of Entity X in a transition or in a State, that does not involve invoking an operation of another Entity, should be accounted for in its Static View, either as**

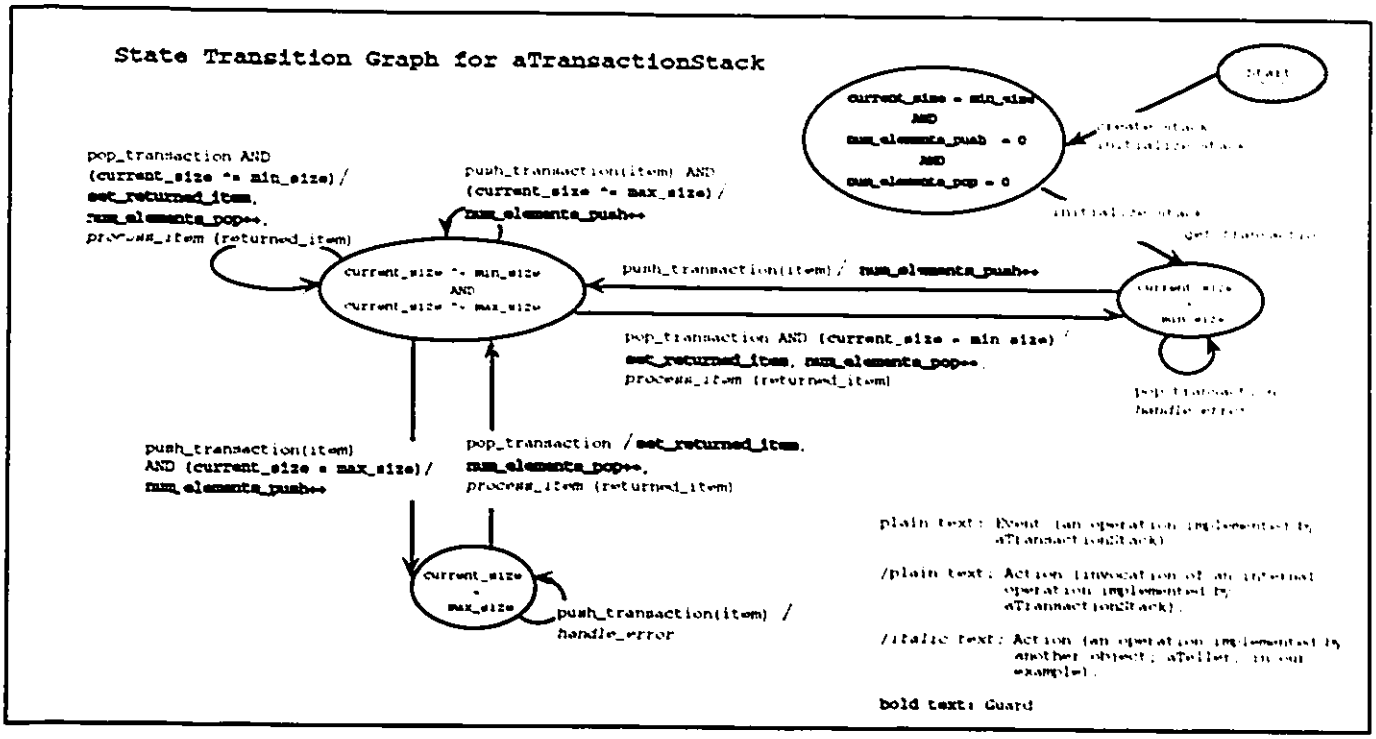
- **a Property (as in the case of setting data values) or**
- **as an internal Operation**

We observe that actions `num_elements_pop++` and `num_elements_push++` in the State Transition Graph are expressed in terms of properties of the Class Diagrams. We also observe that actions `initialize_stack` and `set_returned_item` (denoting internal operations) in the State Transition Graph are present in the Class Diagram.

The figure below illustrates the updated Class Diagram after Static/Dynamic Semantic Verification. Changes are shown in outlined text.



The figure below illustrates the updated State Transition Graph after Static/Dynamic Semantic Verification. Changes are shown in outlined text.



6.3.2.4 Static_Dynamic_Communication/Processing Semantic Verification

Static_Dynamic_Communication/Processing Semantic Verification should occur at the levels of abstraction in software development where these Views are offered (i.e. Class Object and Class/Object internal levels).

The Processing View (e.g. Data Flow Diagrams, DFDs) is currently undergoing controversy in the Object-Oriented world. "The idea was (and is) that you need to document not only structure but also function, in the mathematical sense of mapping from one set of values to another... However, I and others have found in practice that DFDs are not a very compact way to do this. Better ways include equations (for numerical things), before-and-after conditions, and in most cases a simple text description will suffice. You do need to specify the behavior of operations, but diagrams are probably not the best way. I think that the OO DFDs that I proposed more recently are a valid pedagogical device but probably not so useful in practice, especially in large systems".[30] " DFDs are not useless, they just fall short in a lot of cases. It's nice to have a richer notations when you want to make more detailed and rigorous statements about what you want... Use DFDs if they work for you, but don't stress too much if they don't show up in the OOA/D/P toolbox."[31] We agree with the latter point.

The Processing View is not feasible for large systems due to the level of detail it provides. The functional nature of operations should be described only for complex operations in large systems and not all operations.

In terms of verification, we should ensure that those specific functions are valid. Therefore, the aims of Static_Dynamic_Communication/Processing Semantic Verification is to ensure the validity of these transformation.

More specifically, the purpose of this testing activity is to ensure the following:

The data, as described in the Static and Dynamic Views, undergoes valid mappings through Processes from one set of values to another. (i.e. every Process, Data and Data flow is described in terms of Operations, Entities and Properties of the Static, Dynamic and Communication Views)

---

[30] James Rumbaugh, in email discussions from Rational 'otug@Rational.com'.

[31] Response to email on feedback on DFDs for object-oriented notations.

This is labeled as (4) in Fig. 6.1. The aim is to verify that data transformations are valid in the context of structure, behavior and communication. By valid, we understand that:

(1) Every Process is implemented as
- **one or more Operations in an Entity of the Static View. (these processes are usually labeled with the Entity name they belong to[32] or by placing the Process within the Entity itself[33]) and**
- **an Event in a transition or an Action in a transition or an Action in a State in the Dynamic View and**
- **a Message sent from one Entity to the next in the case of non-internal Operations in the Communication View**

(2) Every Data can be accounted for as
- **a Property in the Static View and**
- **a State and/or an Action in a transition and/or an Action in a State and/or a Guard**

(3) Every Data Flow can be traced to the transformation of a Property in the Static View. This is usually achieved by labeling the Data Flow with the name of the Property being changed by the Process.

The latter corresponds to the Coverage criteria for the Processing View This ensures the *completeness* and *correctness* of the Processing View.

---

[32]Action Data Flow Diagrams in Object LifeCycles Modeling the Word in States by Shlaer and Mellor.

[33]Object-Oriented Data Flow Diagrams in OMT: The functional Model by Rumbaugh.

Examples of incorrect Processing View in Static_Dynamic_Communication/Processing Semantic Verification

(1) When a Process cannot be traced to an Operation in the Static View. In this case, either the Processing View should remove the Process, or the Process should be expressed in terms of valid Operations of the Static View or the Static View should add the Operation and all other Views should be updated accordingly.

(2) When a Data cannot be accounted for in a State, Action in a transition, Action in a State or in a Guard in the Dynamic View. In this case, either the Processing View should remove this Data or express the Data Flow in terms of valid data of the Dynamic View or the Dynamic View should add the new data in a State, Action in a transition, Action in a State or in a Guard.

Notations that fall under the Class Structure Model provides aspects of the Processing View, since it describes operations, control flow and data flow, but does not describe the processing of data.

Currently the Object Interaction Diagram notation in Rumbaugh [14] provide a framework for identifying completeness and correctness of the Processing View since the Model glues the Communication and Processing Views. The Model describes both the processing of data within Entities as well as the communication between Entities.

Chapter 7 Validation and Incorporation Testing Activity Types

7.1 Introduction

The purpose of this section is to discuss Validation and Incorporation testing activity types. These are discuss in the same chapter due to their similarity (this becomes apparent through the chapter).

7.2 Validation testing activity types

Validation aims at validating the semantic transformations that notation can undergo. These transformations are usually applied when moving from one phase of development (e.g. Analysis) to a more detail one (e.g. Design). The aim is to provide Vertical Upwards and Vertical Downwards traceability, identified in section 3.2.2.3.1. The goal is to test the consistency of a View through different levels of abstraction as shown in Fig. 7.1.

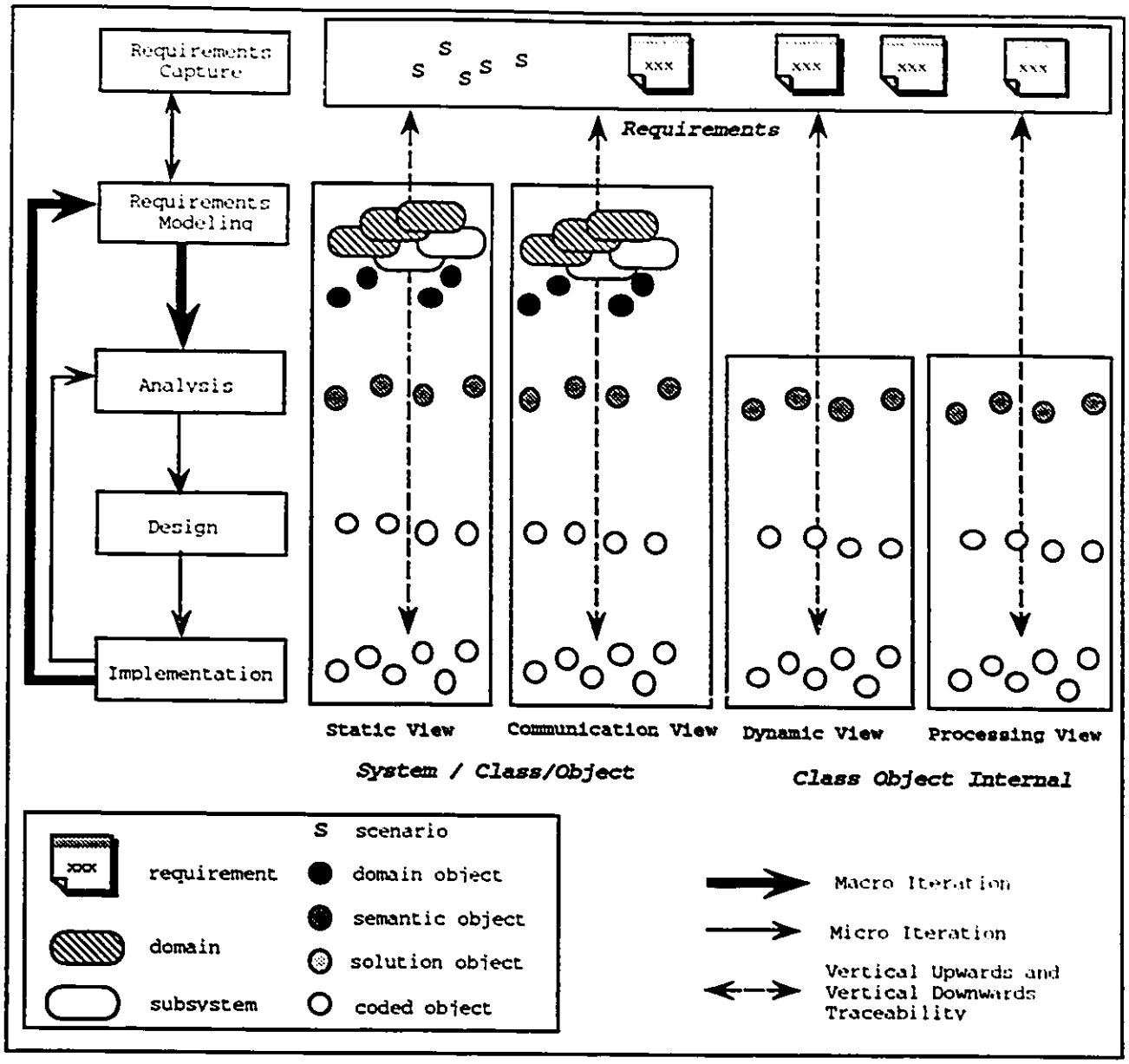


Figure 7.1 Semantic Validation

More specifically, we say that Semantic Validation aims at ensuring that

(1) every modeling construct of a notation at phase *n* of the Generic Object-Oriented Development Methodology is present or rationalized (as in the case that the modeling construct is deleted or changed) at phase *n+1*, and that

(2) every modeling construct at phase *n+1* of the Generic Object-Oriented Development Methodology is a valid transformation of a modeling construct at phase *n*.

Note that in the above definition

'every' deals with completeness and
'is present or rationalized' and 'is a valid transformation of' deals with correctness.

As stated in Chapter 3, section 3.2.2.3.2, the attributes: *completeness* and *correctness* were folded into *consistency* (i.e. consistency of a View includes its completeness and correctness). Therefore in the discussion that follows, when the attribute: consistency is used, the reader should remember that we mean both completeness and correctness.

Before getting to Semantic Validation, the following observations are made, which in turn mold the definition of Semantic Validation testing activity types.

Observation 1

In the previous section, when we discussed Verification, we mentioned that this testing activity type should take place at each phase, before moving to the next phase. The time when Validation should take place is not so straight forward and must be taken in the context of the iterative style of development.

We say that both aspects of the definition of Validation (i.e. **(1)** and **(2)** above) should take place after each development phase of an iteration except for *boundary* phases[34]. Checking for consistency at boundary phases must be split: aspect **(1)** should not take place until the goal of the iteration is achieved; aspect **(2)** can take place after each phase of an iteration.

[34]A boundary phase is a phase at which partitioning of the system for iteration takes place. In our Generic Object-Oriented Software Development methodology, the Analysis and Requirements Modeling phases are boundary phases, since they define the Micro and Macro-Iteration goals respectively.

Lets take a brief example to illustrate our observation.

In Fig. 7.2, there is no value in performing aspect **(1)** of `Requirements Modeling vs. Analysis Validation` of Analysis notations, until `time=3`. That is, aspect **(1)** of validation should not take place until the goal of the current Macro-Iteration is achieved. This occurs at `time=3`, in Fig. 7.2. Note that at this point, the next Macro-Iteration can start. The point is that before `time=3`, Analysis notations are incomplete with respect to Requirements Modeling notations. Therefore testing for consistency, which includes completeness, of Requirements Modeling notations in the current Analysis notations will definitely fail. Aspect **(2)** of the definition of validation can and should take place after each Analysis phase of an iteration since we are testing the consistency of Analysis notations with respect to Requirements Modeling notations (and not vice-versa as in the previous case).

The same discussion is applicable to `Requirements Capture vs. Requirements Modeling Validation`. Aspect **(2)** of the definition of validation can and should take place after each Requirements Modeling phase of an iteration. Aspect **(1)** should only take place at `time=n`, time at which there are no more Macro-Iterations left.

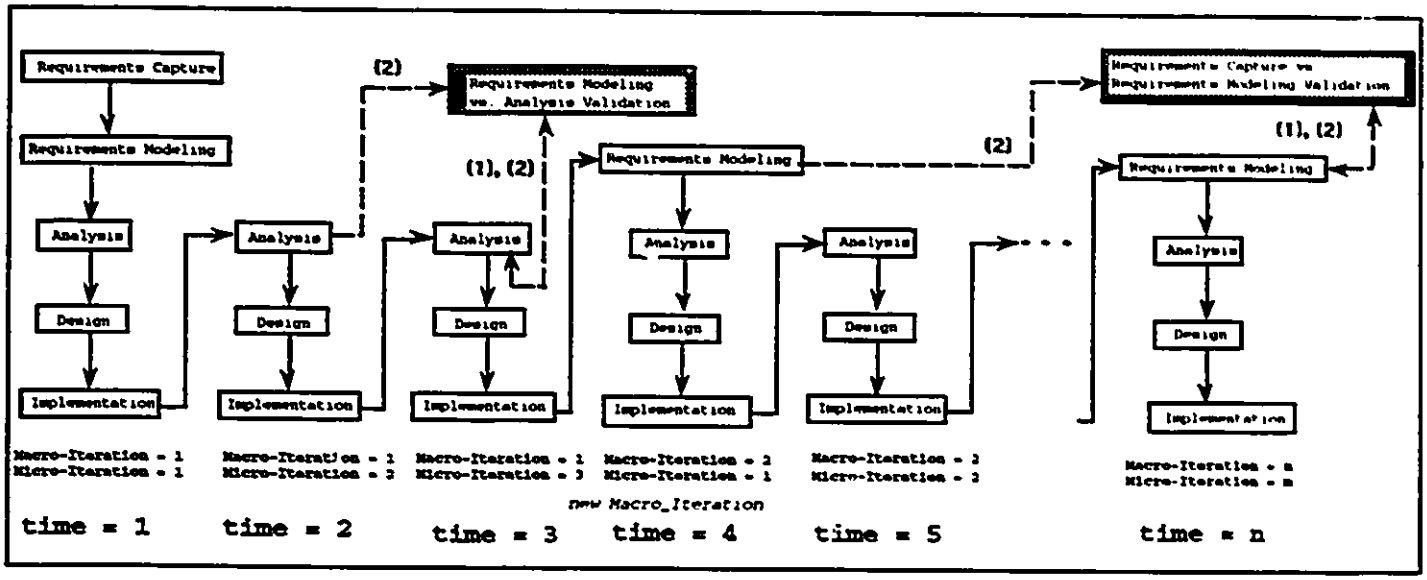


Figure 7.2 Example of consistency in the context of Iterations

Observation 2

In the OO paradigm, certain transformations or mappings, from one phase to the next, drive other transformations. For example, the Static View at the Analysis phase drives the transformations of the Communication, Dynamic and Processing Views from the Analysis phase to the Design phase.

The last point is very important and many times ignored. When moving from one phase to the next, the first notations that are refined are those that describe the Static View (e.g. Class Diagrams [14, 22]). Then only, other Views are refined (e.g. State Transition Graphs[18] for the Dynamic View).

How can we test that certain transformations indeed drive other transformations?

As said earlier, the purpose of Semantic Verification is to ensure that

- Views are correct in themselves.
- Views are complete with respect to other Views at the same phase of development

The purpose of Validation is to ensure that

- Views are consistent with respect to other Views at different phases of development

The combination of Semantic Verification and Semantic Validation ensures that the Static View at Analysis drives the Communication, Dynamic and Processing Views from the Analysis phase to the Design phase. This is illustrated with an example in Fig. 7.3. Note that in Fig. 7.3, we only show the transformation driving of the Dynamic View from Analysis to Design.

* when we achieve `(1)`, `(3)` and `(7)`, we have ensured that the Static View at Design is a valid transformation of a corrected and complete Static View at Analysis;

* when we achieve `(2)`, `(5)`, `(4)` and `(6)`, we have ensured that Static View at Analysis drove the transformation of the Dynamic View from Analysis to Design, therefore Dynamic Semantic Validation of the Dynamic View has been achieved (i.e. line labeled as `(8)` in Fig. 7.3).

Therefore, Dynamic, Communication and Processing Semantic Validation are transitively obtained and do not require to be separate testing activities We provide a list of heuristics that can be applied to double check that validation has indeed occurred. Semantic Validation of the Static View is however very important. We concentrate our effort in the latter.

Note that we do not talk of Syntactic Validation, because Semantic Validation aims at validating the semantic transformations between notations that have been syntactically verified. For example, in the process described in Fig. 7.3, Semantic Validation (depicted by the directed line labeled as `(5)` occurs between two notations that have undergone Syntactic Verification (depicted by the directed lines labeled as `(1)` and `(3)`).

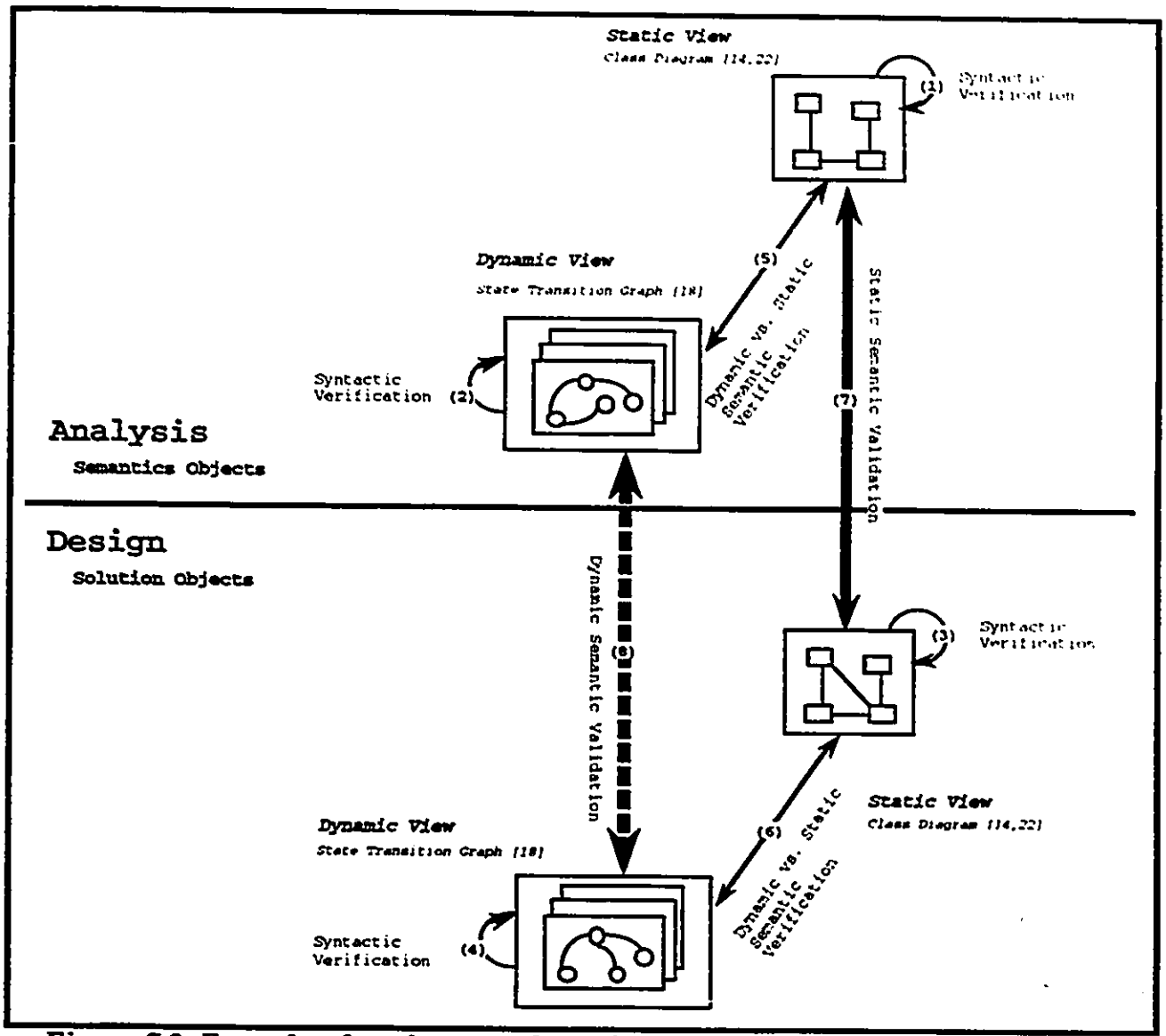


Figure 7.3 Example of testing transformations driving other transformations in Validation

7.2.1 Requirements Capture vs. Requirements Modeling Validation

As discussed in section 6.3.1, requirements are usually capture using a combination of text to capture different aspects of the application to be developed (e.g. performance, data, architecture, behavior, etc.) and using notations that fall under the Scenario model.

Notation that fall under the Scenario Model, like Use Cases, are helpful in defining functional requirements and deriving objects as shown in [18, 74].

We will use the term Use Case to illustrate our discussion; note that we could have used the term Scenario since a Scenario corresponds to an instance of a Use Case.

"Use Cases and objects are related in that a Use Case is a tool to identify objects, class inheritance and object aggregates" [74]. This is usually capture in an Object Model [18,22] consisting of Domain Objects and their relationships of the problem domain. The latter provides the Static View.

Use Cases also provide aspects of the Communication View as described in Chapter 4. Every Use Case of the system can be viewed as a communication between the User and the System. More specifically, a Use Case shows how several object may participate given their responsibilities and how each object may participate in several Use Cases.

This mapping from Use Cases to the Static and Communication Views is what is tested in Requirements Validation. More specifically:

in the Static View

(1) every Entity (i.e. Domain Class/Object) must be derived from at least one Use Case and

(2) every Use Case must have at least one Entity allocated to it.

in the Communication View

(1) every interaction between Entities offers at most one Use Case or part of a Use Case and

(2) the course of events of every Use Case should be realized by at least one interaction between Entities

A useful tool for describing the relationships between Use Cases and Entities is the 'Use Case - Object Matrix' which shows the dualism between the latter. This matrix described in [74], shows how several Entities (objects), participate in each Use Case and how each Entity (object) may participate in several Use Cases.

Example 4, illustrates an abstract example of Requirements Validation.

Example 4: Requirements Validation

Use Cases or Scenarios specify a complete course of events of the application to be developed from a User's perspective. From these, a structural configuration (i.e. Domain Classes/Objects) of the problem domain are identified. These should be traced to the Use Cases or Scenarios. Thus, Requirements Validation aims at ensuring two things in the Static View:

(1) every Domain Class/Object is derived from at least one Use Case or Scenario

if the latter is not true as shown by C9 in the figure below then

a. the Domain Class/Object (C9) should be removed or

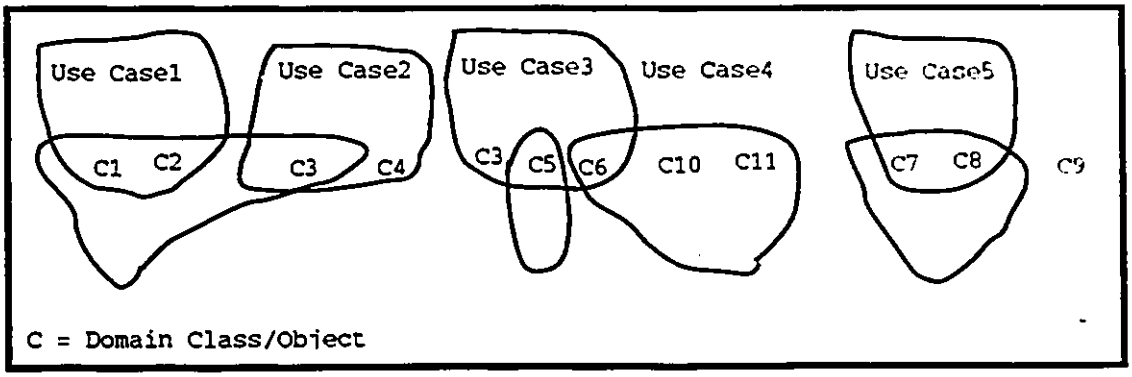
b. a new Use Case or Scenario should be identified to include C9.

(2) every Use Case or Scenario must have an Entity allocated to it

if the latter is not true as shown by `Use Case4` in the figure below then

a. `Use Case4` should be removed or

b. new Domain Class(es)/Object(s) must be identified to support `Use Case4`.



The interactions between the Domain Class/Objects identified by Use Cases can be described by notations in the Object Interaction Model such as Interaction Diagrams [18]. Note that the creation of Interaction Diagrams is driven by the creation of Domain Classes/Objects. These interactions should be traced to the Use Cases or Scenarios (we will use Interaction Diagrams as an example notation in the discussion that follows). Semantic Validation in the Communication aims at ensuring two things:

(1) every Interaction Diagram offers at most one Use Case or part of a Use Case

if the latter is not true as shown by `ID10` in the figure below, then

a. the Interaction Diagram (`ID10`) should be removed **or**

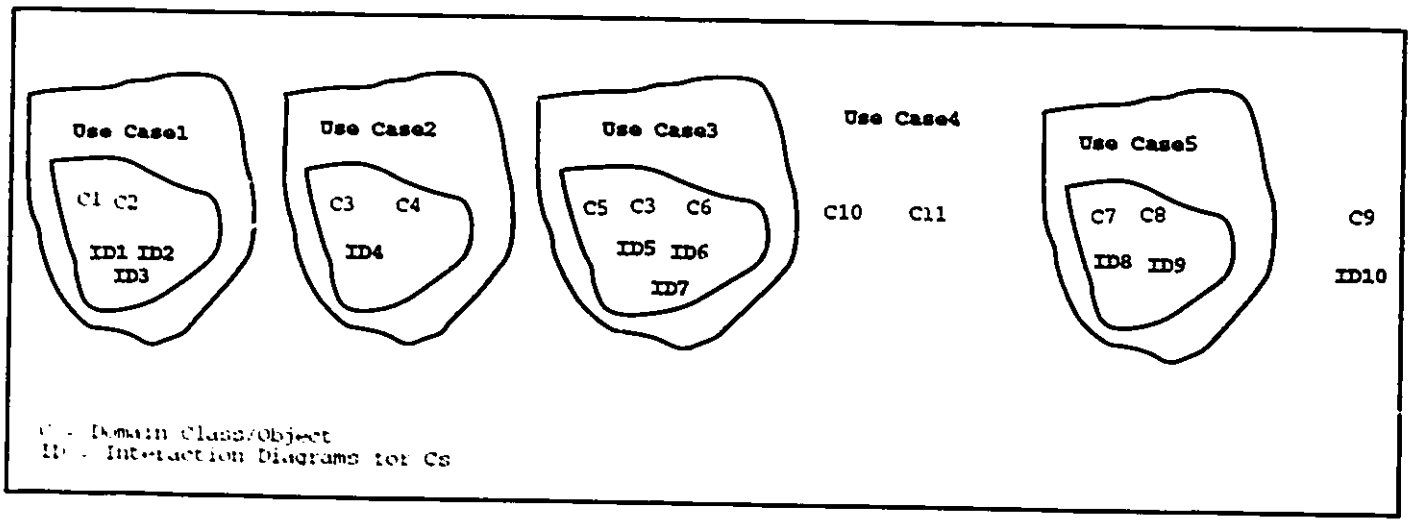
b. a new Use Case or Scenario should be identified to include `ID10`.

(2) the course of events of every Use Case should be realized by at least one Interaction Diagram

if the latter is not true as shown by `Use Case4` in the figure below, then

a. `Use Case4` should be removed **or**

b. a new Interaction Diagrams must be identified to support `Use Case4`.



7.2.2 Development Validation

As described in Chapter 4, models used at different phases of the Generic Object-Oriented Software Development Methodology provide different semantic (views) of the application to be developed. The four basic views are: Static, Communication, Dynamic and Processing.

These Views are provided at different levels of abstraction (i.e. System, Class/Object and Class/Object Internal) as described in Fig. 5.1.

Semantic Validation aims at verifying that the semantics provided by a View is consistent throughout **different** phases of the Generic Object-Oriented Software Development Methodology. The semantics of Views is described in terms of generic modeling constructs as described in Chapter 4. Semantic Validation will be defined in terms of these generic modeling constructs. These were defined in Chapter 4.

Static View: Entities, Relationships, Properties, Operations
Communication View: Entities, Messages, Ordering of messages
Dynamic View: States, Events, Actions, Guards
Processing View: Process, Data, Control flow and Data Flow

7.2.2.1 Static Semantic Validation

The Static View is provided by notations that were classified into the following models, as described in Chapter 4:

- Domain/Subsystem Configuration Model
- Class Configuration Model
- Object Configuration Model

These models provide the Static View of the application to be developed at different levels of abstraction. Semantic Validation aims at ensuring that the Static View is consistent through these different levels of abstraction. More specifically, it aims at ensuring that Use Cases or Scenarios in the Requirements Capture phase can be traced to the Coded Classes/Objects of the Implementation phase, as shown in Fig. 7.4, and that the relationships between entities (i.e. Domains, Subsystems, Semantic Classes/Objects, etc.) are also traced through the different levels of abstraction.

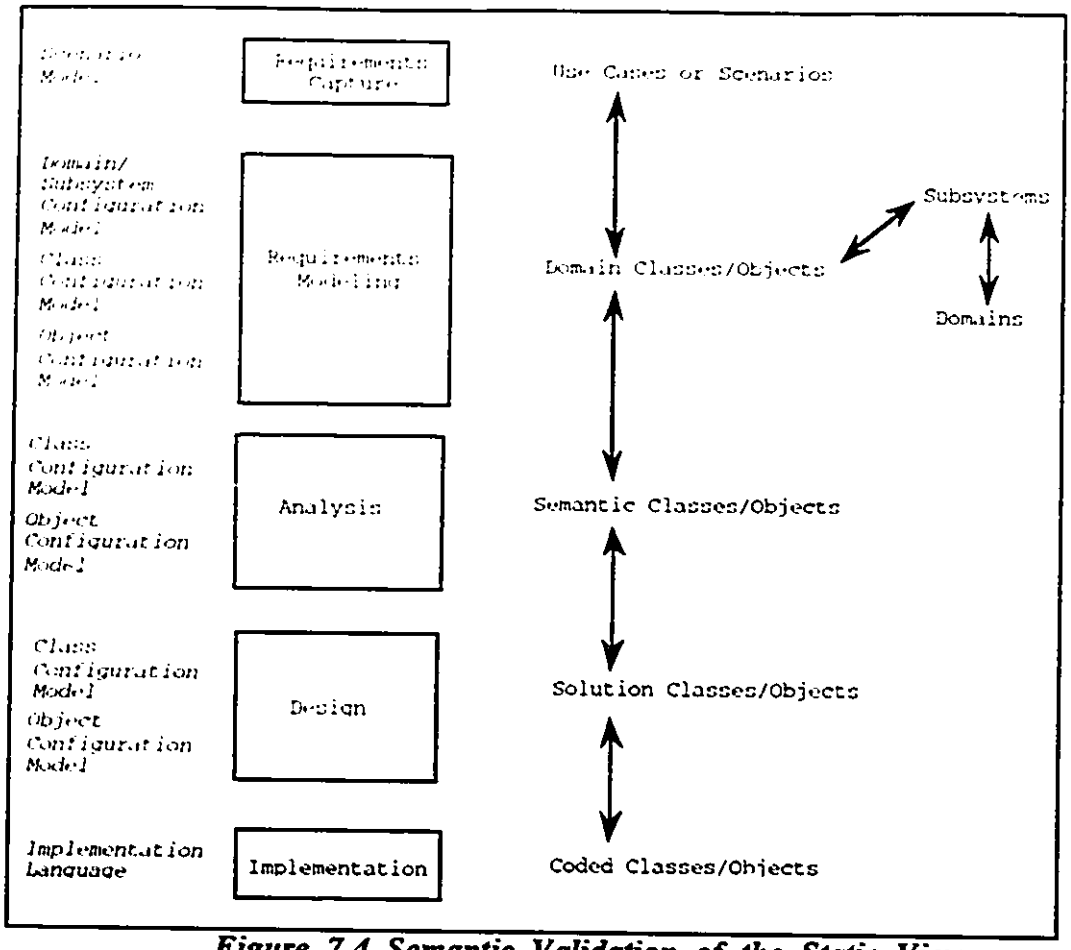


Figure 7.4 Semantic Validation of the Static View

Example 5, illustrates an abstract example of Static Semantic Validation of Subsystems vs. Domains. We now analyze the rest of the transformations illustrated in Fig. 7.4.

Semantic Classes/Objects are derived from Domain Classes/Objects; Solution Classes/Objects are derived from Semantic Classes/Objects and Coded Classes/Objects are derived from Solution Classes/Objects, as shown in Fig. 7.4. Transformations become more complex in nature due to inheritance. In Example 5, we only talked of decomposition, now we must talk of generalization and specialization. However, the basic principles of Semantic Validation remains the same; i.e. every Semantic Class/Object, should be rationalized (changed, deleted or added) from its Domain Class(es)/Objects and vice-versa; every Solution Class/Object should be rationalized from its Semantic Class(es)/Object(s) and vice-versa; etc. This is shown in Fig. 7.5.

From Requirements Modeling down to Implementation, the models on the left of Fig. 7.4 describe entities (i.e. Domains, Subsystems, Domain Classes/Objects, Semantic Classes/Objects, etc.) and their relationships (i.e. Bridges, Subsystem relationships, etc.). The transformations from one phase to the next can be generalized as shown in Fig. 7.5.

Fig. 7.5 captures the set of transformation heuristics of the Static View from one level of abstraction to the next. Tools that can automate such a set and rationalize the transformations are greatly in demand. Lets take an example, most tools (e.g. Argos CASE tool), ensure that if a class is deleted, the associations to and from that class are also deleted. The missing piece is the rationalization of such a change, i.e. ensuring that either a new class will take care of the functionality of the deleted class and the associations of the deleted class are handled by the new class, or the deleted associations are no longer required or are handled by other classes.

The rationalization of transformations is difficult to automate. This is because, of the difficulty in keeping track of the transformation, how invalid transformations should be handled, and the need to identify a set of transformations that will not restrict the development of a notations through different levels of abstractions. The case tool Argos provides a crude type of Semantic Validation but restricts important refinements. For example, it allows to refine a by-directional association at Analysis into one of its unidirectional relations at Design, but disallows the creation of a new class at Design that is not present at Analysis.

A tool that would automate Semantic Validation would allow the software designer to navigate through a class diagram and apply the set of transformation that a class diagram can undergo to generate a new class Diagram at a lower level of abstraction. For example in the case of inheritance, he/she would click on the classes that will be generalized and would be prompted to enter the name of the new class; the system, thus keeps tracks of the transformation.

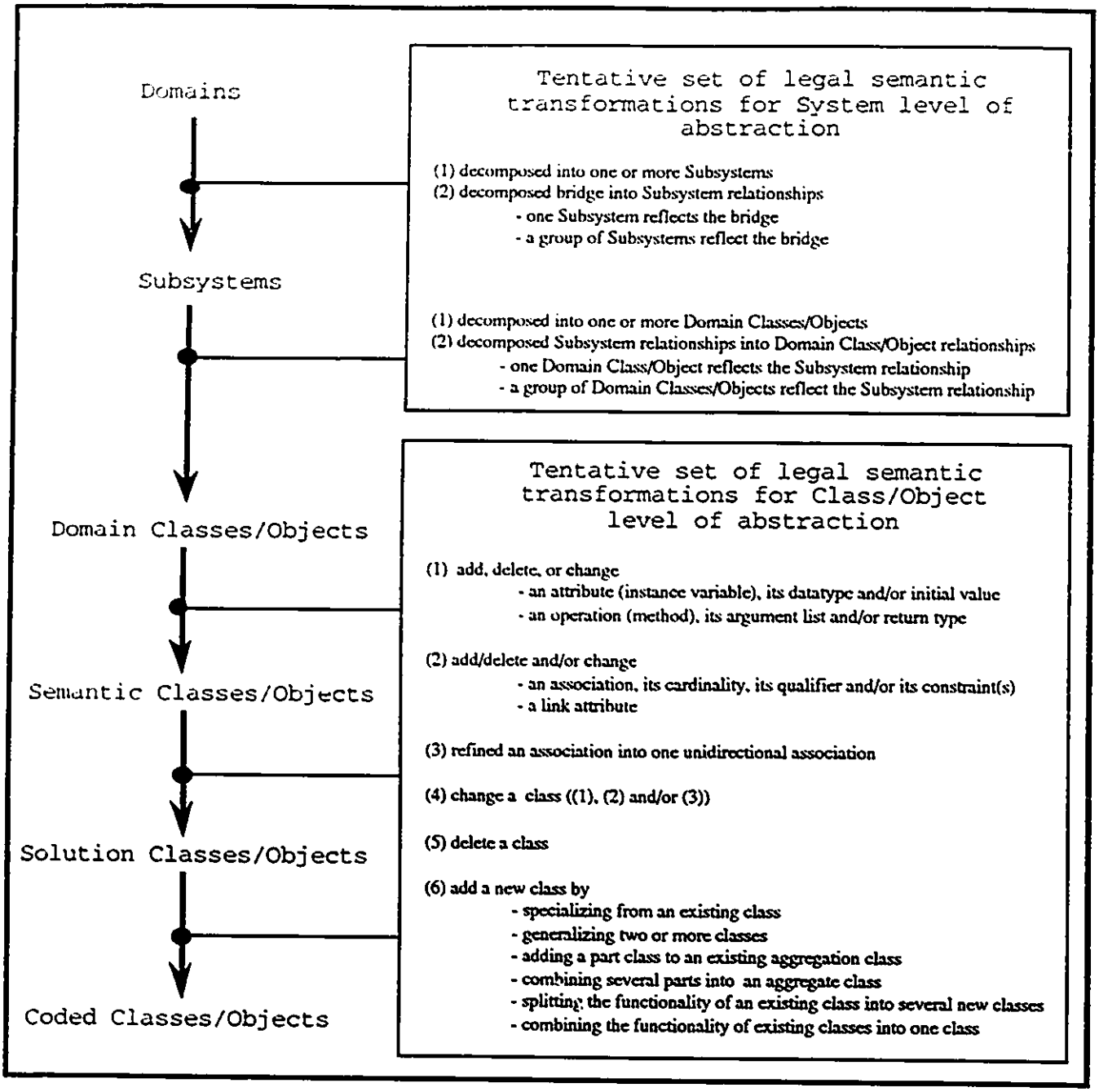


Figure 7.5 Generalized static semantic transformations in Validation

Example 5: Static Semantic Validation of Subsystems vs. Domains

Subsystems are classified into Domains. Semantic Validation aims at ensuring that:

(1) every identified Subsystem is part of a decomposition of a Domain; if this is not true as shown by S4 in the figure below then

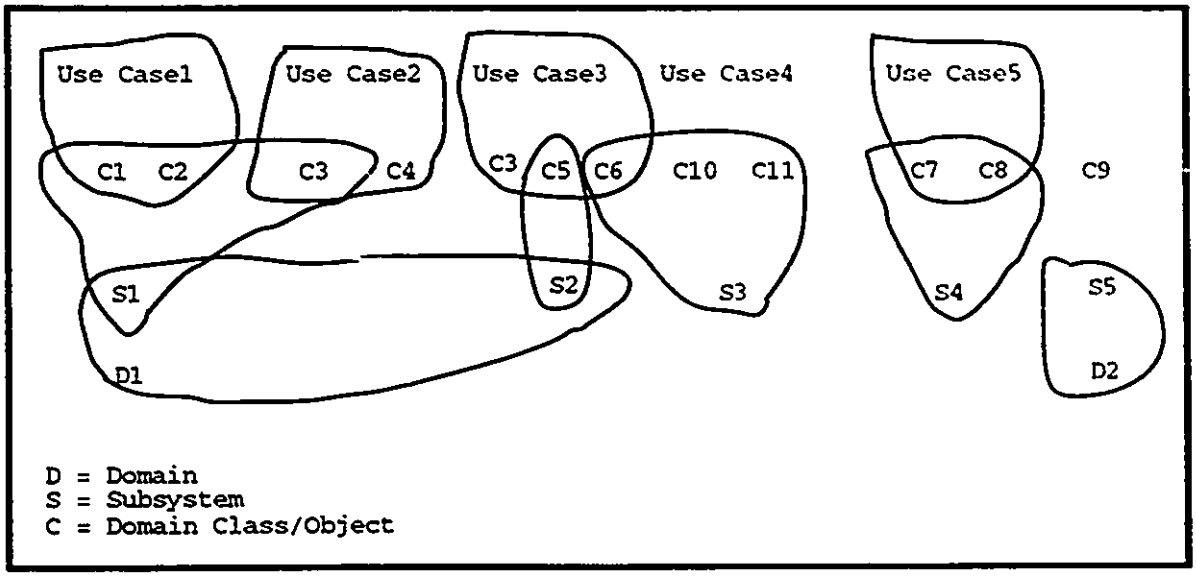
   a. the Subsystem should be removed and Domain Class(es)/Object(s) (e.g. C7, C8) as well as the Use Cases or Scenarios (e.g. Use Case5) should be updated **or**

   b. a new Domain should be created to include S4.

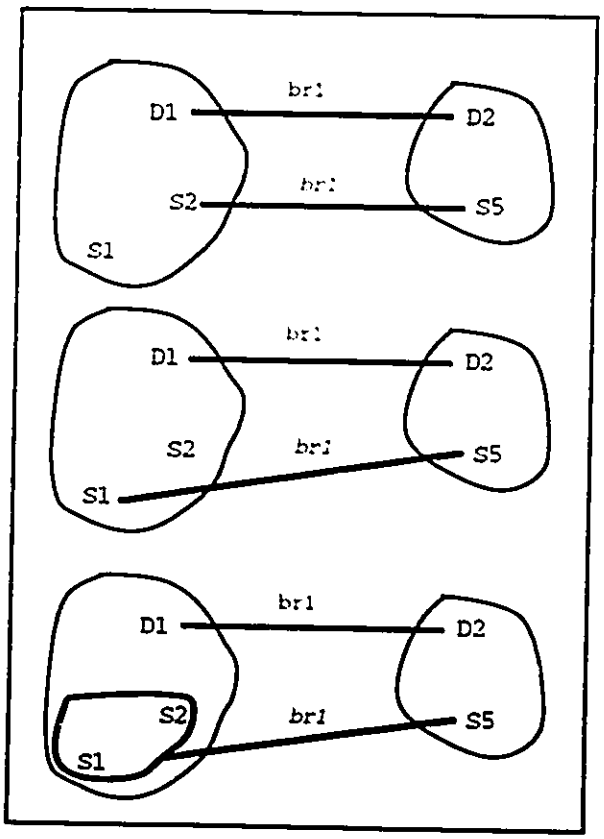
(2) every Domain is decomposed into identified Subsystems (by Domain Classes/Objects and in turn by Use Cases or Scenarios); if this is not true as shown by D2 in the figure below, then

   a. the Domain should be removed and Subsystems (e.g. S5) should be updated) **or**

   b. the new Subsystem should be updated (i.e. new Domain Class(es)/Object(s) and Use Case(s) or Scenario(s) should be identified to include S5).



(3) every Domain relationship is reflected in the identified Subsystems. In the figure below, `br1` is a bridge (relationship) between Domains `D1` and `D2`. Semantic Validation aims at ensuring that one of the configuration below must is true.



7.2.2.2 Communication Semantic Validation

The Communication View is provided by notations that were classified into the following models, as described in Chapter 4:

- Scenario Model
- Domain/Subsystem Interaction Model
- Object Interaction Model

Their generic modeling constructs were:

- Entities,
- Messages
- Ordering of Messages

These models provide the Communication View of the application to be developed at different levels of abstraction. Semantic Validation aims at ensuring that the Communication View is consistent through these different levels of abstraction. More specifically, it aims at ensuring that Use Cases or Scenarios in the Requirements Capture phase can be traced to the interactions between Coded Classes/Objects of the Implementation phase, as shown in Fig. 7.6.

Example 6 illustrates an abstract example of Communication Semantic Validation of interactions of Domain Classes/Objects vs. interactions of Subsystems.

Example 7 illustrates an abstract example of Communication Semantic Validation of interactions of Subsystems vs. interactions of Domain. We now analyze the rest of the transformations illustrated in Fig. 7.6.

Semantic Classes/Objects are derived from Domain Classes/Objects; Solution Classes/Objects are derived from Semantic Classes/Objects and Coded Classes/Objects are derived from Solution Classes/Objects, as shown in Fig. 7.6. At each phase, the interactions between these entities is described by the notation that fall under the Object Interaction Model, as described in Chapter 4.

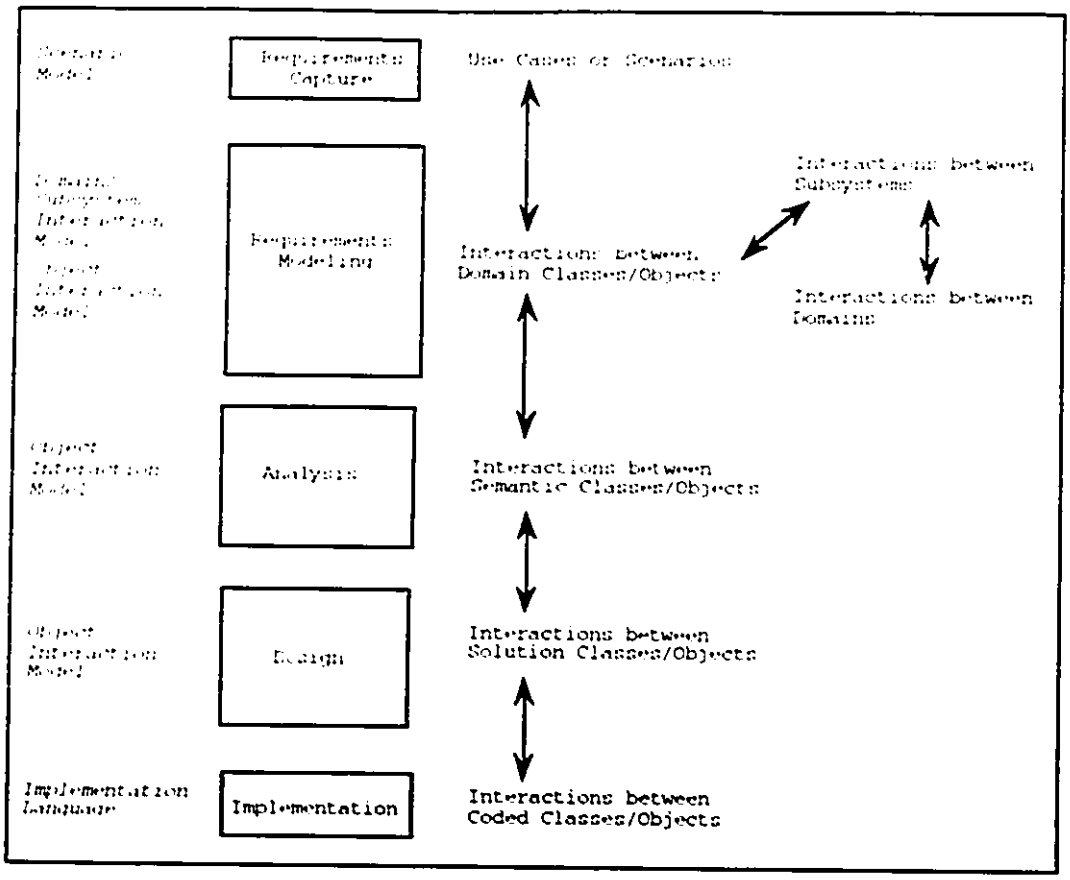
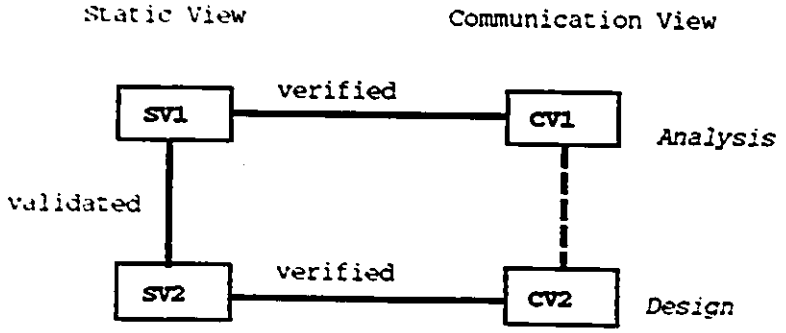


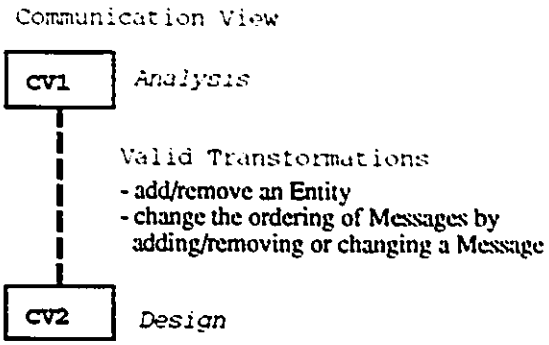
Figure 7.6 Semantic Validation of the Communication View

Semantic Validation aims at ensuring that the interactions between entities at one phase are reflected in the next phase and vice-versa. As said earlier, in observation 2 in section 7.2, from one phase to the next, the transformations in interactions (i.e. Communication View) are driven by the transformations in the Static View. In the figure below, if Semantic and Syntactic Verification has occurred between SV1 and CV1 and between SV2 and CV2; and Semantic Validation has occurred between SV1 and SV2; then we can ensure that Semantic Validation has occurred between CV1 and CV2.



Lets take a brief example to illustrate our discussion: assume that CV1 describes an Interaction Diagram between an instance of class A and an instance of class B. In order to achieve Semantic Validation of CV1 vs. CV2, we must have an understanding of the transformation of classes A and B from SV1 to SV2. If class A's functionality was split into classes A1 and A2, then either an instance of class A1 or an instance of class A2 must now interact with an instance of class B to achieve semantic consistency from CV1 to CV2.

Fig. 7.5, provided a tentative set of static semantic transformations for the Class/Object level of abstraction. We now provide a set of heuristics based on these transformations to double check that Communication Semantic Validation has indeed occurred.



The figure above illustrates the valid transformations of the Communication View. These transformations depend on the static transformations as follows:

(1) a class is deleted in the transformation from SV1 to SV2

The Interaction Diagrams in CV2 should not depict an instance of the deleted Class.

(2) a class is added in the transformation from SV1 to SV2

Fig. 7.5, lists the circumstances under which a class is added in a static transformation.

a) If a class is added in SV2 by specializing from an existing class in SV1 then the Interaction Diagrams for this new class in CV2, should include the Interaction Diagrams from CV1 that involve the superclass (i.e. this set, is a clone of the

Interaction Diagrams in CV1 that involve instances of the superclass of the new class but in place of the instance of the superclass, an instance of the new class should be drawn). The basic rational is that Interaction Diagrams that involve an instance of a superclass can be inherited by an instance of the subclass of this superclass. Note that this is true if the new class and its superclass are concrete classes.

b) If a class in SV2 is added by generalizing two or more classes in SV1 then the Interaction Diagrams in CV2, should include the Interaction Diagrams from CV1 that involve the subclasses (i.e. this set is a clone of the Interaction Diagrams in CV1 that involve instances of the generalized classes but in place of these instances, an instance of the new superclass should be drawn). Note that this is true if the new superclass and its subclasses are concrete classes.

c) If a class is added in SV2 by creating a new part to an existing aggregation class in SV1 then nothing can be said about the communication transformation of CV1 to CV2, since the new class may have an entirely new set of interactions.

d) If a class is added in SV2 by combining several parts into a new aggregation class in SV1 then this is similar to case b), the Interaction Diagrams in CV2, should include the Interaction Diagrams from CV1 that involve the part classes (i.e. this set is a clone of the Interaction Diagrams in CV1 that involve instances of the part classes but in place of these instances, an instance of the new aggregation class should be drawn).

e) If a class is added in SV2 by splitting the functionality of an existing class into several new classes in SV1 then this is similar to case d), the Interaction Diagrams in CV2, should include the Interaction Diagrams from CV1 that involve the previous uncombined classes (i.e. this set is a clone of the Interaction Diagrams in CV1 that involve instances of the uncombined classes but in place of these instances, an instance of the new functional class should be drawn).

f) If a class is added in SV2 by combining the functionality of existing classes into a new class in SV1 then it is very difficult to determine the communication transformation from CV1 to CV2. All it is possible to assert is that the Interaction Diagrams in CV1 that involve an instance of the unsplitted class must be present in CV2 with instances of the new splitted classes.

(3) a class is changed in the transformation from SV1 to SV2

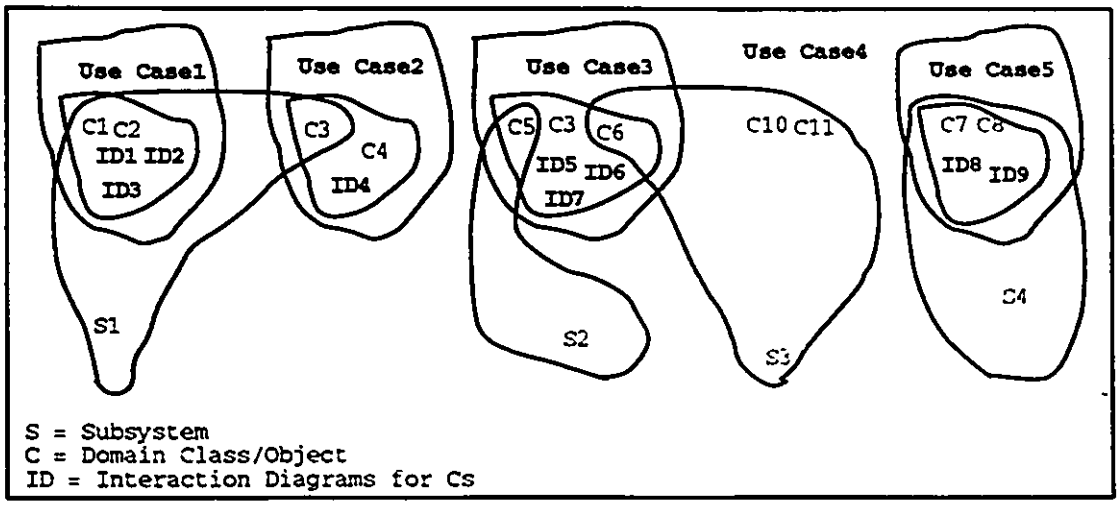
Fig. 7.5. lists the circumstances under which a class is changed in a static transformation.

a) If an association and/or an non-internal operation is added to a class from SV2 to SV1 then the Interaction Diagrams in CV2 should depict the interactions of the changed class by including invocations of the new operation and/or interactions that implement the new association.

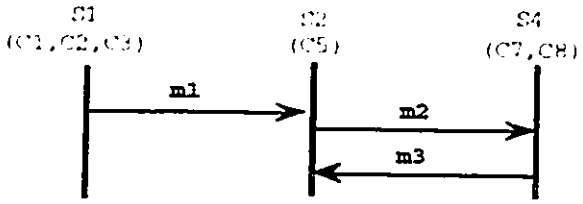
b) If an association and/or an non-internal operation is deleted in a class from SV2 to SV1 then the Interaction Diagrams in CV2 should not depict the interactions of the changed class by excluding invocations of the deleted operation and/or interactions that implement the deleted association

Example 6: Communication Semantic Validation of interactions of Domain Classes/Objects vs. interactions of Subsystems

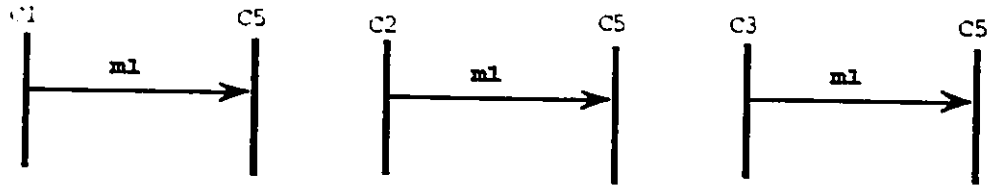
Domain Classes/Objects were classified into Subsystems. The interactions between these Subsystems can be described by notations in the Domain/Subsystem Interaction Model such as the 'augmented Subsystem Communication Model' notation described in example 1 of section 6.3.2.1. Note that the creation of 'augmented Subsystem Communication Models' was driven by the creation of Subsystems. We will show how the Semantic Validation at this stage augments the set of Interaction Diagrams defined so far. Consider the figure below.



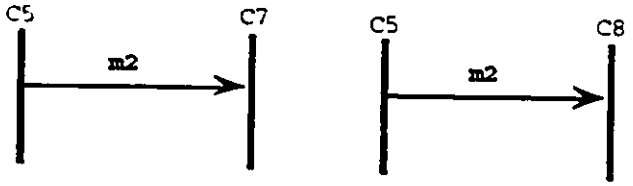
Assume that an 'augmented Subsystem Communication Models' models looks like this (note that we have included the Domain Classes/Objects that each Subsystem contains, for clarity)



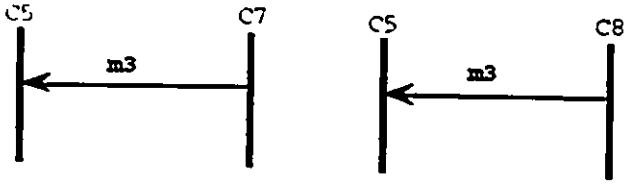
For **m1** to take place, C1 and/or C2 and/or C3 must interact with C5. Currently the set of Interaction Diagrams (ID1 through ID10) do not reflect this since they are on a per Use Case or Scenario base (i.e. the Interaction Diagrams are encapsulated to the specific Use Case or Scenario). Semantic Validation, will ensure that the set of Interaction Diagrams for Domain Classes/Objects are extended to include either of the following Interaction Diagrams:



For **m2** to take place, C5 must interact with C7 and/or C8. Semantic Validation, will ensure that the set of Interaction Diagrams for Domain Classes/Objects are extended to include either of the following Interaction Diagrams:



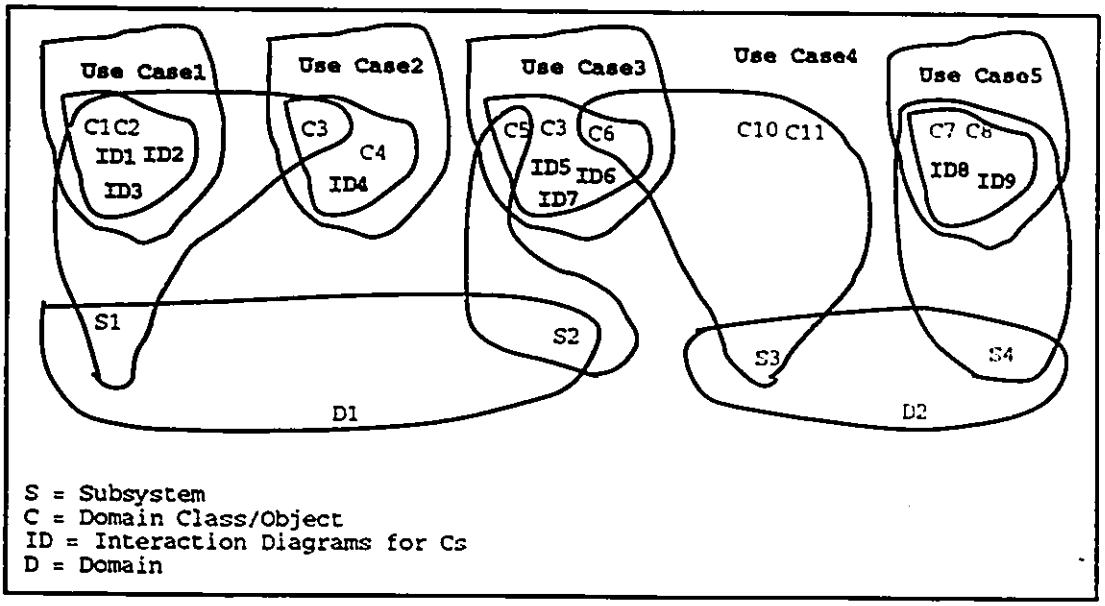
Finally for **m3** to take place, C7 and/or C8 must interact with C5. Semantic Validation, will ensure that the set of Interaction Diagrams for Domain Classes/Objects are extended to include either of the following Interaction Diagrams:



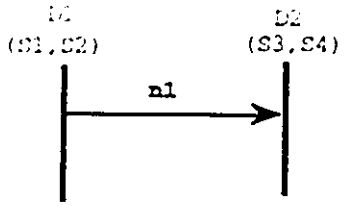
In more general terms, Semantic Consistency, should ensure that every 'augmented Subsystem Communication Model' describing Subsystem interactions is reflected in at least one Interaction Diagram describing their contained Domain Classes/Objects interactions (this was described above). If such an Interaction Diagram is not supported, then the 'augmented Subsystem Communication Model', should be removed. Also Semantic Validation ensures that every Subsystem participates in at least one 'augmented Subsystem Communication Model', since every Subsystem must eventually interact with other Subsystems in the whole application to be developed.

Example 7: Communication Semantic Validation of interactions of Subsystems vs. interactions of Domain

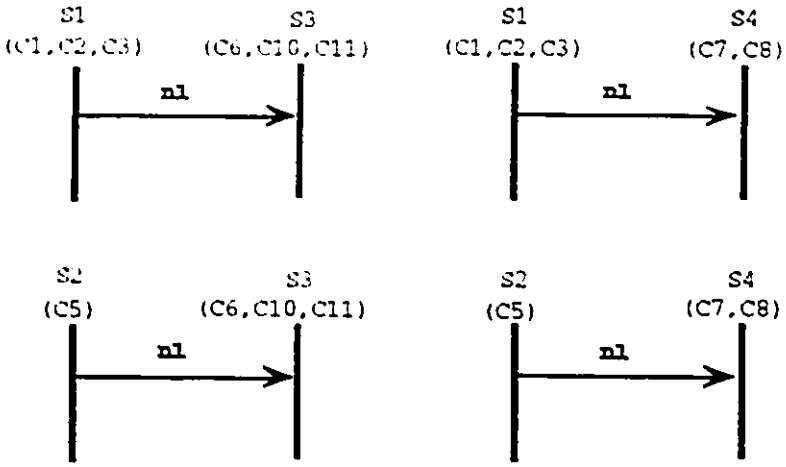
Subsystems are classified into Domains. The interactions between these Domains can be described by notations in the Domain/Subsystem Interaction Model such as the 'augmented Domain Communication Model' (this is similar to the 'augmented Subsystem Communication Model' notation, described in section 6.3.2.1, but with Domains instead of Subsystems). Note that the creation of 'augmented Domain Communication Models' was driven by the creation of Domains. Consider the figure below.



Assume that an 'augmented Domain Communication Models' models looks like this (note that we have included the Subsystems that each Domain contains, for clarity)



For `n1` to take place, S1 and/or S2 must interact with S3 and/or S4. Semantic Validation, will ensure that the set of 'augmented Subsystems Communication Models' for Subsystems includes either of the following 'augmented Subsystems Models':



(note that, in turn, the above must be supported by the Interaction Diagrams, as described in the previous section)

In more general terms, Semantic Consistency, should ensure that

(1) every 'augmented Domain Communication Model' describing Domain interactions is reflected in at least one 'augmented Subsystem Communication Model' as described above. If such an 'augmented Subsystem Communication Model' is not supported, then
   a. the 'augmented Domain Communication Model' should be removed **or**
   b. a new 'augmented Subsystem Communication Model' must be created to support such a Domain interaction (in this case, the Interaction Diagrams describing Domain Classes/Objects interactions should be updated, to support the new 'augmented Subsystem Communication Model').

(2) every 'augmented Subsystem Communication Model' describing interaction between Subsystems in different Domains should be reflected in at least one 'augmented Domain Communication Model'. If such an 'augmented Domain Communication Model' is not supported, then

a. the 'augmented Subsystem Communication Model' should be removed or

b. a new 'augmented Domain Communication Model' must be created to support such a Subsystem interaction.

7.2.2.3 Dynamic Semantic Validation

The Dynamic View is provided by notations that were classified into the following model, as described in Chapter 4:

- Object Internal Behavior Model

Their generic modeling constructs were:

- States,
- Events
- Guards
- Actions

The Object Internal Behavior Model provides the Dynamic View of the application to be developed at different levels of abstraction. Semantic Validation aims at ensuring that the Dynamic View is consistent through these different levels of abstraction. More specifically, it aims at ensuring that the internal behavior of a Semantic Class/Object at the Analysis phase can be traced to its Coded Class/Object of the Implementation phase, as shown in Fig. 7.7.

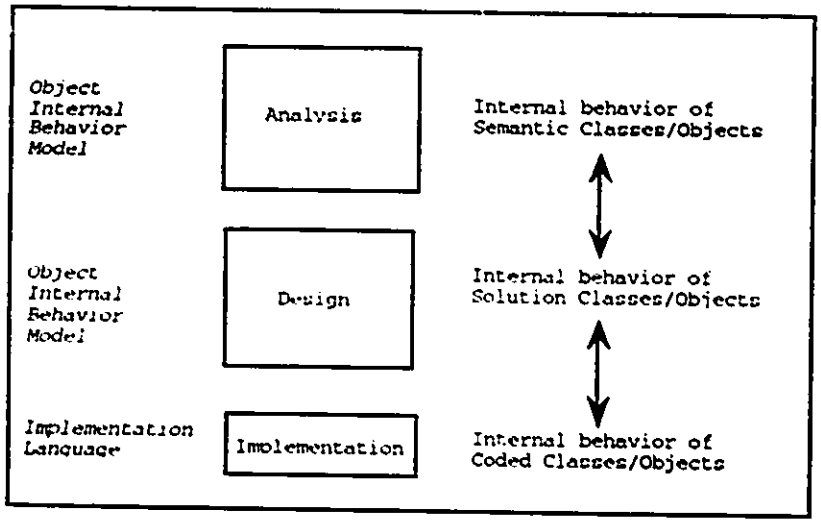
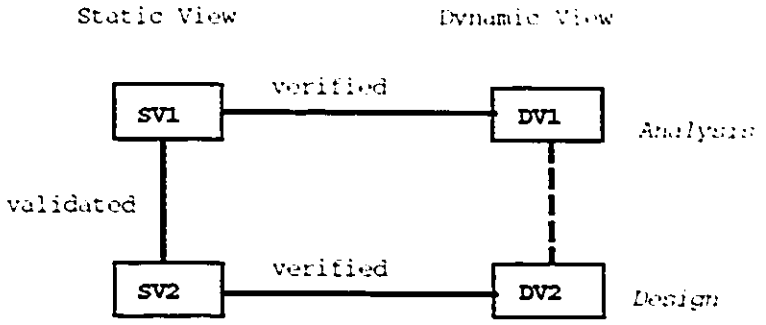


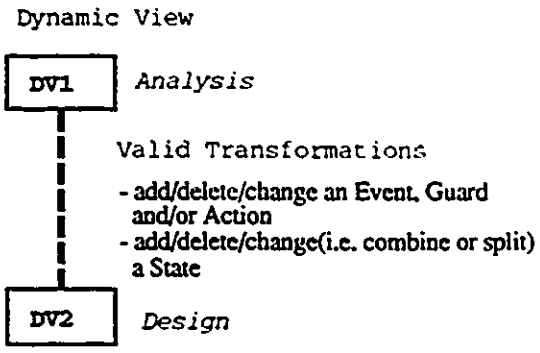
Figure 7.7 Semantic Validation of the Dynamic View

Semantic Validation aims at ensuring that the internal behavior of an entity at one phase is reflected in the next phase and vice-versa. As said earlier, in observation 2 in section 7.2, from one phase to the next, the internal behavior of an entity (i.e. Dynamic View) is driven by the transformations in the Static View. In the figure below, if Semantic and Syntactic Verification has occurred between SV1 and DV1 and between SV2 and DV2; and Semantic Validation has occurred between SV1 and SV2; then we can ensure that Semantic Validation has occurred between DV1 and DV2.



Lets take a brief example to illustrate our discussion; assume that DV1 describes the internal behavior of class A. In order to achieve Semantic Validation of DV1 vs. DV2, we must have an understanding of the transformations of class A from SV1 to SV2. If a Property was added to the class from SV1 to SV2, then the internal behavior of the class in DV2 must reflect this as either a new or changed State, a new or changed Action or a new or changed Guard.

Fig. 7.5, provided a tentative set of static semantic transformations for the Class/Object level of abstraction. We now provide a set of heuristics based on these transformations to double check that Dynamic Semantic Validation has indeed occurred.



The figure above illustrates the valid transformations of the Dynamic View. These transformations depend on the static transformations as follows:

(1) a class is added or deleted in the transformation from SV1 to SV2

The internal behavior for that class should be created or deleted respectively in DV2..

(2) a class is changed in the transformation from SV1 to SV2

Fig. 7.5, lists the circumstances under which a class is added in a static transformation.

a) If an Operation is added, changed or deleted in a class from SV1 to SV2 then an Event or an Action in DV2 should be added, changed or deleted respectively.

b) If a Property is added, changed or deleted in a class from SV1 to SV2 then a State, an Action or a Guard in DV2 should be added, changed or deleted respectively.

c) If a Relation is added, changed or deleted in a class from SV1 to SV2 then an Event in DV2 should be added, changed or deleted respectively.

7.2.2.4 Processing Semantic Validation

The Processing View is provided by notations that were classified into the following model, as described in Chapter 4:

- Data Manipulation Model

Their generic modeling constructs were:

- Processes,
- Data Flows
- Data

The Data Manipulation Model provides the Processing View of the application to be developed at different levels of abstraction. Semantic Validation aims at ensuring that the Processing View is consistent through these different levels of abstraction. More specifically, it aims at ensuring that the internal manipulations of data in a Semantic Class/Object at the Analysis phase can be traced to its Coded Class/Object of the Implementation phase, as shown in Fig. 7.8.

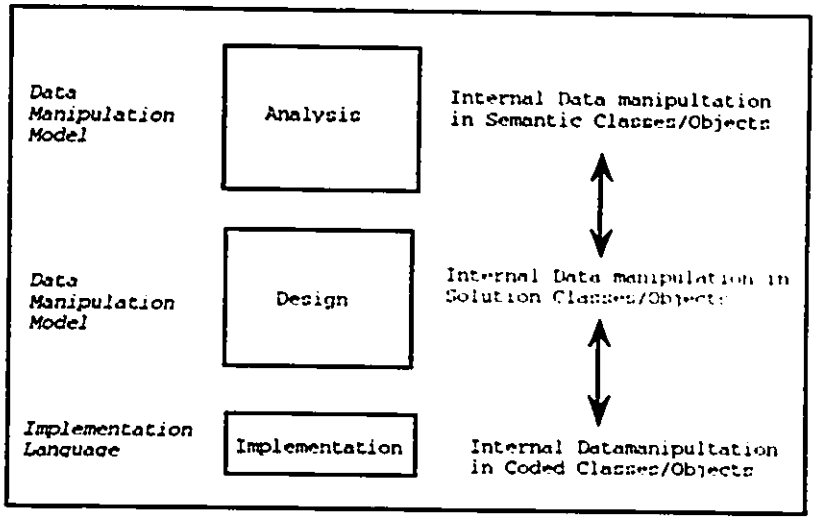
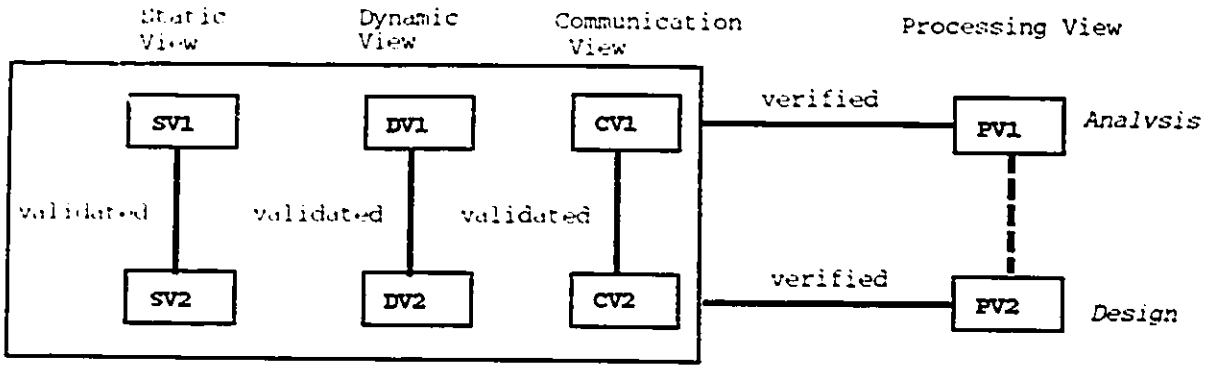


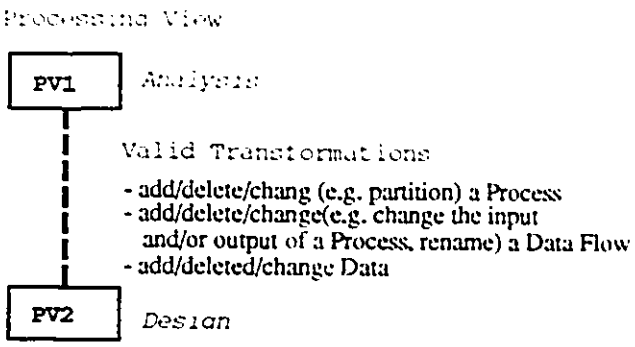
Figure 7.8 Semantic Validation of the Processing View

Semantic Validation aims at ensuring that the internal manipulations of data in an entity at one phase is reflected in the next phase and vice-versa. As said earlier, in observation 2 in section 7.2, from one phase to the next, the internal manipulations of data in an entity (i.e. Processing View) is driven by the transformations in the Static View. In the figure below, if Semantic and Syntactic Verification has occurred between SV1, DV1, CV1 and PV1 and between SV2, DV2, CV2 and PV2; and Semantic Validation has occurred between SV1 and SV2, DV1 and DV2, CV1 and CV2; then we can ensure that Semantic Validation has occurred between PV1 and PV2.



Lets take a brief example to illustrate our discussion; assume that PV1 describes the internal data manipulation of class A. In order to achieve Semantic Validation of PV1 vs. PV2, we must have an understanding of the transformations of class A from SV1 to SV2. If a complex Operation involving a Property was added to the class from SV1 to SV2, then the internal data manipulation of the class in PV2 must reflect this as a new Process(es) or part of a Process.

Fig. 7.5, provided a tentative set of static semantic transformations for the Class/Object level of abstraction. We now provide a set of heuristics based on these transformations to double check that Processing Semantic Validation has indeed occurred.



The figure above illustrates the valid transformations of the Processing View. These transformations depend on the static transformations as follows:

(1) a class is added or deleted in the transformation from SV1 to SV2

The internal data manipulation model for that class should be created (i.e. in the case that the class contains complex operations) or deleted respectively in DV2..

(2) a class is changed in the transformation from SV1 to SV2

Fig. 7.5, lists the circumstances under which a class is added in a static transformation.

a) If a complex Operation is added, changed or deleted in a class from SV1 to SV2 then a Process(es) or part of a Process in PV2 should be added, changed or deleted respectively with its corresponding inputs and outputs.

b) If a Property is added, changed or deleted in a class from SV1 to SV2 then if this Property is being manipulated by a complex Operation then a Data Flow in DV2 should be added, changed or deleted respectively with its corresponding inputs, Process and outputs.

7.3 Incorporation testing activity types

Incorporation is the process of determining whether Notations used at different iterations are incorporated in one another. More specifically, Incorporation refers to the set of activities that allow to determine if Notations at phase n of iteration i are integrated in notations at phase n of iterations $i+1$ and whether Notations at phase n of iteration $i+1$ are integrated in notations at phase n of iterations i. Incorporation provides Forward Iteration traceability and Backward Iteration traceability.

As discussed in section 7.2, in the OO paradigm, certain transformations or mappings, from one phase to the next, drive other transformations. For example, the Static View at the Analysis phase drives the transformations of the Communication, Dynamic and Processing Views from the Analysis phase to the Design phase. The same is applicable when moving from one iteration to the next. For example, the Static View at the Analysis phase of Iteration i influences the transformations of the Communication, Dynamic and Processing Views of the Analysis phase of Iteration $i+1$. This is ensured by a combination of Semantic Verification and Semantic Incorporation . The latter is illustrated in Fig. 7.9.

Note that in Fig. 7.9, we only show the Dynamic View from Analysis in Iteration i to the Dynamic View of Analysis in Iteration $i+1$.

* when we achieve `(1)`, `(1.1)` and `(4)`, we have ensured that the Static View at Iteration $i + 1$ has been incorporated with the Static View at Iteration i ;

* when we achieve `(2)`, `(3)`, `(2.1)` and `(3.1)`, we have ensured that Static View at Iteration i has influence the Dynamic View from Iteration i to Iteration $i+1$, therefore Dynamic Semantic Incorporation of the Dynamic View has been achieved (i.e. line labeled as `(5)` in Fig. 7.9).

Therefore, Dynamic, Communication and Processing Semantic Incorporation are transitively obtained and do not require to be separate testing activities. We provide a list of heuristics that can be applied to double check that validation has indeed occurred. Semantic Incorporation of the Static View is however very important. We concentrate our effort in the latter.

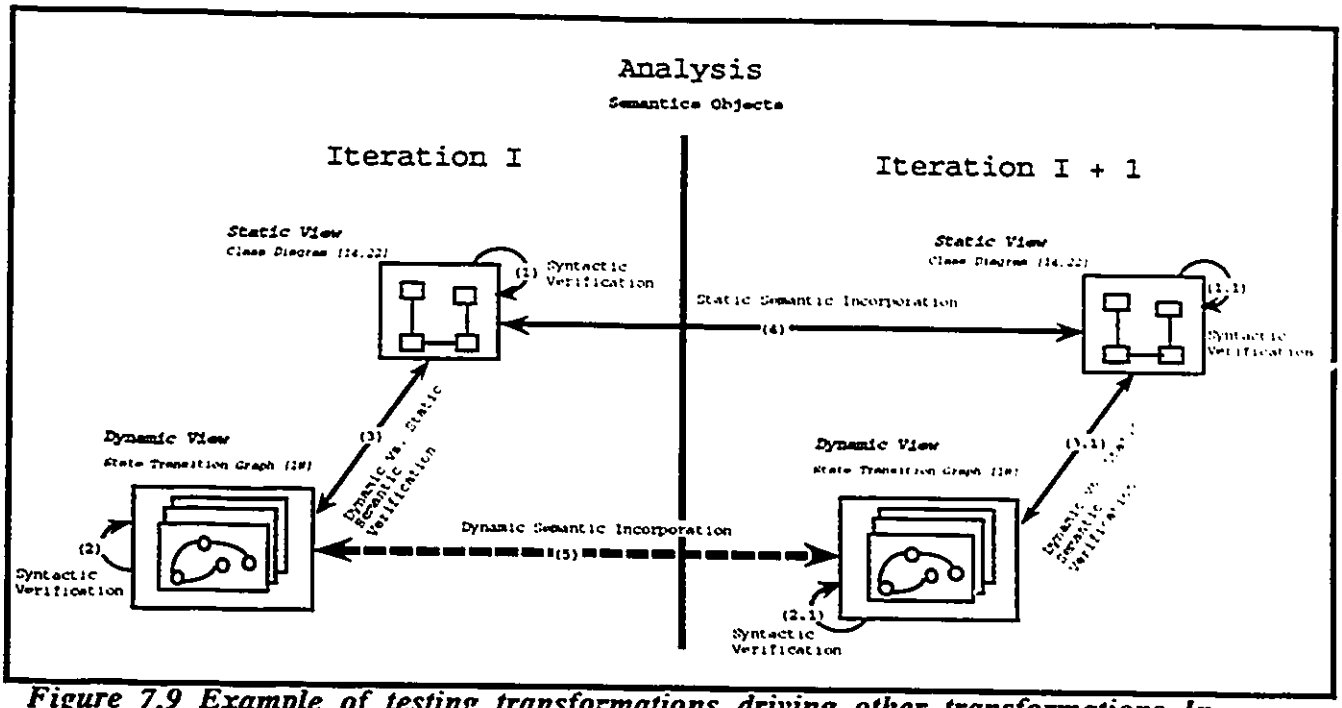


Figure 7.9 Example of testing transformations driving other transformations In Incorporation

7.3.1 Requirements Capture Incorporation

As described in section 5.2.1, the incorporation as well as the verification process (described in section 6.3.1) of the Requirements Capture phase is not the focus of the work presented, therefore we briefly introduce it. This activity fall under Specification Based testing techniques discussed in that section.

Text and notations that fall under the Scenario Model (discussed in section 4.3.3.1) are usually used to capture requirements. This forms a specification document.

Text can capture different aspects of the application to be developed (e.g. performance, data, architecture, behavior, etc.). Notations that fall under the Scenario model, provides aspects of the Communication View, since every use of the system can be viewed as communication between the User and the System.

Testing for incorporation is highly coupled to the type of specification used to capture requirements. Specification Based testing techniques, aim at detecting incomplete, inconsistent and misleading requirements. An example is illustrated in [51]. When requirements are capture as notations in the Scenario Model, we must ensure that the Uses Cases [18] or Scenarios [14] identified in iteration i are incorporated and do not invalidate the ones derived in iteration $i+1$. Assessing Use Cases and Scenarios for their incorporation still remains a topic of research.

7.3.2 Development Incorporation

As discussed at in section 7.3 Static Incorporation is the most important testing activity in Incorporation. Incorporation is used in the context of Iterations. The are two basic reasons to move to a next iteration. Either, more of the same is required (i.e. more detailed functionality is required) or new functionality is required. In both cases, the process of Incorporation aims at ensuring Forward Iteration traceability and Backward Iteration traceability.

Static Semantic Incorporation is very similar to Static Semantic Validation as illustrated in Fig. 7.10. The first occurs in the context of different Iterations, the latter occurs in the context of different Phases. That is, the valid transformation of the Static View from one Iteration to the next are very similar to the valid transformations of the Static View from one Phase to the next. The latter were discussed in section 7.2.2.1.

The set of heuristics to determine if Communication, Dynamic and Processing Incorporation was transitively achieved are the same as the set of heuristics to determine if Communication, Dynamic and Processing Validation was transitively achieved. These were discussed in sections 7.2.2.2, 7.2.2.3 and 7.2.2.4 respectively.

To clarify our point of view, the reader can obtain Fig. 7.3 by rotating Fig. 7.9 90 degrees clockwise.

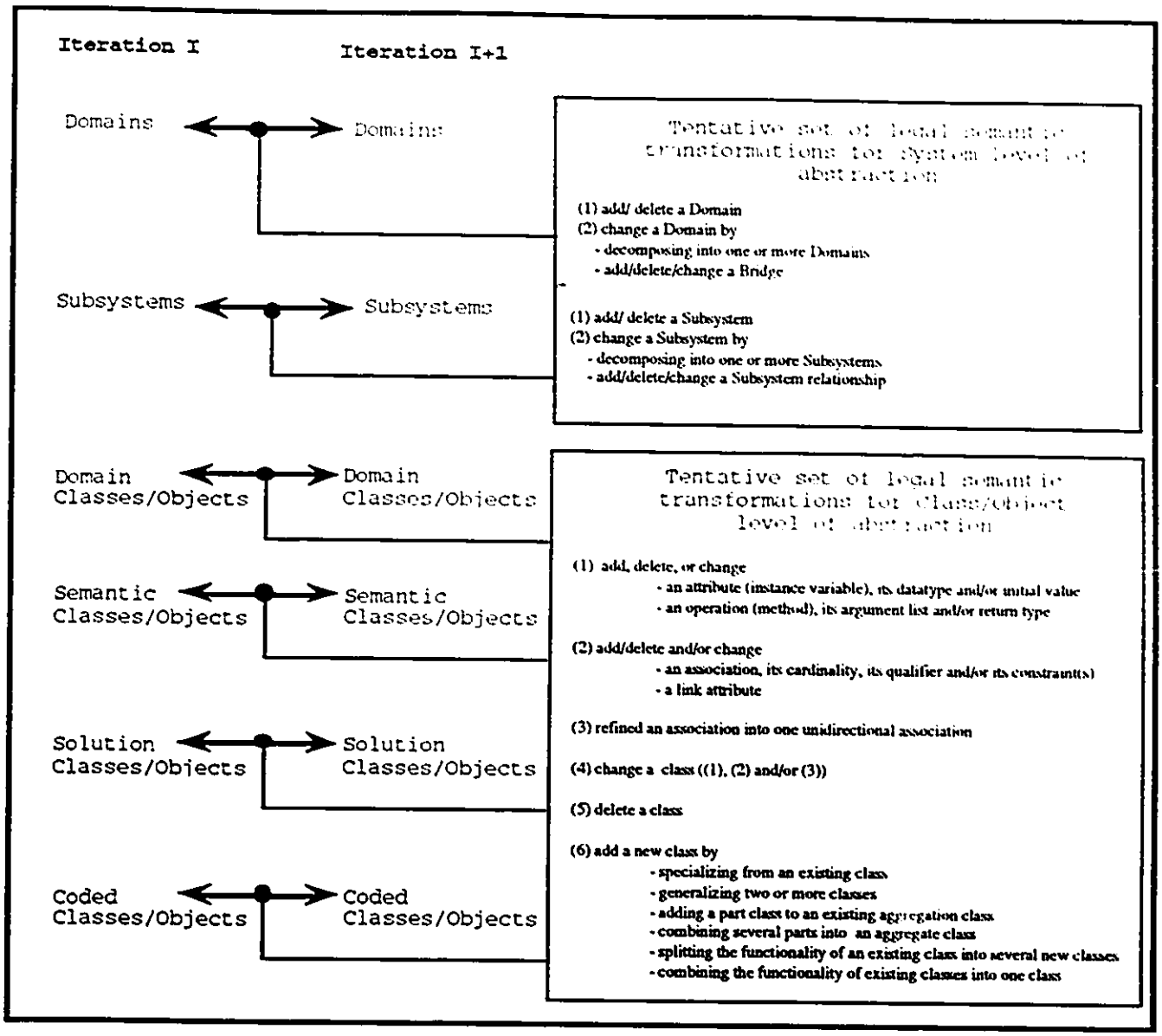


Figure 7.10 Generalized static semantic transformations in Incorporation

Chapter 8 The Generic Integrated Object-Oriented Testing and Development Methodology

8.1 Introduction

The purpose of the chapter is to combine the work of Chapters 4, 6 and 7 and produce the Generic Integrated Object-Oriented Testing and Development Methodology.

In Chapters 6 and 7, we introduced testing activities that should be incorporated to the Generic Object-Oriented Software Development Methodology of Chapter 4, in order to increase the testability of implementations. This integration results in the Generic Integrated Object-Oriented Testing and Development Methodology. We will show how the work in [8] is extended in basically two ways: (1) by incorporating testing activities described in Chapters 6 and 7 and (2) by incorporating a manner of dealing with change in a consistent manner (i.e. error correction, enhancements, new requirements, etc.) to achieve the traceability described in Chapter 3, (this was described in Chapter 4).

8.2 The Generic Integrated Object-Oriented Testing and Development Methodology

The Generic Integrated Object-Oriented Testing and Development Methodology is based on the work by the authors [8] and can be best described as *analyze a little, test a little, design a little, test a little, code a little, test a little and analyze a little,....* Testing activities should be closely coupled or integrated with development activities at every phase of the software development process.

As stated in Chapter 1, the Generic Integrated Object-Oriented Testing and Development Methodology is composed of a Process (described in Chapter 4) and a set of Development (defined in Chapter 4) and Testing phases (defined in Chapters 6 and 7) as defined by the Process. The definition also includes the integration ordering of the phases (e.g. Analysis followed by Verification followed by Design) and specifics points in software development where change (i.e. error correction, enhancements, new requirements, etc.) should take place.

8.2.1 The Integration ordering of Development, Verification, Validation and Incorporation

In Chapters 5, we introduced Verification, Validation and Incorporation. Fig. 8.1, illustrates the integration of these testing activity types with development activities and their ordering of occurrence. Note that Validation must be taken in the context of iteration as illustrated in observation 1 of section 7.2. Every testing activity type is blown up, as described in the next pages.

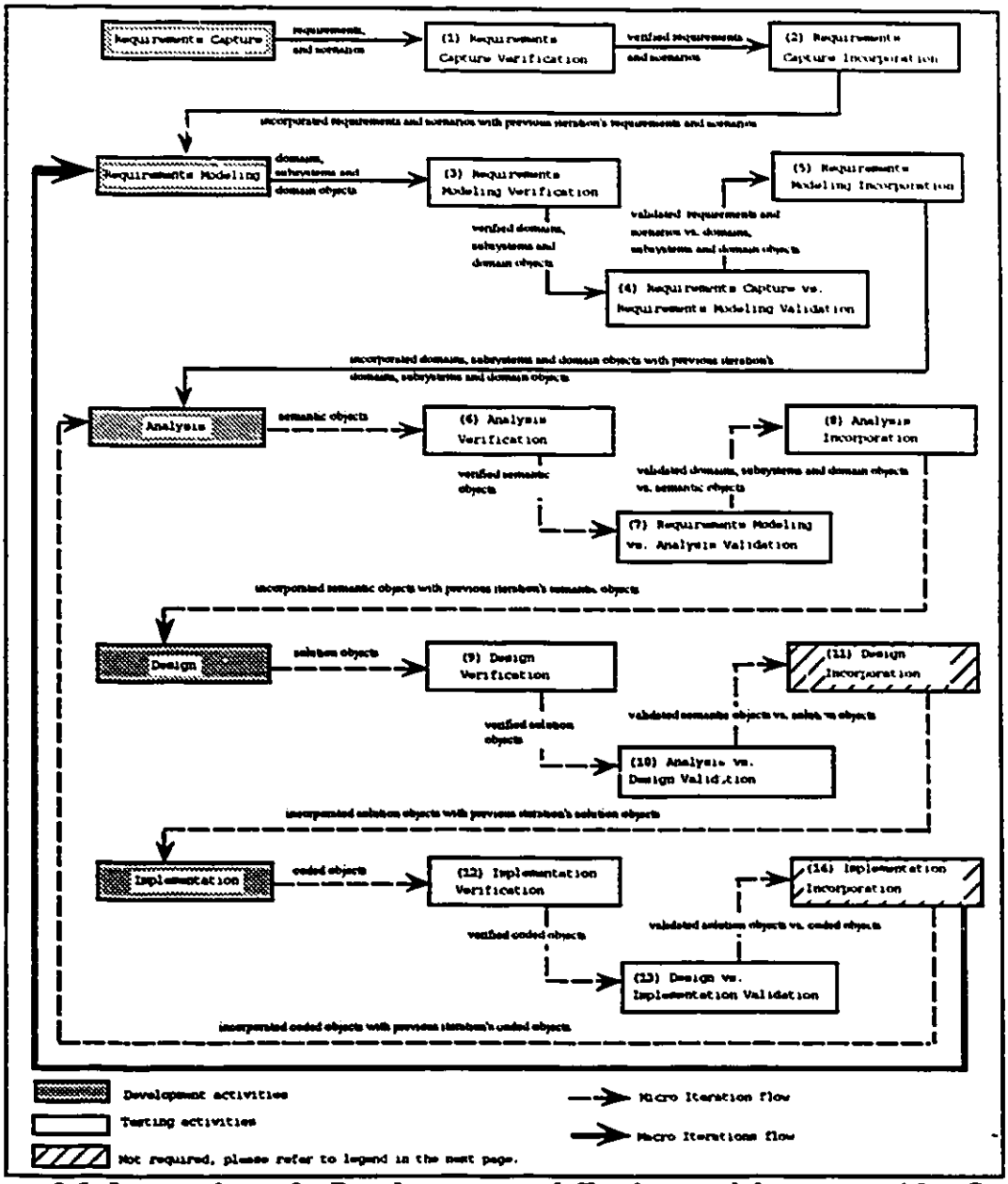


Figure 8.1 Integration of Development and Testing activity types (the Generic Integrated Object-Oriented Testing and Development Methodology)

(1) Requirements Capture Verification

corresponds to an instance of class Requirements Capture Verification in Fig. 5.7 .

(2) Requirements Capture Incorporation

corresponds to an instance of class Requirements Capture Integration in Fig. 5.7.

(3) Requirements Modeling Verification

includes an instance of the following classes for Requirements Modeling notations:

- Syntactic Verification in Fig. 5.7
- Static/Communication Semantic Verification in Fig. 5.7

(4) Requirements Capture vs. Requirements Modeling Validation

corresponds to an instance of class Requirements Capture vs. Requirements Modeling Validation in Fig. 5.7.

(5) Requirements Modeling Incorporation

includes an instance of the following classes for Requirements Modeling notations of iterations i and $i+1$

- Static Semantic Integration in Fig. 5.7
- Communication Semantic Integration in Fig. 5.7

(6) Analysis Verification

includes an instance of the following classes for Analysis notations:

- Syntatic Verification in Fig. 5.7
- Static/Communication Semantic Verification in Fig. 5.7
- Static/Dynamic Semantic Verification in Fig. 5.7
- Dynamic/Communication Semantic Verification in Fig. 5.7
- Static_Dynamic_Communication/Processing Semantic Verification in Fig. 5.7

(7) Requirements Modeling vs. Analysis Validation

includes an instance of teh following classes for Requirements Modeling and Analysis notation.

- Static Semantic Validation in Fig. 5.7
- Communication Semantic Validation in Fig. 5.7

Note that the second testing activity type corresponds to the guidelines discussed in section 7.2.2.2.

(8) Analysis Incorporation

includes an instance of the following classes for Analysis notations of iterations i and $i+1$

- Static Semantic Integration in Fig. 5.7
- Communication Semantic Integration in Fig. 5.7
- Dynamic Semantic Integration in Fig. 5.7
- Processing Semantic Integration in Fig. 5.7

(9) Design Verification

includes an instance of the following classes for Design notations:

- Syntatic Verification in Fig. 5.7
- Static/Communication Semantic Verification in Fig. 5.7
- Static/Dynamic Semantic Verification in Fig. 5.7
- Dynamic/Communication Semantic Verification in Fig. 5.7
- Static_Dynamic_Communication/Processing Semantic Verification in Fig. 5.7

(10) Analysis vs. Design Validation

includes an instance of the following classes for Analysis and Design notations:

- Static Semantic Validation in Fig. 5.7
- Communication Semantic Validation in Fig. 5.7
- Dynamic Semantic Validation in Fig. 5.7
- Processing Semantic Validation in Fig. 5.7

Note that the latter three testing activity types corresponds to the guidelines discussed in section 7.2.2.2, 7.2.2.3 and 7.2.2.4 respectively.

(11) Design Incorporation

The aim of this testing activity is to ensure that Design notations for the current Micro-iteration are integrated with Design notations of the previous Micro-iteration. This is already achieved transitively as follows.

Analysis vs. Design Validation (testing activity number (10)) ensured that Design notations for the current Micro-iteration are derived from Analysis notations of the current Micro-iteration.

Furthermore, Analysis Integration (testing activity number (8)) ensured that Analysis notation for the current Micro-iteration are integrated with Analysis notations of the previous Micro-iteration.

Therefore, Design notations for the current Micro-iteration are derived from integrated Analysis notations.

(12) Implementation Verification

includes an instance of the following classes for Implementation notations (i.e code):

- Syntatic Verification in Fig. 5.7
- Static/Communication Semantic Verification in Fig. 5.7
- Static/Dynamic Semantic Verification in Fig. 5.7
- Dynamic/Communication Semantic Verification in Fig. 5.7
- Static_Dynamic_Communication/Processing Semantic Verification in Fig. 5.7

(13) Design vs. Implementation Validation

includes an instance of the following classes for Design and Implementation notations (i.e. code)

- Static Semantic Validation in Fig. 5.7
- Communication Semantic Validation in Fig. 5.7
- Dynamic Semantic Validation in Fig. 5.7
- Processing Semantic Validation in Fig. 5.7

Note that the latter three testing activity types corresponds to the guidelines discussed in section 7.2.2.2, 7.2.2.3 and 7.2.2.4 respectively.

(14) Implementation Incorporation

The aim of this testing activity is to ensure that Implementation notations (i.e. code) for the current Micro- iteration are integrated with Implementation notations of the previous Micro-iteration. This is already achieved transitively as follows.

Design vs. Implementation Validation (testing activity number (13)) ensured that Implementation notations for the current Micro-iteration are derived from Design notations of the current Micro-iteration.

Furthermore, Design Integration (testing activity number (11)) ensured that Design notation for the current Micro-iteration are integrated with Design notations of the previous Micro-iteration. (This was a transitive testing activity type).

Therefore, Implementation notations (i.e. code) for the current Micro-iteration are derived from integrated Design notations.

We present a brief example to further illustrate the ordering of the above testing activities and more specifically why certain testing activity types such as numbers (11) and (14) are not required.

Consider Fig. 8.2. Assume that at Analysis and Design of iteration *i*, the notations used to provide the Communication, Static and Dynamic Views are Interaction Diagrams [18,22], Class Diagrams [14,22] and State Transition Diagrams [18] respectively (we have not included the Processing View, for clarity of the picture). Assume also that Analysis notations have undergone, Verification, Validation and Incorporation and that Design notations have been generated. We are currently in the process of testing these Design notations (note that we do not concentrate on how or when a notation should be generated from another notation; the numbering scheme denotes ordering of testing activities and not development activities).

At Design, each notation should first undergo Syntactic Verification as indicated by the directed lines labeled as (1). As stated in Chapter 6, this testing activity type aims at verifying that notations use the correct syntax.

The next testing activities are Static/Communication Semantic Verification, Dynamic/Communication Semantic Verification and Static/Dynamic Semantic Verification as indicated by the directed lines labeled as (2), (3) and (4) respectively. As stated in Chapter 6, these testing activities aim at verifying the semantic completeness and semantic correctness of notations.

At this point, the Design notations have been verified. The next step is to verify that these notations are indeed derived from Analysis notations. Thus the process of validation can start. Design notations will undergo Static Semantic Validation, Dynamic Semantic Validation and Communication Semantic Validation as indicated by the directed lines labeled as (5), (6) and (7) respectively. As stated in Chapter 7, these testing activities aim at verifying the semantic consistency of notations. The latter two testing activities correspond to the guidelines described in sections 7.2.2.3 and 7.2.2.2 respectively.

The next step is to verify that Design notations are integrated with the previous iteration's Design notations. This was depicted as testing activity type number (11) in Fig. 8.1. In Fig. 8.2, the dashed lines indicate that Analysis notations have been integrated with the previous' Micro-iteration Analysis notations via Incorporation testing activity types. Since the process of Validation of Design notations, illustrated by the directed lines labeled as (5), (6) and (7), ensured that Design notation are derived from the integrated Analysis notation, Design notations are transitively integrated with the previous Micro- iteration Design notations.

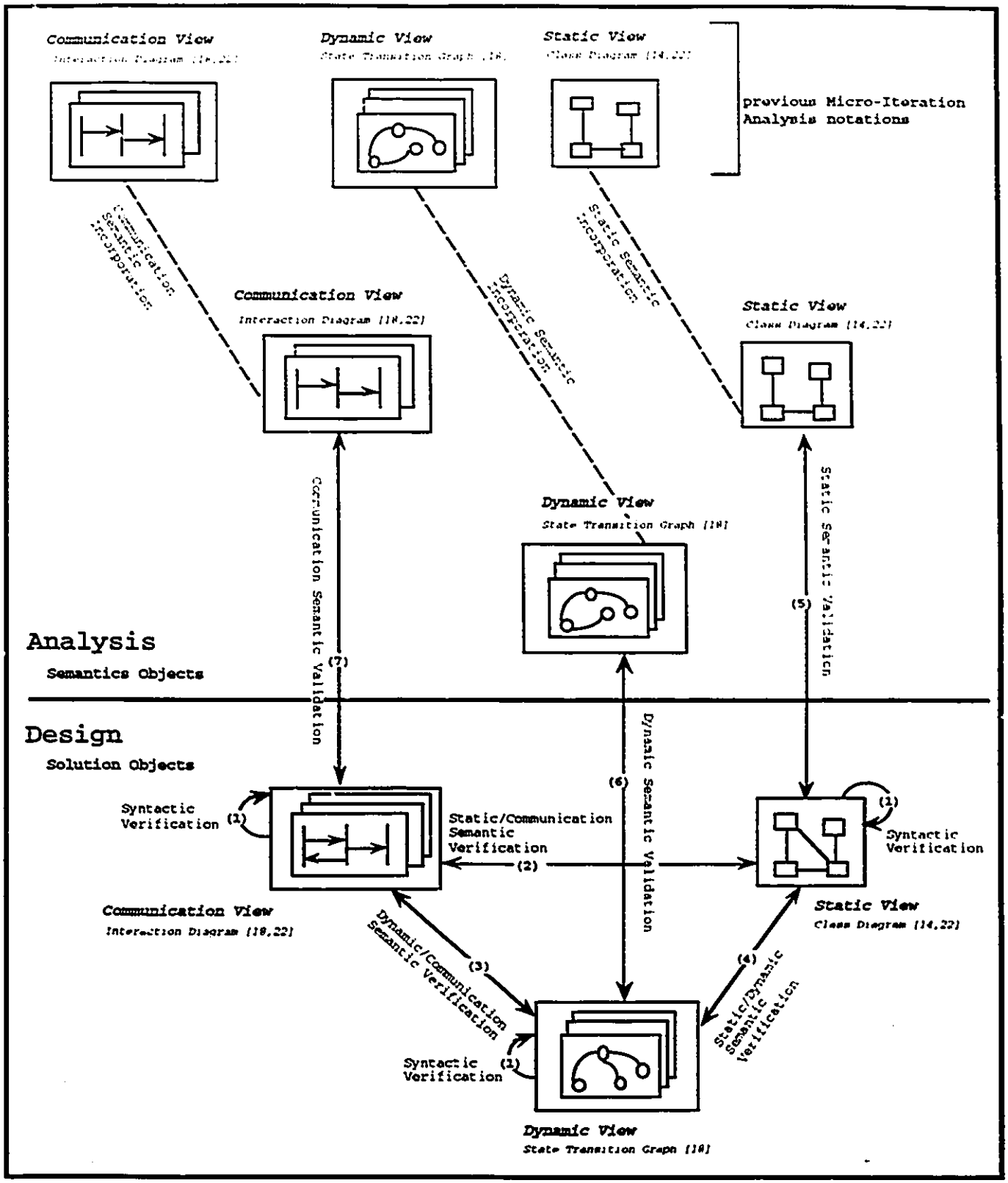


Figure 8.2 Example of ordering between Verification, Validation and Incorporation testing activities

8.2.2 Handling change in the Generic Object-Oriented Software Development Methodology

In Chapter 4, in Fig. 4.8, we described the Generic Object-Oriented Software Development Methodology to accommodate change (i.e. error correction, enhancements, new requirements, etc.). The purpose of the Methodology was to provide a consistent manner of dealing with change.

The testing activity types described in Chapter 5 will generate changes to notations used at different phases of the software development process. In order to handle these changes in a consistent manner, we propose to integrate the process of handling changes described in Chapter 4 (section 4.3.1.3) with the Generic Integrated Object-Oriented Testing and Development Methodology described in the previous section.

Lets look in details at how changes should be handled in order not to loose traceability. We will later clarify our discussion with an example.

Verification testing activity types will most likely generate changes to notations of a specific phase in a given iteration, simply because of the nature of the testing activity type[35]. These changes will not affect other phases, since verification aims at detecting errors in notations of a specific phase (e.g. Analysis). Thus, these changes should be applied to the notations of the tested phase (e.g. Analysis) immediately, that is before moving to the next phase (e.g. Design).

Validation testing activity types will most likely generate changes to consecutive phases of a given iteration simply because of the nature of the testing activity type[36]. These changes will affect either or both of the consecutive phases (e.g. Analysis and Design), since validation aims at detecting errors in notations of consecutive phases. Thus, the changes should be applied to the notations of the consecutive tested phases (e.g. Analysis and Design) immediately, that is before moving to the next phase (e.g. Implementation).

---

[35] Verification aims at determining whether Notations used at phase *n* of a given iteration are complete and correct with respect to other Notations at phase *n* of the same iteration and whether these Notations are correct in themselves.

[36] Validation allows to determine whether Notations used at phase *n* (e.g. Analysis) are consistent with respect to Notations used at phase *n+1* for a given iteration (e.g. Design) and whether Notations used at phase *n+1* (e.g. Design) are consistent with respect to Notations used at phase *n* for the same iteration.

Incorporation testing activity types will most likely generate changes to the same phase of consecutive iterations simply because of the nature of the testing activity type[37]. These changes will affect either or both phases in the consecutive iterations (e.g. Analysis at iteration *i* and Analysis at iteration *i+1*), since Incorporation aims at detecting errors in notations of the same phase in consecutive iterations. Changes affecting the notations of the tested phase in the current iteration (i.e. *i+1*) should be applied immediately; changes affecting the notations of the tested phase in the previous iteration (i.e. *i*) should be stored in appropriate repositories.

We now illustrate our discussion with an example. In Fig. 8.3, assume that we are currently at the Analysis phase of a given iteration. Note that Fig. 8.3, only illustrates the flow of changes due to testing activity types and not the flow of changes due to development activities; the latter were discussed in section 4.3.1.3.

The `Analysis` phase is the next activity after `Requirements Modeling Incorporation` as illustrated in Fig. 8.1. While performing `Analysis Verification`, changes to the `Analysis` phase should be applied immediately to analysis notations as indicated by the dashed line labeled as `1`. As explained above, these changes do not affect other phase, therefore they should be addressed before moving to the next development activity (i.e. `Design`).

The next testing activity type is `Requirements Modeling vs. Analysis Validation`. Since it validates notations of the current iteration's `Requirements Modeling` phase and `Analysis` phase, changes to these phases should be applied immediately, before moving to the development activity (i.e. `Design`). This is indicated by the dashed lines labeled as 2 and 3.

The next activity type corresponds to `Analysis Incorporation`. It aims at ensuring that the analysis notations of the current Micro-iteration are integrated with the analysis notations of the previous Micro-iteration. Changes to the current Micro-iteration analysis notations should be applied immediately, before moving to the `Design` phase. This is indicated by the dashed line labeled as 4. Changes to the previous Micro-iteration analysis notations, should **not** be applied immediately. They should be stored in repository `(2)` in order to be addressed as part of the next cycle of change for the current Macro-iteration (this was discussed in section 4.3.1.3).

---

[37]Integration refers to the set of activities that allow to determine if Notations at phase *n* of iteration *i* are integrated in notations at phase *n* of iterations *i+1* and whether Notations at phase *n* of iteration *i+1* are integrated in notations at phase *n* of iterations *i*.

The reason for this is that if changes are applied directly to the previous Micro-iteration analysis notations, nothing guarantees that other phases (i.e. Design and Analysis) of the previous Micro-iteration, that are affected by the changes, will be updated accordingly. This violates one of the principles of traceability (i.e. traceability within a given Micro-iteration).

The same idea is used for other phases of the Generic Integrated Object-Oriented Testing and Development Methodology. In this manner, we ensure traceability

(1) within a Macro/Micro iteration,

(2) from a Macro/Micro iteration to the next and

(3) from a Macro/Micro iteration to its previous iteration.

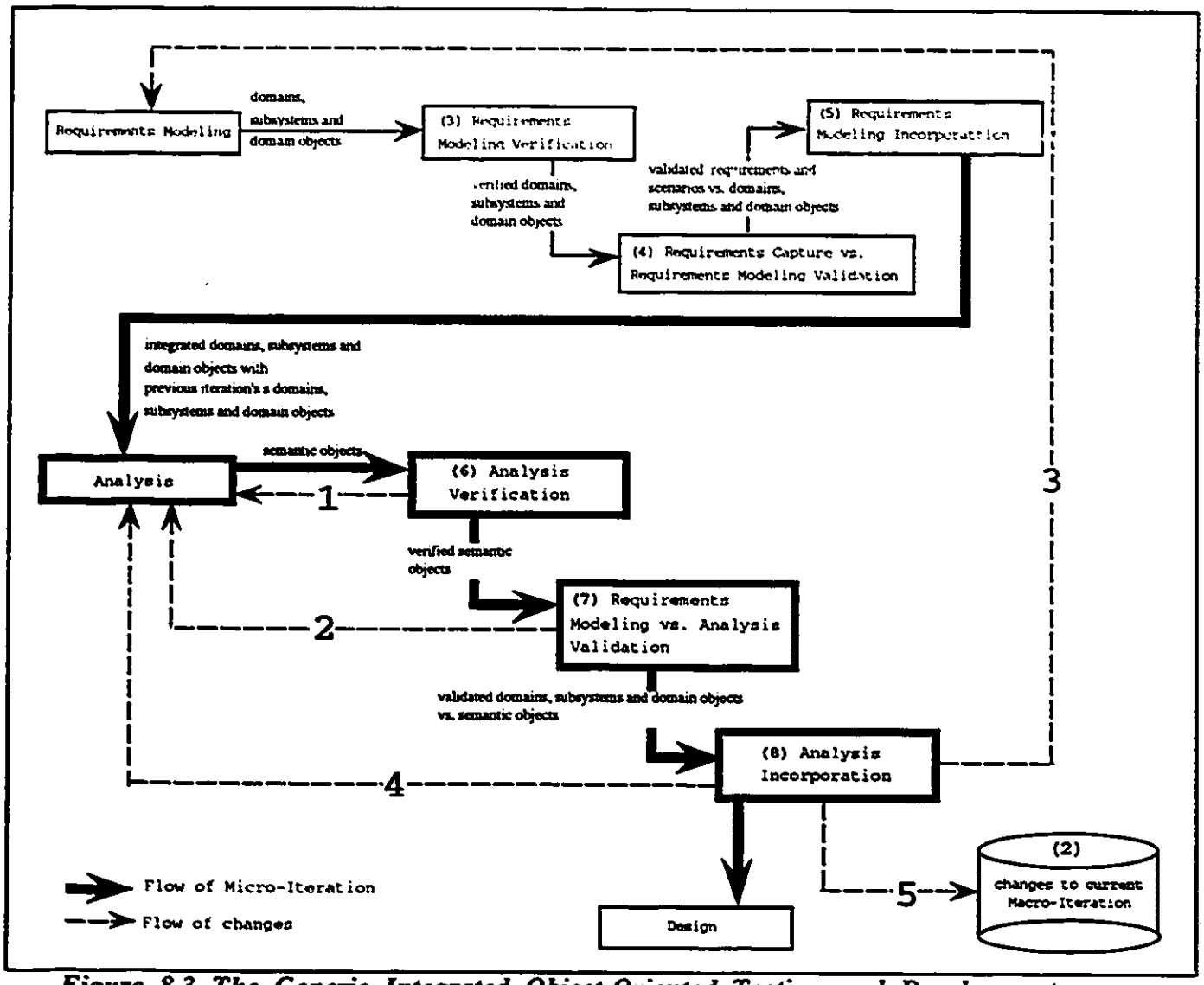


Figure 8.3 The Generic Integrated Object-Oriented Testing and Development Methodology (Design)

Chapter 9 Conclusion

Testability is a measure of how easy (less complex, less tedious, less boring, less costly) the effective testing of implementations is made.

In Object-Oriented development, the author in [67] conceives, testability as a result of (1) the characteristics of OO implementations, (2) the test support environment, (3) the characteristics of representations (i.e. notations) and (4) the software process in which testing is conducted. In this thesis, we examined in details the last two factors.

9.1 What did we achieve and why

(1) the software process in which testing is conducted

When addressing the software process in which testing is conducted, we proposed an integration of development and testing activities. This leads to our proposed Generic Integrated Object-Oriented Testing and Development Methodology. Our work was based on the work in [8,67] but augmented in what we believe are the two major omitted areas: first, the process should enforce a consistent method of dealing with change (i.e. error correction, enhancements, new requirements, etc.); second, the process should determine what are the testing activities that should take place at different phases of the OO software development (this is discussed in the next section).

Providing a generic way of handling change in a consistent manner such that testability attributes are not lost, is crucial. The usual tendency in software engineering is to accommodated changes at the phase where they are discovered, this leads to a loss of traceability (i.e. if a change to Requirements notations is identified at Design, the usual tendency is to modify the Design, perhaps update the Requirements and then continue development from Design on). This results in *spaghetti development* (Design --> Requirements --> Design) which in turn causes the loss of traceability of notations within a given iteration and between iterations. Once traceability is lost, it is no longer possible to determine if a notation is derived from or traceable to another tested notation, thus questioning the effort spend testing the latter notation, since the errors corrected can reoccur in the derived notation.

To avoid this type of situation, we introduced database repositories that capture changes to non-consecutive phases of a given iteration. These stored changes are addressed at the next iteration; while in the next iteration, if they do not cause changes to other phases, they should be applied immediately, otherwise they should be stored again in appropriate repositories to be addressed at yet the next iteration; in this manner changes bubble-up in the development process, ensuring that all phases will capture the changes.

(2) the characteristics of representations

When addressing the characteristics of notations, the author in [67] defines the basic characteristics of notations to be verifiability, modifiability, traceability, completeness, unambiguity and consistency. These characteristics are mentioned in [67] but no definition is provided. Our contribution is this area is to provide concrete definitions of the above characteristics taking into account the iteration style of development of Object-Oriented systems. We divided, the characteristics into *testability requirements* and *testability attributes*. Notations that conform to testability requirements can exhibit testability attributes. We determined that the basic testability requirements are *verifiability*, *modifiability* and *traceability*, and that the basic testability attributes are *completeness*, *correctness*, *consistency* and *incorporation*. This is because a notation that is *verifiable* can be tested for testability attributes of *completeness*, *correctness*, *consistency* and *incorporation* ;a notation that is *modifiable* can accommodate changes to achieve testability attributes and finally a notation that is *traceable* exhibits testability attributes.

The rational for using notations that exhibit the testability requirements of *verifiability* and *modifiability* is that these notations can be assessed for consistency, correctness, completeness and incorporation via testing activity types. This provides the ability of (1) uncovering errors early in the process where they are less costly to fix, (2) generating test cases that can be applied to the implementation, (3) guiding the testing of the implementation and (4) facilitating the location of source(s) of error(s) for modification.

We identified testing activity types that are decoupled from the syntax of notations and from development methodologies in order to avoid their proliferation; testing activity types define what needs to be tested (i.e. different semantics) as well as coverage criteria; they are generic in the sense that they are refinable (adaptable to different levels of detail) in order to be uniformly applied at all phases of development.

In order to arrive to generic testing activity types, it was necessary to examined carefully the semantics that notations provide at different phases of development as well as the level of abstraction of this semantics at that specific phase. We concluded that different notations can be used to describe a specific level of abstraction and that within that level of abstraction, these different notations can be grouped based on the semantics they provide. This classification was called a Model. A Model is thus a classification of notations based on the level of abstraction and semantics they provide. Furthermore Models can themselves be grouped into Views by considering only the semantics they provide. We arrive to four basic Views provided by notations: Static, Communication, Dynamic and Processing. We based testing activity types on these Views. The following testing activity types were proposed:

in the area of **Verification** (i.e. testing involving a single phase of the software development process), we introduced

- Static/Communication Semantic Verification
- Static/Dynamic Semantic Verification
- Dynamic/Communication Semantic Verification
- Static_Dynamic_Communication/Processing Semantic Verification

the latter testing activities types ensure the testability attributes of *semantic completeness* and *semantic correctness* of notations to achieve our proposed testability requirement of Horizontal Traceability.

in the area of **Validation** (i.e. testing involving two consecutive phases of the software development process), we introduced

- Static Semantic Validation
- Communication Semantic Validation
- Dynamic Semantic Validation
- Processing Semantic Validation

the latter testing activities types ensure the testability attribute of *semantic consistency* of notations to achieve our proposed testability requirements of Vertical Upward and Vertical Downward Traceability.

in the area of **Incorporation** (i.e. testing involving two same phases in consecutive iterations of the software development process), we introduced

- Static Semantic Incorporation
- Communication Semantic Incorporation
- Dynamic Semantic Incorporation
- Processing Semantic Incorporation

the latter testing activities types ensure the testability attribute of *semantic incorporation* of notations to achieve our proposed testability requirement of Forward Iteration and Backward Iteration Traceability.

Furthermore, notations should be tested for their syntax. For this, we introduced the testing activity type: Syntactic Verification in order to ensure the testability attribute of *syntactic correctness* of notations so as to achieve our proposed testability requirement of Horizontal Traceability. Note that the latter is the only testing activity type dependent on the syntax of notations.

Finally, we also proposed that requirements should be tested in the context of iterations. In this area, we proposed the following testing activity types:

- Requirements Verification (ensure that requirements are complete and correct)
- Requirements Validation (ensure that requirements are addressed at the first phase of development, i.e. Requirements Modeling)
- Requirements Incorporation (ensure that new or updated requirements identified at a given iteration are addressed as part of the next iteration)

9.2 Assessment

The intend of this section is to present an assessment of our work.

(1) Assessment of our proposed way of dealing with change

The proposed way of dealing with change via the use of repositories (i.e. partitioning of a database) is generic, applicable and scalable.

Generic in the sense that the number of repositories and their distribution is based on the degree of traceability desired in the software development project and is therefore user defined. For example, in section 4.3.1.3, we wanted to keep a very detailed level of traceability (i.e. within and between Micro-Iterations and within and between Macro-Iterations). For this, we introduced 4 repositories. If we would have wanted traceability only within and between Macro-Iteration, only 3 repositories would have been needed.

Applicable in the sense that our proposed way of dealing with change can be adapted to several iteration styles. We illustrated this in section 4.3.1.2.3 were we configured a process of change for 4 different iteration styles.

Scalable in the sense that our propose way of dealing with change can be implemented via the use or repositories. In general, the number of repositories required depends on the desired degree of granularity of traceability, the number of phases in the software development process and the iteration style. The basic guideline when configuring a process of change, is that sufficient repositories to capture changes to non-consecutive phases of a given iteration should be created. The stored changes are addressed at the next iteration; while in the next iteration, if they do not cause changes to other phases, they should be applied immediately, otherwise they should be stored again in appropriate repositories to be addressed at yet the next iteration; in this manner changes bubble-up in the development process, ensuring that all phases will capture the changes.

We emphasize that our proposed way of dealing with change through the use of repositories accommodates itself well to metric collection. For example, the number of cycles of change (i.e. the number of times that notations of a given phase of an iteration need to be revisited before moving to the next iteration) can be an indicator of the complexity of the models addressed at that phase or the level of stability of the iteration goal (i.e. constantly changing goals for an iteration will lead to several cycles of changes until the goal of the iteration is stable). Also the level of complexity or stability of one-iteration over another can be determined based on the cycles of changes for each iteration. Overall, this can indicate the level of complexity of the software development and its level of stability (e.g. constantly changing requirements).

(2) Assessment of our proposed testing activity types

The proposed testing activity types are generic to a certain extent, applicable and scalable.

Generic since they do not depend on the syntax of notations (i.e. they are based on abstract constructs of a group notations: generic constructs of Views), and can be adapted to different phases of the software development process and levels of abstraction. A testing activity type can be applied to a notation as long as the notation's constructs can be translated into the generic modeling constructs of the View it provides. For example, if a new notation provides the Static View of a system and its modeling constructs can be translated into the modeling constructs of the Static View, then the notation can be tested by Static Semantic Validation and Static Semantic Incorporation. Furthermore, the notation can be tested in conjunction with other notations that provides other Views via Verification testing activity types such as Static/Communication Semantic Verification. The latter encourages new notations to be accompanied with the View they provide since this directly determines the set of testing activity types that notations can be subjected to. This provides a metric for how testable notations are. However, if a new notation provides one of the four Views but its modeling constructs cannot be translated to the generic modeling constructs of the View it provides, then this notation cannot be accommodated in our framework of View and testing activity types. This is why we claimed that testing activity types are generic to a certain extent. Even though our framework was based on notations widely used in both academia and industry as illustrated in Chapter 4, nothing prevents a new notation to use modeling constructs not identified by our Views. In this case, new testing activity types must be created to test the new notations. We leave this a future topic of research.

Our proposed testing activity types are applicable, since they can be applied informally to the software development process as a checklist of what to verify at each phase. We provided examples in Chapter 5 to test the applicability of the testing activity types. As a software designer involved in a large industry project, I tend to informally verify the notations that I generate at each phase of the software development. This is done in a very ad-hoc manner with really no indication for completeness (i.e. when I can feel confident that I have checked my notations fully). Having a checklist as defined by testing activity types brings organization to the whole process of testing notations. We now have concrete reference point for reviewing notations.

Testing activity types can be scalable since they are implementable. This is discussed in the next section.

Finally, different notations have different strengths for modeling, by performing cross Validation and Verification, we believe that we are likely to identify oversights or errors.

Note that our proposed way of dealing with change as well as the testing activities introduced are applicable to Software Engineering in general and are not only applicable to Object Oriented software development. However only OO offers multiple views of the system to be developed.

(3) A high level tool's proposal

Fig. 9.1 presents a very high level tool's proposal to illustrate how testing activity types can be used.

The upper box illustrates a process outside of our Testing tool proposal (e.g. a CASE tool like Argos[55]). We assume that Syntactic Verification (i.e. testing the syntax of the notation's modeling constructs) is done at entry time, by the CASE tool that provides the editor for the notation. The CASE tool should provide a parser: `Translation to Model modeling constructs`, which generates two outputs: a set of abstract Model modeling constructs and templates that contain specific information about the notation's modeling constructs). These templates were presented in [73]; they capture two types information. First, they capture information about the notation's modeling constructs, when the notation is considered as a stand-alone unit. Example of such information for the Design Class Specification template includes:

- the name of the class
- the type of class : abstract or concrete
- a unique identifier
- a description
- inheritance information: superclasses, subclasses.
- an instance variable table
- a method table

The second type of information that these templates capture is information that requires other notations (i.e. traceability links to other notations and test cases to test the traceability between the notations).

Going back to Fig. 9.1, the templates in the upper box (i.e. CASE tool) capture the first type of information only, since the notation is considered as a stand-alone unit.

The generation of Model modeling constructs should be performed by the CASE tool since it is notation specific. The Model modeling constructs were presented in section 4.3.3 where we showed that notations can be grouped into a Model depending on the semantics provided by the notation and the level of abstraction of this semantics.

The Model modeling constructs are input to our Testing tool and are translated into View modeling constructs based on the View provided by the notation. Note that this process is not notation specific but Model specific. It is only if when a new Model is generated that the translation mechanism needs to be updated.

The View modeling constructs can now be tested by the appropriate selected testing activity types. The implementation of a testing activity type (i.e. a method type), is shown in the inner box. As the modeling constructs of the View the notation provides are tested, the Templates are updated with the second type of information described above (i.e. traceability links and test cases to test the traceability links).

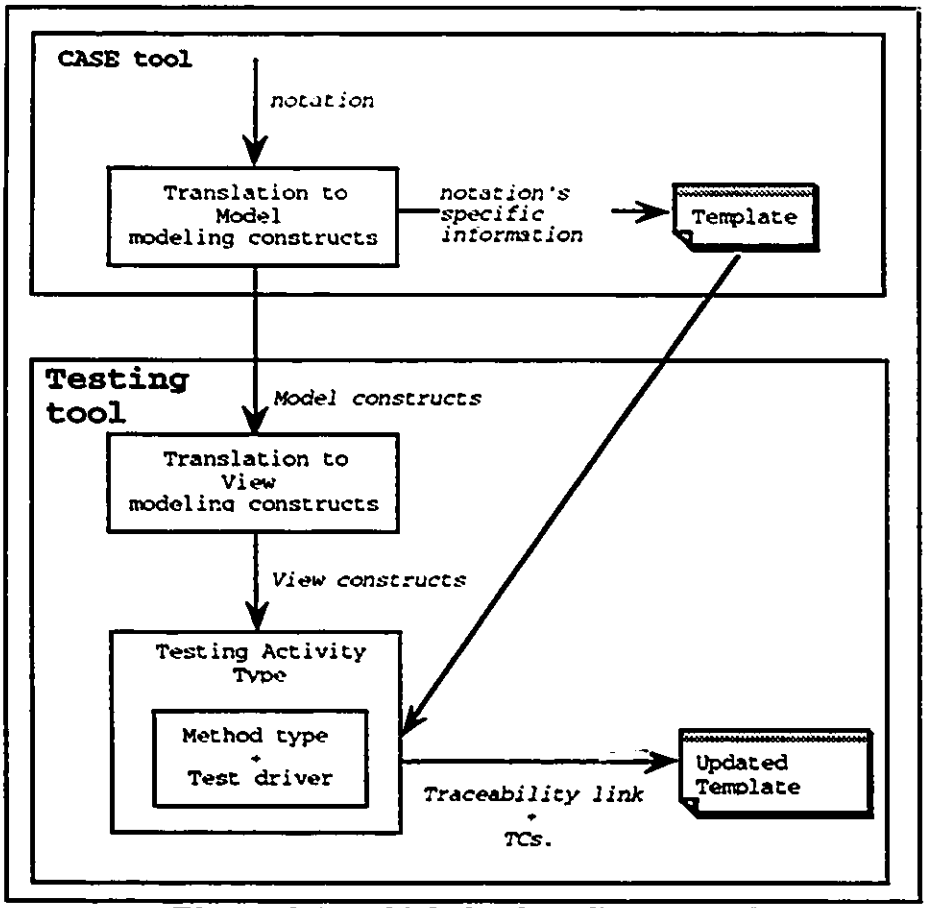


Figure 9.1 A high level tool's proposal

By augmenting the templates in [73], via including different types of traceability, we can directly correlate the author's traceability links and our work. In this manner:

- Verification, which provides Horizontal traceability within one iteration, provides Horizontal Traceability Links (HTL). The latter is an augmentation to the templates.

- Validation, which provides Vertical Upwards and Vertical Downwards traceability within an iteration, can provide the *Backward* (Upward in our case) and *Forward* (Downward in our case) Traceability Links described in [73]. We will refer to these as UTLs and DTLs, respectively.

- Incorporation, which provides Forward Iteration and Backward Iteration traceability between two consecutive iterations, can provide Forward Iteration and Backward Iteration traceability links. We will refer to these as FTLs and BTLs, respectively. The latter is an augmentation to the templates.

Lets take an example. Fig. 9.2, illustrates the discussion that follows.

At the top left corner of the figure, the modeling constructs of a Class Diagram [14, 22] for the Analysis phase, are translated into the Class Configuration Model modeling constructs. This classification was presented in section 4.3.3. As the Class Diagram is parsed, the Design Class Specification templates described above are populated with available information such the name of the class, the type of class, a unique identifier, a description etc.

The Class Configuration Model modeling constructs can now be translated into Static View modeling constructs. This classification was presented in section 4.3.4.

At the top right corner of the figure, the modeling constructs of Interaction Diagrams [18, 22] for the Analysis phase, are translated into the Object Interaction Model modeling constructs. This classification was presented in section 4.3.3. As the Interaction Diagrams are parsed, the Interaction Diagram Specification templates are populated with available information. This template was presented in [73].

The Object Interaction Model modeling constructs can now be translated into Communication View modeling constructs. This classification was presented in section 4.3.4.

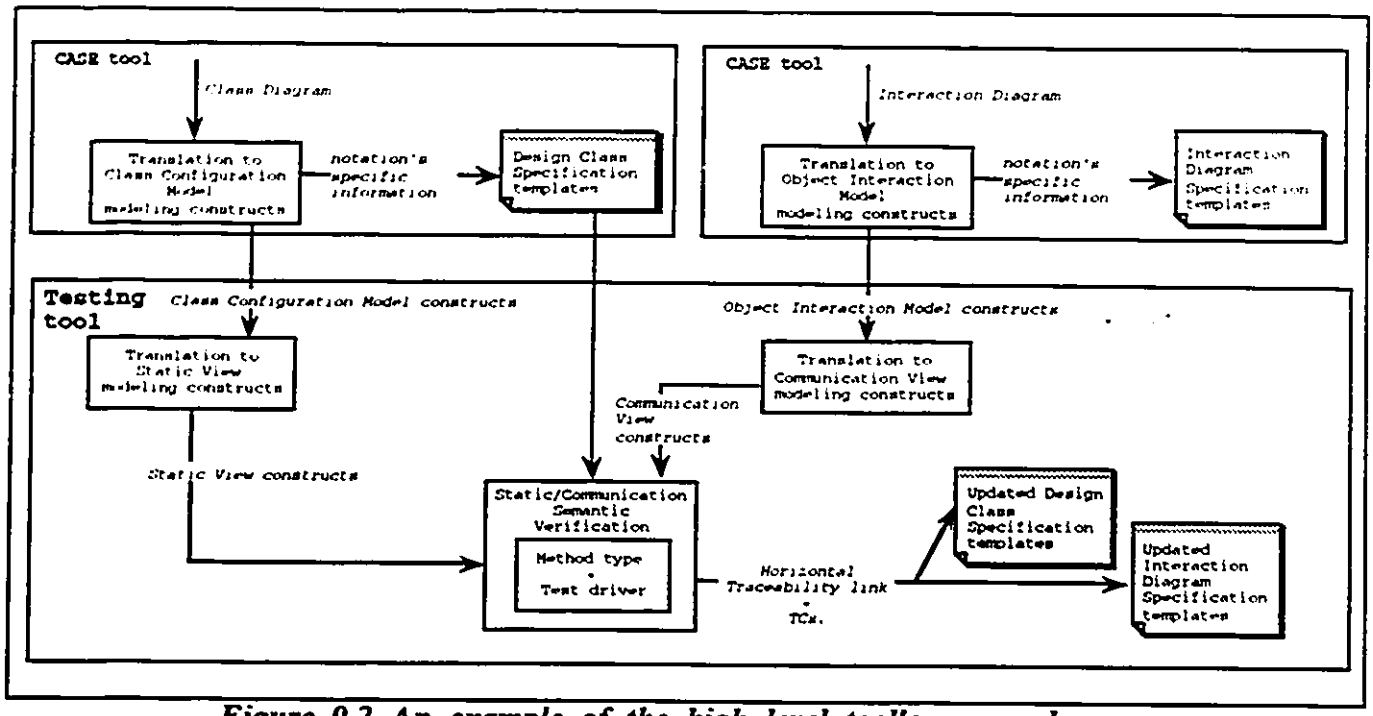


Figure 9.2 An example of the high level tool's proposal

Having the Static View and Communication View Modeling constructs, the Static/Communication Semantic Verification testing activity type can take place. These will generate HTLs between the two notations. In this example:

- every relationship between two classes in the Class Diagram will have an HTL to an aspect of an Interaction Diagram (i.e. part of, one or more than one Interaction Diagrams) describing which operation and their ordering satisfy the relationship (in the Interaction Diagram this is described as message exchanges).

- every Interaction Diagram will have a HTL to the relationships that it implements (i.e. a relationship is implemented as a set of operations on the participating classes).

In the same manner other notations will generate other traceability links. For example, if instead of Interaction Diagrams, a Class Diagram notation generated at the Design phase would have been used, then Static Semantic Validation would have generated UTLs and DTLs.

9.3 Future Research

We identified three areas for future research.

First, one of the motivations for the work presented in this thesis was the need for a concrete methodology for performing testing activities early in the Object-Oriented software development process. This concrete methodology included three aspects. First, a *process* describing when testing activities should occur (i.e. "when"). Second, the *testing activities* that should take place in the process (i.e. "what"). Third, the *methods* that implement the different testing activities (i.e. "how"). We addressed the "when" and the "what". The "how" still remains a topic research. However, we did provide the basis for identifying methods. We defined the concept of a *method type* which corresponds to the implementation of a *testing activity type*. By implementation we understand: the test case design for the *testing activity type* (e.g. use of Black and White Box testing techniques [59]); how the *testing activity type* will be executed (e.g. how to observe and control the notations being tested); test case reports, etc. The idea of a *method type* is interesting since it is notation independent and aims at testing the semantics of notations without having to relying on code generation.

Second, the testing activity type: Requirements Verification which aims at ensuring that requirements are complete and correct, was not the focus of the work presented. Testing semantic completeness and correctness of requirements is highly coupled to the type of specification used to capture these requirements. Specification Based testing techniques, described briefly in section 5.2.1, aim at detecting incomplete, inconsistent and misleading requirements. When requirements are capture as notations in the Scenario Model, we must ensure that as new Uses Cases [18] or Scenarios [14] are identified, they do not invalidate already derived ones. Overall, assessing Use Cases and Scenarios for their completeness and correctness, still remains a topic of research.

Third, to complete the study of testability in Object-Oriented development, the two remaining areas not addressed by our work were (1) the characteristics of OO implementations, (2) the test support environment. These two areas should be carefully examined.

In the area of characteristics of Object-Oriented implementations, the author in [67] provides metrics to assess the testability of implementations, but no definite guidelines of what the characteristics of implementations should be in order to achieve this testability are provided (i.e. we can measure the testability, but we are not sure how to achieve it). Interesting areas of research are the trade-offs between guidelines (e.g. weakly coupled classes are desirable; inheritance is also desirable for reusability purposes but it tightly couples superclasses and their subclasses) and whether guidelines could be language dependent.

In the area of test support environment, the support environment should not only include support in terms of test implementations (i.e. built-in tests, test suites, test tools etc.), but also support in terms of test organization. Examples of the latter include the management commitment to ensure that sufficient resources (staff, time, funding) are allocated for testing, establishing formal channels of communication between teams performing the testing activities (e.g. testers) and the team performing notation changes as a result of the testing activities (e.g. analysts, designers), promoting the value of traceability within and between Macro/Micro Iterations and finally creating a mind set in the organization of commitment to quality.

Also, we only briefly touched on the area of test case generation from Notations (section 5.2.2) as well as on the area of how testing Notations can direct or guide the testing of implementations (section 5.2.3).

Test case construction (i.e. coverage, correctness issues, etc.), still remains a topic research, but we propose that test case construction should follow the same principles of traceability as development, that is, a Unit Test Case at the Implementation phase should be traceable to its Cluster test case at Implementation, its Unit Test Case at Design, its Cluster Test Case at Design, its Unit Test Case at Analysis, its Cluster Test Case at Analysis, its System Test Case and finally its Requirements Test Case.

In the area of how testing Notations can direct or guide the testing of implementations, we propose that the ordering of invocation of operations (which is a challenge when testing object-oriented implementations as described in section 5.2.3 due to the Randomness of Operation problem) should be identified from the Static View of Notations. More specifically , the ordering of operations should be derived so as to validate the semantics of relationships between classes, as illustrated in Example 1 of Chapter 5. The idea is to identify a Client and a Server for every relationship between classes. The relationship becomes the function that a Server must provide to its Clients. There are two areas the require careful investigation.

First, the semantics of Aggregation is sometimes not clear. For example, it becomes difficult to derive a sequence of operations (and therefore to identify which class is the Client and which is the Server) to validate an Aggregation relationship, when it is not clear whether the Aggregation is mandatory in both directions (i.e. every Whole must have a Part and every Part must be part of a Whole [75]); whether the Whole can exists with or without its Parts; whether the Parts can exits with or without their Wholes, etc. Also the cardinality of relationships between classes imposes yet another challenge.

Second, when dealing with Inheritance, it is not straight forward to validate inherited relationships. If a relationship was validated at the Superclass level, this does not necessarily mean that the same relationship exists for every one its subclasses, since the subclass may not implement all the inherited operations. (i.e. a Subclass does not necessarily participates in the same relationships as its Superclass). The problem, thus becomes one of completeness (i.e. when do we know that all relationships of a subclass have been validated?).

Finally, it is useful to investigate whether our work could be ISO 9000 compliant. For this, specific metrics would need to be added.

Bibliography

[1] J. A. Lewis et all., "An Empirical Study of the Object-Oriented Paradigm and Software Reuse", Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1991, October 1991, pp. 184-196.

[2] R. V. Binder et all., "Object-Oriented Paradigm Software Testing", Communications of the ACM, September 1994, Vol. 37, No. 9, pp. 29.

[3] D. Firesmith, "Testing Object-Oriented Software", Proceeding of the 11th. TOOLS USA Conference, pp. 407-426, Santa Barbara, Aug. 1993.

[4] B. Dewayne, E. Perry and G. E. Kaiser, "Adequate Testing and Object Oriented Programming", Journal of Object-Oriented Programming, Vol. 1, No. 1, April/May 1988.

[5] M. D. Smith and D. J Robson, "Object-Oriented Programming- the Problem of Validation", Proceedings: Conference on Software Maintenance, pp. 272-281, San Diego, CA, November, 1990.

[6] M. J. Harrold and McGregor, "Incremental testing of Object-Oriented Class Structures", Proceedings of the Fourteenth International Conference on Software Engineering, 1992.

[7] R. L. Probert, "Grey-Box (Design Based) Testing Techniques", Proc. of the 15th Hawaii International Conference on System Science, pp. 94-102, 1982.

[8] J. D. McGregor and T. D. Korson, "Integrated Object-Oriented Testing and Development Processes", Communications of the ACM, September 1994, Vol. 37, No. 9, pp. 59-77.

[9] E. Berard, "Tutorial notes: Testing Object-Oriented Software", Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1992.

[10] B. Bohem, "A Spiral model of Software Development and enhancement", ACM Sigsoft Software Engineering Notes, Vol. 11, No. 4, August, pp. 14.

[11] P. S. Deepinder, L. Ting-Kau, "Formal Methods for Protocol Testing: A Detailed Study", IEEE Transactions on Software Engineering, Vol. 15. No. 4, April 1989.

[12] R. Saracco and P. A. J. Tilanus, "CCITT SDL: Overview of the Language and its Applications", Computer Networks and ISDN Systems, Vol. 13, 1987, pp. 65-74.

[13] D. Harel, "State Chart: a visual formalism for complex systems", Science of Computer Programming, Vol. 8, 1987, pp. 231-274.

[14] J. Rumbaugh et all, "Object-Oriented Modeling and Design", Prentice-Hall Inc., 1991.

[15] D. De Champeaux, et all. "Object-Oriented System Development", Addison-Wesley Publishing Company. 1993.

[16] N. Hazeltine, Panel on "Managing the transition to Object-Oriented Technology", Object-Oriented Programming Systems, Languages and Applications (OOPSLA), OOPS Messenger, Vol. 3, No. 4, October 1992.

[17] B. Cameron, C. Geldrez et. all, "Software Architecture at BNR", Workshop Proceedings of OOPSLA 94, Precise Behavioral Specifications in Object-Oriented Information Modeling, October 1994.

[18] I. Jacobson, "Object-Oriented Software Engineering, A Use Case Driven Approach", Addison-Wesley Publishing Company. 1992.

[19] S. Shlaer and S. J. Mellor, "Object-Oriented system analysis, Modeling the World in data", Prentice-Hall Inc., 1988.

[20] B. Meyer, "Object-Oriented Software Construction", Prentice-Hall Inc., 1988.

[21] G. Myers, "Advances in Computer Architecture", John Wiley and Sons, 1982.

[22] G. Booch, "Object-Oriented Analysis and Design with applications", The Benjamin/Cummings Publishing Company, Inc., 1994.

[23] L. L. Constantine, "Objects, Functions and Program Extendibility", Computer Language, Vol. 7 , No. 1, 1990, pp. 34-56.

[24] S. Shlaer and S. J. Mellor, "Object Lifecycles, Modeling the World in States", Prentice-Hall Inc., 1992.

[25] F. Barbier, "Object-Oriented Analysis of systems through their dynamical aspects", Journal of Object-Oriented Programming, May 1992, pp. 45- 51.

[26] F. Hayes and D. Coleman, "Coherent Models for Object-Oriented Analysis", Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1991.

[27] G. M. Hoydalsvik and G. Sindre. "On the purpose of Object-Oriented Analysis", Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1993.

[28] D. De Champeaux, D. Lea, and P. Faure, "The Process of Object-Oriented Design", Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1992.

[29] J. M. Nerson, "Applying Object-Oriented Analysis and Design", Communications of the ACM, Vol. 35, No. 9, September 1992, pp. 63-74.

[30] D. De Champeaux, "Object-Oriented Analysis and Top-Down Software Development", ECOOP, 89.

[31] R. Wirfs-Brock et. all, "Designing Object-Oriented Software", Prentice-Hall Inc., 1990.

[32] B. Henderson-Sellers and J. M. Edwards, " Object-Oriented Systems Life-Cycles", Communications of the ACM, Vol. 33, No. 9, September 1990, pp. 143-159.

[33] "Panel: Methodology Standards: Help or Hindrance?", Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1994.

[34] D. De Champeaux and P. Faure, " A comparative study of object-oriented analysis methods', Journal of Object-Oriented Programming, March/April 1992, pp. 45- 51.

[35] D. E. Monarchi and G. I. Puhr, "A research Typology for Object-Oriented Analysis and Design", Communications of the ACM, Vol. 33, No. 9, September 1992, pp. 35-47.

[36] I. J. Walker, "Requirements of an Object-Oriented design method", Software Engineering Journal, March 1992, pp. 012-113.

[37] D. F. D'Souza, "Tutorial notes: A Comparison of Object-Oriented Analysis and Design methods", Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1994.

[38] O. I. Lindland, G. Sindre and A, Solvberg, "Understanding Quality in Conceptual Modeling", IEEE Software, March 1994.

[39] J. Rumbaugh, "Tutorial notes: The OMT Method", Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1994.

[40] R. Helm, I. M. Holland and D. Ganopadhyay, "Contracts: Specifying Behavioral Composition in Object-Oriented Systems", Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1990.

[41] Ince, "Test cases", Systems International, Vol. 17, No. 4, pp. 60-62, April 1989.

[42] C. Robach and S. Guibert, "Testability Measures: a review", Computer System Science and Engineering, vol. 3, no. 3, pp. 117-126, July 1988.

[43] J. Turino, "Designing for Testability", Proceedings of the technical program: National Electronic Packaging and Production Conference: NEPCON West '89, pp. 1525-1534, March 1989.

[44] M. Hare and S. Sicola, "Testability and Test Architectures" IEEE Region 5 Conference 1988: Spanning the peaks of electrotechnology, pp. 161-166, March 1988.

[45] A. DeSena, "Military standard defines method of designing testability into systems", Computer Design, pp. 116-117, October 1988.

[46] L. Ungar, "A successful implementation of a military standard for testability", ATE & Instrumentation Conference East: for test & instrumentation users: TEST 1988, pp. 197-210, June 1988.

[47] Test Strategy Development External Bell Northern Research, Ltd., Ottawa, "DSFT: Designing for Testability - Methods Development Strategy- ", October 3, 1990

[48] R. L. Probert and C. Geldrez, "Communications Systems Design for Testability" Canadian Conference on Electrical and Computer Engineering, Vol. 2, pp. 43.3.1-43.3.5, Ottawa, Sept. 1990.

[49] D. Hoffman, "Hardware testing and Software ICS", North West Conference on Software Quality.

[50] R. A. Kemmerer, "Testing Formal Specification to Detect Design Errors.", IEEE Transaction on Software Engineering, Vol. 12. No. 1, pp. 32-43, January 1985.

[51] J. C. Sampaio do Prado and P.A. Freeman, "Requirements Validation Through Viewpoint Resolution", IEEE Transaction on Software Engineering, Vol. 17. No. 12, pp. 1253-1269, December 1991.

[52] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M Politi, R. Sherman and A. Shtul-Trauning, "Statemate: A working environment for the development of complex reactive systems", Proceedings of the 10th International Conference on Software Engineering, Singapore, April 1988.

[53] A. Ek, Verifying Message Sequence Charts with the SDT Validator", SDL'93.

[54] F. Belina and G. Nilsson, "SDT, SDL Design Tool", SDL'87, Amsterdam, April 1987.

[55] Argos by Miramar Technology, S. A. de C. V. Ejercito Mexicano #501 Local 6 y 7. Cd. Madero, Tamaulipas 89140, Mexico

[56] M. D. Smith and D. J Robson, "A framework for testing Object-Oriented Programs", Journal of Object-Oriented Programming, pp. 45-53, June. 1992.

[57] L. Cardelli and P. Wegner, "On understanding Types, Data Abstraction and Polymorphism", *Computing Surveys*, Vol. 17, No. 4, December 1985.

[58] T. Budd, "An Introduction to Object-Oriented Programming" Addison-Wesley Publishing Company.

[59] G. J. Myers, "The Art of Software testing", John Wiley & Sons Inc., 1989

[60] T. D. Korson and J. McGregor, "Object-Oriented Analysis and Design Concepts", Software Architects.

[61] P. J. Robinson, "Hierarchical Object-Oriented Designs", Prentice-Hall, Object-Oriented series.

[62] M. Wolczko, "Encapsulation, Delegation and Inheritance in Object-Oriented languages", Software Engineering Journal, March 1992, pp. 95-101.

[63] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages", Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1986, September, pp. 38-45.

[64] A. Taivalsaari, "Object-Oriented Programming with Modes", Journal of Object-Oriented Programming, June 1993.

[65] J. Richardson and P. Schwarz, "Aspects: Extending Objects to Support Multiple, Independent Roles", ACM SIGMOD'91, Denver, Colorado, May 29-31, 1992, pp. 298-306.

[66] D. G. Kafura and K. H. Lee, "Inheritance in Actor Based Concurrent Object-Oriented Languages", ECOOP'89, pp. 131-145.

[67] R. V. Binder, "Design for Testability in Object-Oriented Systems", Communications of the ACM, September 1994, Vol. 37, No. 9, pp. 87-101.

[68] Requirements Traceability and Management (RTM) by Marconi System Technology Inc.

[69] R. S. Pressman, Software Engineering, A Practitioner's Approach, McGraw-Hill, Inc., 1987.

[70] Dynamic Object-Oriented Requirements System (DOORS) by Quality Systems Software.

[71] D. Coleman, F. Hayes and S. Bear, "Introducing ObjectChart or How to Use Statecharts in Object-Oriented Design", IEEE Transaction on Software Engineering, Vol. 18, No. 1, pp. 9-18, January 1992.

[72] K. Beck and W. Cunningham, "A laboratory for teaching object-oriented thinking", OOPSLA 89, Proceedings, SIGPLAN Notices, Vol. 24, Number 10, 1989.

[73] J.P. Corriveau, template work.

[74] I. Jacobson and M. Christerson, "A growing consensus on Use Cases", Journal of Object-Oriented Programming, Vol. 8, Number 1, March/April 1995.

[75] F. Civello, "Roles for composite objects in object-oriented analysis and design", Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 1993.

Appendix A Object-Oriented Programming Testing Challenges

The purpose of this appendix is to summarize the challenges of Object-Oriented Programming Testing.

1.1 Encapsulation and Information Hiding

Encapsulation was discussed in section 2.3.4: it is a technique for enforcing information hiding, where the interface and implementation of a program unit are syntactically separated [4]. Encapsulation prevents objects from seeing the inside view of each other: the access to an object is only provided through its interface. The advantage of encapsulation is that it narrows the possible interdependencies among objects (weak coupling). If an object changes the implementation (private features) of an operation but its interface remains unchanged, the change should be transparent to other objects using this operation as long as the new implementation of the operation is entirely compatible (same functionality) with the previous version.

The advantages of encapsulation in testing is that it is potentially easier to determine what is needed to be test under a change: the changed operation and perhaps objects that directly invoke the operation [4] (the latter is to verify that the functionality provided by the operation is the same as its previous version).

The problem with encapsulation in the context of testing is that it becomes more difficult for the tester to observe and control what is being tested without inspecting the underlying implementation. For example, how can *internal operations* be tested (operations that make the implementation of other operations cleaner)? How can the addition, deletion or change of an instance variable be observed and controlled? (Instance variables denote an object's state and are completely hidden within the object's implementation). The only way to access them (i.e. set, use, etc.) these variables is to invoke operations of the objects where the instance variable is located, that will act on the instance variable. Therefore how can the state of an object be queried or set to a specific value, or how can state changes be monitored without inspecting the underlying implementation.

Finally, in the concurrent model based on active objects in [35], an object is allowed to change state without being operated upon by another object (i.e. spontaneously). In this model it becomes even more difficult to monitor state changes and to determine if the object is changing its state spontaneously in a correct manner. This is because there cannot be any correlation of a state change with an operation invoked that caused the state change. Active objects can support *autonomous operations* (operations that issue a request for themselves). These operations may change the st of the active object. As in the case of internal operations, it becomes difficult to test these operations.

1.2 Randomness of Operations

The Von-Neuman model [21] of processing (input - process - expected output) is not appropriate for the object-oriented paradigm as explained in section 2.2. In the object-oriented paradigm, a message is sent or passed to an object requesting an operation on that object. In [56], the authors qualify this as active data and passive process as opposed to passive data and active processor of the Von-Neuman model.

A very interesting observation is made in [56]. The authors state that the testing process becomes a searching problem since programs (objects) are not executed sequentially as in the traditional Von-Neuman model. This is because operations in a class can be combined in arbitrary order. The testing thus involves searching for the order the operations can be executed with various parameters that yield errors. We will refer to this problem as the *randomness of operations* problem.

Consider an example of the randomness of operations problem from [56]. A class represents a clock object and provides operations to set and retrieve the time, increment and decrement the time by one second, and produce a display of the current time in a graphical form. There is no stipulation in the ordering that these operations should be invoke. The time may be retrieve a number of times in a timing exercise or set a number of times, if the user wishes to examine the effect of different time zones. Operations simply provide a service or a functionality. Given that an error may occur after a particular sequence of operation invocations of an object, the problem in terms of testing can be seen as a search for the correct order to uncover a specific error.

1.3 Localization

The author in [9] discusses that one reason why testing of object-oriented programs is different from the testing of more conventional software is *localization*. Localization is the process of placing items in close physical proximity to each other. He observes that in function/data paradigm (software created using functional decomposition), the localization is based on functionality and thus there is a high degree of correlation between "testing a function" and a "unit of software". In object-oriented paradigm, the localization of information is around classes or instances of these classes: objects. Thus a class or an object is the "unit of software". The correlation between "testing a function" and a "unit of software" is not one-to-one as in the function/data paradigm. A class or an object may provide many functions and one function may involve several classes or objects.

1.4 Object creation and destruction

This is an area that has not receive much attention. In general how can object creations and destructions be tested? Some languages like Smalltalk provide automatic garbage collection, therefore the destruction of objects may not be explicitly tested by the tester. Usually classes provide for the creation and destruction of objects, in order to test these operations, they should be invoked explicitly by the tester and implicitly by the code. Their consistency should be observed (i.e. whether the operations behave in the same manner, if invoked implicitly or explicitly). Sometimes the abnormal destruction of an object may raise an exception. The issue in terms of testing is one of controlling and observing object creations and destruction, specially if the language provides these operations.

1.5 Exceptions

An exception is a facility for dealing with errors or other exceptional conditions that arise during program execution. The main principle is that when a subprogram finds an unusual situation, it raises a named exception which is then processed in another part of the program called an exception handler [61].

In the context of object-oriented programming, it can be defined as the expected response of an object to an abnormal condition. An abnormal condition could be a message sent to an object to invoke a, operation not implemented by the object, inappropriate data supplied as parameter of a message (wrong number of parameters, wrong type, data out of range etc.), hardware failures etc.

Exceptions are usually handled by passing control back to the client object (calling object) or to a handler object which takes appropriate action.

Testing exceptions in the object-oriented paradigm does not necessarily bring new challenges then in the function/data paradigm. The challenge in testing exceptions is one of identifying the unusual conditions that the object should be placed under, specially when the exception is not developer-defined (through requirements regarding the exception). These unusual conditions are not easily identifiable by simply looking at the object-oriented ode. Also special care should be taken when coding exception test cases since they may disrupt the normal flow of the program execution.

1.6 Polymorphism and Dynamic Binding

Polymorphism and *Dynamic Binding* are two features of object-oriented programming that could make an unexpected object be the receiver of a message.

Polymorphism can be discussed at two levels: at the level of programming and at the level of the class relationships (Inheritance, Containment and Uses, these relationships are discussed in Chapter 5).

At the level of programming [57] discusses different types of polymorphism (Parametric, Inclusion, Overloading and Coercion).

We are concerned with two types of polymorphism that apply to the class relationships. These are discussed in [58]. When polymorphism exist in the context of inheritance is it refer to as *overriding*, when it exist without inheritance is it refer to as *overloading*.

Overriding, also refer to as Inclusion in [57], is the action that occurs when an operation in a subclass with the same name as an operation in a superclass takes precedence over the operation in the superclass. A special type of overriding is refer to as *deferred methods* (operation). A differed operation is an operation that defines an interface (arguments and results types), but no implementation. Implementation is provided by subclasses that override the deferred operation, preserving the interface. Overriding allows a name to denote different classes that are related by some common superclass. For example, the classes `Triangle` and `Square` inherit from the class `Polygon`. Each has their own version of operation `Perimeter`. If we declare a variable `figure` of type `Polygon` and we have instances of the class as `aPolygon`, `aTriangle` and `aSquare`, the variable `figure` could denote `aPolygon`, `aTriangle` or `aSquare`. Therefore, sending the message `Perimeter` to `figure` would invoke operation `Perimeter` depending on what `figure` is. We can think of `figure` as having the type of the value it contains. Note that sometimes this type of polymorphism is referred to as *object polymorphism* [56]. This is because a subclass can take the place of a superclass (not vice-versa). `Figure` is declared of type `Polygon`, it can take the value `aTriangle` or `aSquare`.

Testing with overriding is more complex due to the fact that the receiver of a message is not known until execution time. By looking at the code it would seem that the message is sent to an instance, but during execution, an instance of any subclass may receive the message. Also, in the function/data paradigm, the equivalent of overriding would be implemented as an explicit `case` or `if` statement:

```
figure : Polygon
CASE figure. type is
     Triangle:  Perimeter_Triangle
     Square:    Perimeter_Square
     Polygon:   Perimeter_Polygon
ENDCASE
```

As stated [18], this complexity (i.e. three decisions paths to test) is not evident from looking at object-oriented code. Therefore, White-Box Design Decision [59] coverage techniques are more difficult to apply since it is not possible from just looking at the code. to determine the number of possibilities and which unit (object) is being invoked.

Finally, with overriding, errors can sometimes be detected only at execution time. For example when a message is sent to a subclass to invoke an operation that was blocked from being inherited from its superclass(es) (operation cancellation, protocol incompatible inheritance [60]), a run-time error will occur. This is again difficult to observe from just looking at the code.

The other type of polymorphism discussed is overloading. An overloaded operation is an operation that can have different implementations in different classes who are not related by an inheritance relationship. Even though the name of the operations are the same (i.e. name compatibility), the number and type of parameter and return values are different. Name compatible operations can increase the confusion when testing but they do not bring in the problems of determining who the receiver or a message is, as in the case of overriding polymorphism.

Dynamic Binding is a mechanism for associating operations with objects. It is used to implement polymorphism and occurs when the binding (association) of an operation is based on the type of its value and not on the type of its declaration [58]. In terms of message passing, this means that it is not until the message is sent, that the message is bound to an operation at the receiver object.

To summarize, polymorphism and dynamic binding may introduce hard to find errors: an unexpected object acts as the receiver of a message not intended for it. Also it is not until run-time that the actual receiver of the message is known. As stated in [5], testing techniques such as symbolic execution become much more complex since actual actions performed, depend on-run-time conditions in a much more complex way than that determined by normal control-flow constructs in procedural programming.

1.7 Genericity

The authors in [5] describe that one of the problem in object-oriented programming is *genericity* (i.e. the capability of defining a class from an unspecified type). A *generic class* is class that serves as a template for other classes, it must be instantiated (its parameters must be filled in) before objects can be created from it. An example of a generic class is a generic link list class in which the type of entity which is stored in the list is not specified until the user of the class requires it. The type of entity stored in the list is passed as a parameter. It can be integers, reals, arrays, linked lists etc., provided that the type of variable used is storable in such a structure.

In [5], there are two problems that are identified when testing generic classes.

The first is that they can be considered as *abstract* superclasses (i.e. classes that cannot create objects). Therefore by changing the generic class, we may bring in similar problems as when a superclass is changed (as described in section 2.2.4), thus causing classes that are derived from the generic class to be retested.

The second problem is that of choosing a suitable actual type to replace the parameter type in the generic class. If a simple parameter is chosen, perhaps the class is not really being correctly tested. The authors suggest that it would be advantageous to specify constraints on the types which can legally be used, as opposed to parameters.

1.8 Delegation

Delegation is an alternative to inheritance. The principle difference between class-based inheritance and delegation is that the former relates classes whereas the latter relates objects. Delegation implies shared responsibility in the completion of a task. It allows an object receiving a request message to forward it to some other designated object for processing [42].

Fig. A.1 illustrates an example of an object A that delegate one of its tasks to object B. In a delegation-base system, there are two kinds of messages.

The first is present in all object-oriented systems and simply involves sending a message from one object to another possibly passing some parameters (i.e. operation invocation). They are depicted as `normal` and `private message` in Fig. A.1.

The second kind of message occurs when an object delegates a task to another object (i.e. `delegation message`, in Fig. A.1). The object that delegates is expected to be consulted if additional information is required (i.e. `message to customer` in Fig. 3.1).

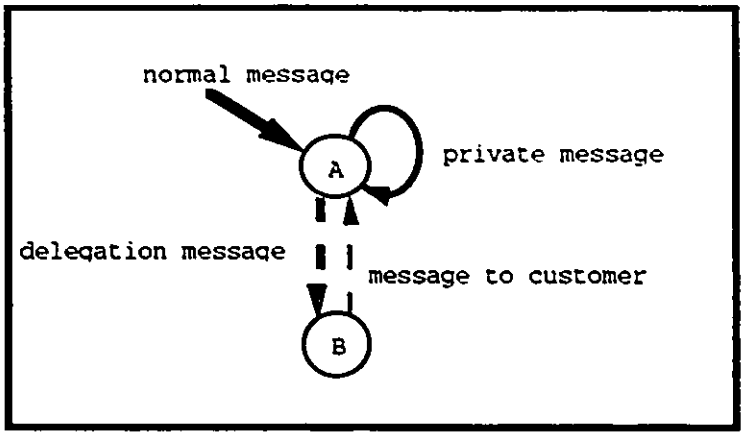


Figure A.1 Messaging in the delegation model

This model brings in new challenges for object-oriented programming. A tester must be able to differentiate between different types of messages as shown in Fig. A.1. Delegations messages and messages to customers need to be differentiate from normal and private messages because delegation brings a new set of possible control. For example, it is common in delegation to have an object only respond to some messages (i.e. messages from objects to which a task was delegated (i.e. B in Fig. A.1).

As stated in [62], the motivation for this is simple; delegation takes place when one object wishes to have another object cooperate to accomplish a task. To achieve this, the object being delegated to (B in Fig. A.1) may need privilege access; just as in real-life a subordinate may have a 'hot line' to their manager. Also an object may have operations that should only be invoked via delegation. If object A wishes to send a message m to object B, B may need privilege access to A in order to fulfill the request implied by m. B can be guarantee to have this privilege by making operation m executable only via delegation.

Another issue with testing delegation is one of *localizing communication* as described in [62]. Fig. A.2 illustrates an example. Consider the response of C to the message n delegated to it by B as part of B's response to message m. If C needs to communicate with the original customer (i.e. A), does it do this via B or directly to A? If B decides whether C sees B as the customer or B's customer (i.e. A in this case) then the interface to C in B should handle two cases (2 messages per say q and q').When q' is received, C is totally invisible to A. When q is received B must pass the message to A, thus making q part of its interface. This type of control flow among objects brought by Delegation is difficult to test unless very specific interfaces are provided.

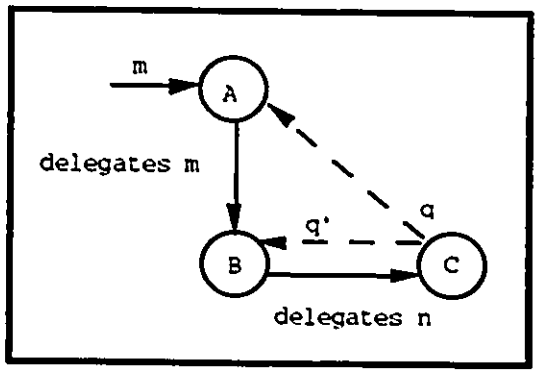


Figure A.2 Example of Localizing Communication

1.9 Inheritance

Inheritance can be seen as violating encapsulation because subclasses have visibility of the hidden encapsulated resources (instance variables and operations) of their superclasses. Thus, the designer of a class is allowed full access to the representation defined by a superclass [63].

When a superclass is modified it is necessary to retest all its subclasses since they depend on it. The implication of a changed superclass has effects on all its subclasses. As discussed in [63], permitting direct access to instance variables weakens one the major benefits of object-oriented programming: the freedom of the designer to change the representation of a class without impacting its clients classes (subclasses). These effects were discussed in section 2.2.4 of Chapter 2.

1.9.1 Multiple Inheritance

Multiple inheritance occurs when a subclass inherits from two or more superclasses. The example in Fig. A.3, taken from [5], illustrates the concept. The class `Play_House` inherits from the classes `Toy` and `Building`, operation `Cost` is inherited from both classes, which one should be inherited? The operations have the same name but very possibly different interfaces and implementations.

Multiple inheritance increases the potential for confusion (understanding the class hierarchy) because the superclasses may have instance variables and operations with the same name but with different interfaces and implementations.

Languages that make use of multiple inheritance have different conflict resolution techniques. Some compilers (i.e. Smalltalk) simply do not allow name clashes and reject the compilation of the class [22]. Other languages allow the renaming of the resources (instance variables and operations) so that there is no ambiguity. Others like C++ allow the name clash but all references to the resources must be fully qualify with the name of where the resource is found. Another possibility is to redefine the operations in the subclass (i.e. override `Cost` in `Play_House`. This is the approach taken by Trellis/Owl programming Language [63]. Typically a precedence ordering is defined, which linearizes the set of superclasses so that there is a unique selection.

The problem is that the designer may intend a different operations to be invoked then the one actually invoked by the compiler, thus leading to the invocation of the wrong operation.

1.9.2 Repeated Inheritance

Repeated inheritance occurs when a subclass is derived from the same superclass more then once. Fig. A.3 illustrates an example of repeated inheritance. The class `Jigsaw` inherits from both the class `Puzzle` and the class `Board_Game` , it is therefore inheriting twice from the class `Toy`. The problem with repeated inheritance is that an operation or instance variable could be inherited twice from the same class; for example the class `Jigsaw` inherits operation `Cost` from both `Board_Game` and `Puzzle`, if the operation `Cost` was not overridden by neither class, then it does not matter which one is selected, but if the class `Puzzle` overrides operation `Cost`, which `Cost` should be selected, the `Cost` of `Toy` inherited from `Board_Game` or the `Cost` overridden by `Puzzle`?

As in the case with multiple inheritance some languages treat duplicates as illegal, and the compilation is rejected. Other languages such as C++ allow the duplication of superclasses but require fully qualify names to refer to members of a specific copy of the superclass. Sometimes multiple references to the same class are treated as denoting the same class. Others, like CLOS use a class precedence list, which is calculated every time a new class is added, the class is flattened, duplicates are removed and the resulting hierarchy is resolved using single inheritance. The class is accepted if there are no cycles in the class dependencies [22].

As in the case of Multiple Inheritance, the designer may indent a different operation to be invoked then the one actually being invoked by the compiler.

In general Multiple Inheritance and Repeated Inheritance can make difficult to determine what is being inherited and from where it is being inherited [3].

1.9.3 Abstract classes

An *abstract class* is a class from which no instances (objects) can be created, but its subclasses have instances. They are developed with the main purpose of being inherited by other classes. The subclasses are expected to add to the structure and behavior of the Abstract class, usually by completing the implementation of its typically incomplete operation [22].

Since it is not possible to create an instance of an abstract class, how can an abstract class be tested? Usually they must be indirectly tested via their subclasses which are instantiable (i.e. concrete classes). However, this is not straightforward since an abstract class may in terms have several subclasses that are themselves abstract.

Thus, knowing that abstract classes must be tested indirectly via their subclasses, the problem is then to search for the first subclasses of the abstract class that are concrete classes. Then the problem is how to partition the hierarchy to test the abstract class in terms of these descendants.

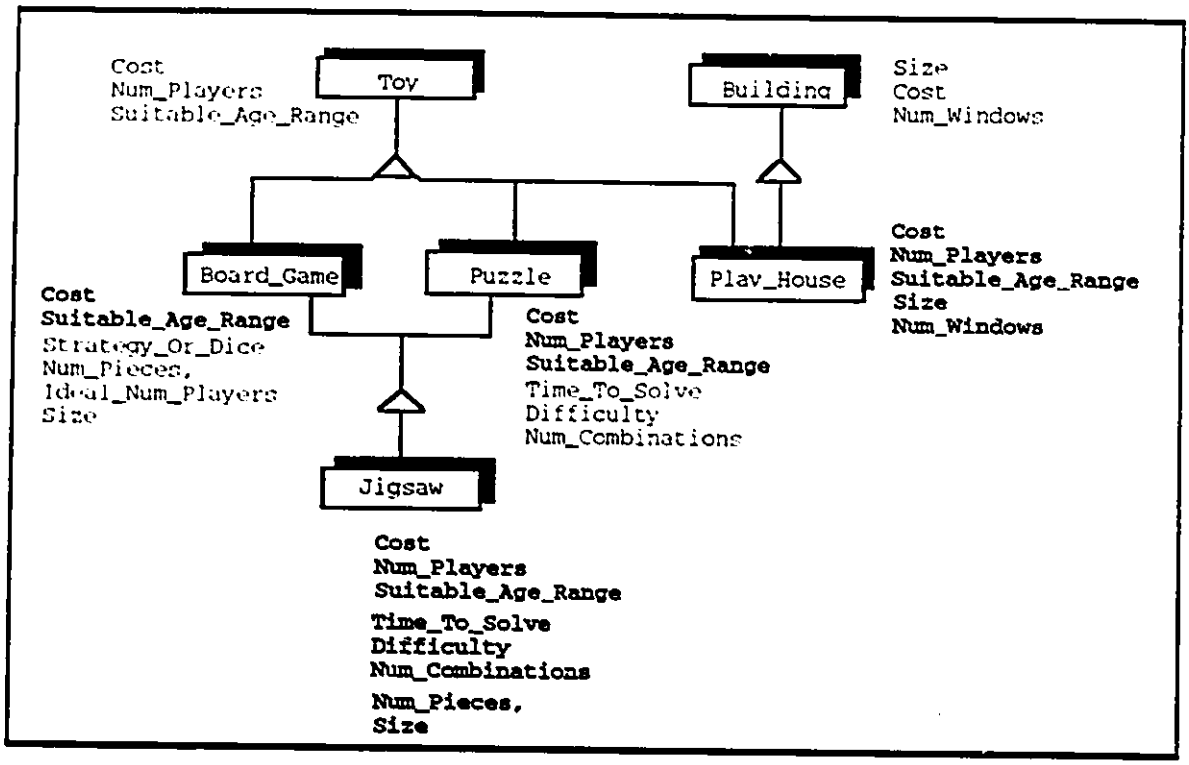


Figure A.3 An Example of Multiple Inheritance and Repeated Inheritance

1.9.4 Operations in superclasses invoking operations in subclasses

It is sometimes the case that an operation in a superclass invokes an operation that is implemented in a subclass. Fig. A.4 illustrates an example of this in Smalltalk. Class B inherits operation `organize` from A. Operation `organize` in A invokes operation `sort` with a quicksort algorithm. B has operation `sort` with a heapsort algorithm. When the statement `'aB organize'` is executed, message `organize` is sent to an instance of B. aB will process the message since it inherits operation `organize` from A. In A `organize` sends message `sort` to `self`. In our case `self` is aB, therefore when aB receives message `sort` and will perform a heapsort. .

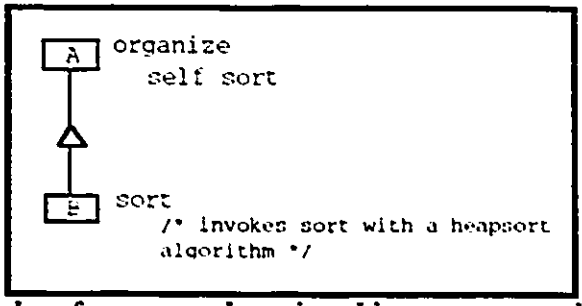


Figure A.4 Example of a superclass invoking an operation of its subclass

In terms of testing this situation is similar to testing a change to a subclass described in section 2.2.4 of Chapter 2. The operations inherited must be retested in the context of the subclass under test. In Fig. A.4, the operation `organize` must be retested in the context of B.

1.9.5 Dynamic Inheritance

Dynamic inheritance (also refer to as *dynamic semantics* or *dynamic classification*) is defined a the ability of an object to change (add, delete, or change) its type (class) dynamically.

The need for modeling entities that change over time has been discussed in [64,65,66] among others. Languages based on the Actor model of concurrency [66] and Object-Oriented database systems [65], use the notion of objects changing their type dynamically.

Different extensions to the conventional class-based object oriented model have been proposed to achieve Dynamic Inheritance.

In [64], the notion of *modes* is presented which allows an object to change its mode (logical state). Note the mode of an object is not the same thing as the concept of state in current object-oriented programming (i.e. state implemented via instance variables), it is a more conceptual notion. For example, in Graphical User Interfaces (GUIs), windows can be open, closed, represented as icons, etc., the logical states: `open`, `closed` and `iconified` would represent the modes of the object `aGUI`. Similarly, a bank account can be valid, empty, frozen or terminated; the modes of the object `aBank` account would be `valid`, `empty`, `frozen` and `terminated`. Modes are added to a class-based system by allowing the definition of multiple sets of operations within class definitions and modifying the late binding algorithm to automatically select between these sets of operations on the basis of the current mode. The objects of a class can then adopt different behaviors depending on their current logical state.

In [65] the notion *aspects* is discussed which extends an existing object with additional state and behavior while sharing the same object identity. An object may have many aspects that come and go over time. Rather than being an instance of some unique subclass defined through multiple inheritance, an object is an instance of many types by virtue of having many aspects. An object of class `Person` may extend is behavior and states to be an employee by adding an extension (i.e. aspect) `Employee` to class `Person`.

In terms of testing, dynamic inheritance can include problems identified with testing simple inheritance. In addition the tester must identify conditions that cause the change in type (class). He/she must be able to control and observe the change and handle exceptional conditions, such as a message being received when the mode or aspect of the object does not handle the specific message. Every mode or aspect should be tested for unexpected messages.

Appendix B Object-Oriented Notations

The purpose of this appendix is to provide a brief overview of certain Object-Oriented notations mentioned in section 4.3.2 of Chapter 4.

1.1 The Use Case model notation

The Use case Model [18] describes a specific way of using the system by using some part of the functionality of the system. "Each Use Case constitutes a complete course of events initiated by an actor and it specifies the interaction that takes place between an actor and the system" [18]. The Use Case model is used to model requirements

Use Cases can have a *basic course* (gives the best understanding of the system) and several alternative courses (variants of the basic course, for example errors). Uses Cases can also be related to each other via the *extend* relationship. It describes how one Use Case description may be inserted into, and thus extend, another Use case description. Uses Cases can also be refined. This is mainly done by identifying similar parts of the Uses Cases and extracting these similar parts. In this way, the similar parts have to be described only once instead of in all Use Cases showing the similar behavior. Any changes to this part will thus automatically affect all Uses Cases that share this part. The extracted Uses Cases are called *abstract* Use Cases since they will not be instantiated on their own, but are only meaningful to describe common parts between other Use Cases. The Use Cases that really will be instantiated are called *concrete* Use Cases. The concrete Uses Cases *use* the abstract Use cases.

Fig. B.1 illustrates an example of a Use Case model. The Use Case model is usually accompanied with a text description for each Use Case as describe below.

> *Change Item* is used by the *Operator* when he wants to print out information about the returned deposit item of the day. The system will print out how many of each deposit item type have been received this day, as well as the overall total for the day. The total number will reset to zero to start a new daily report.

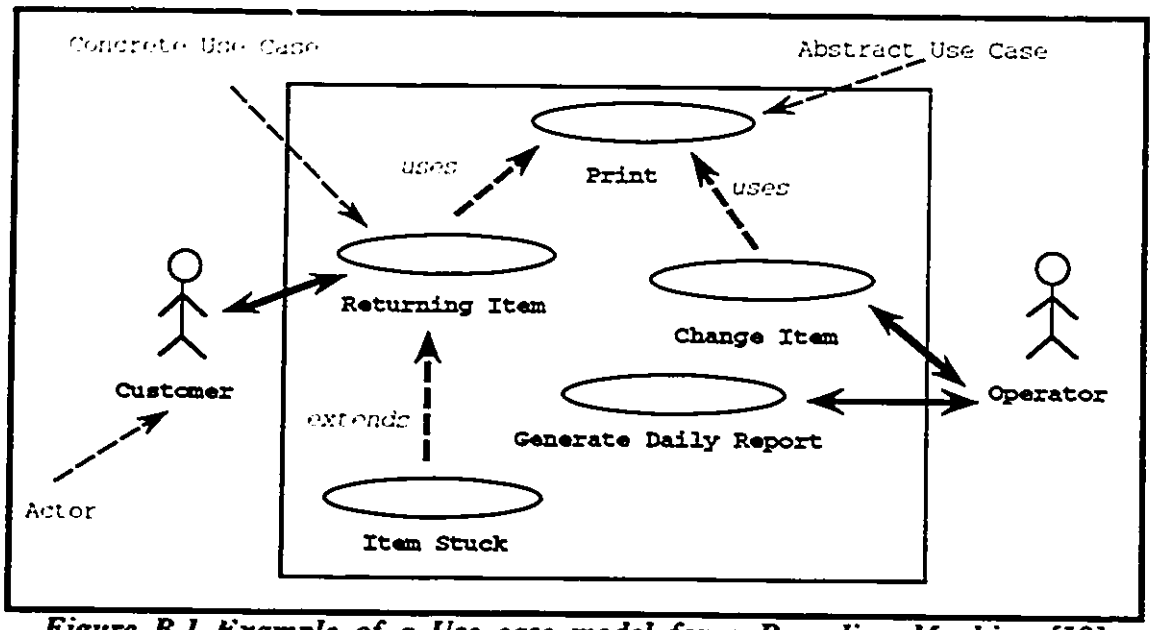


Figure B.1 Example of a Use case model for a Recycling Machine [18]

1.2 The Domain Object model notation

The Domain Object model [18], models a phenomenon in real life that the system needs to be aware of. It aims at understanding the concepts of the problem domain by shows the relations (class and instance associations) between domain objects. Fig. B.2.1 illustrates class associations and Fig. B.2.2 illustrates instance associations. Fig. B.2.3 gives an example of a Domain Object model.

Inheritance *inherits* ⟶	The descendant has all the properties of the ancestor. The arrow is drawn from the descendant and points to the ancestor since the descendant knows of its ancestor(s), but the ancestor(s) does not know of its descendants.
Extends *extends* ⟶	One class extends another class. The arrow point to the class that will be extended.

Figure B.2.1 Class Associations

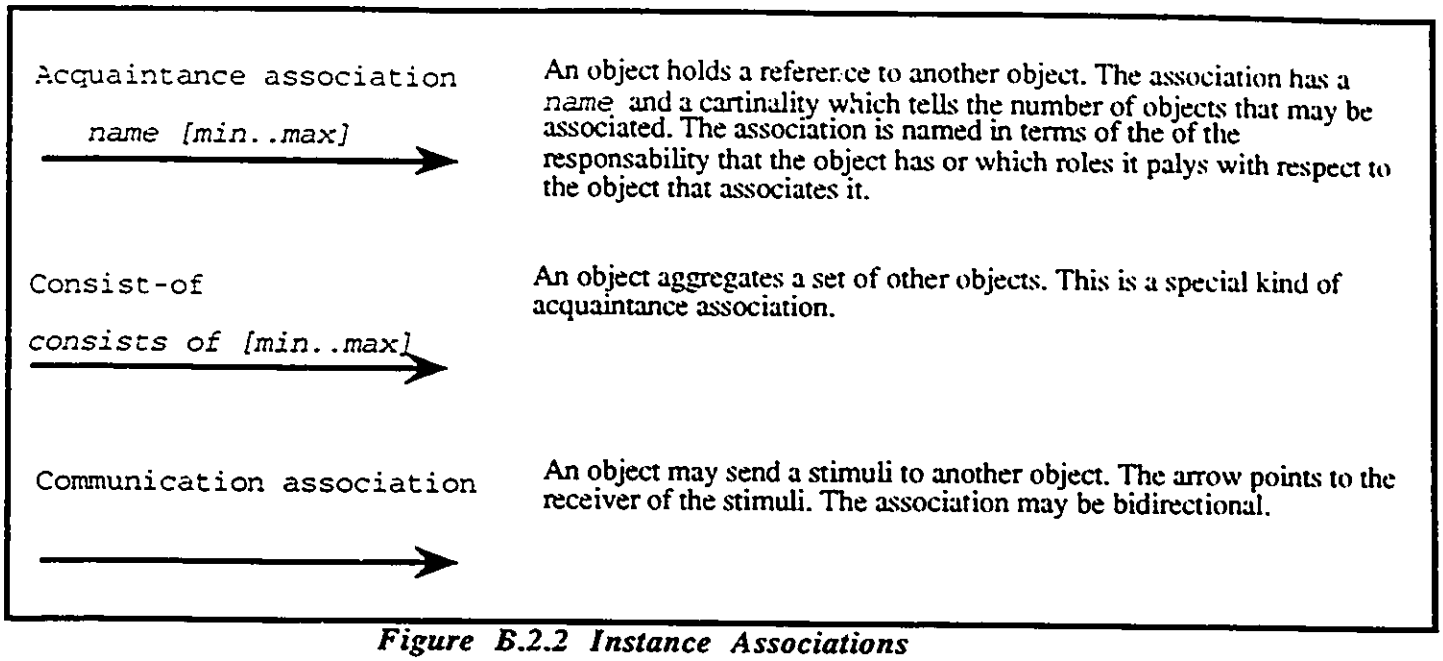


Figure B.2.2 Instance Associations

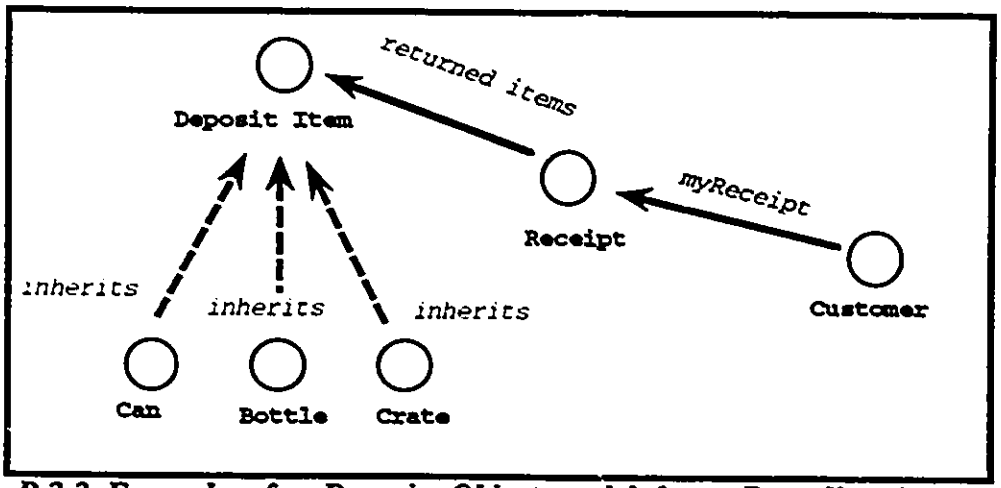


Figure B.2.3 Example of a Domain Object model for a Recycling Machine [18]

1.3 The Analysis Object model notation

The Analysis Object model [18], shows the relations (class and instance associations) between analysis objects (Entity, Interface and Control objects). It also shows the distribution of objects into subsystems.

Class association are described as depicted in Fig. B.2.1. Instance associations are described as depicted in Fig. B.2.2. Other specific constructs of the Analysis Object model are described in Fig. B.3.1. Fig. B.3.2 illustrates an example of an Analysis Object model.

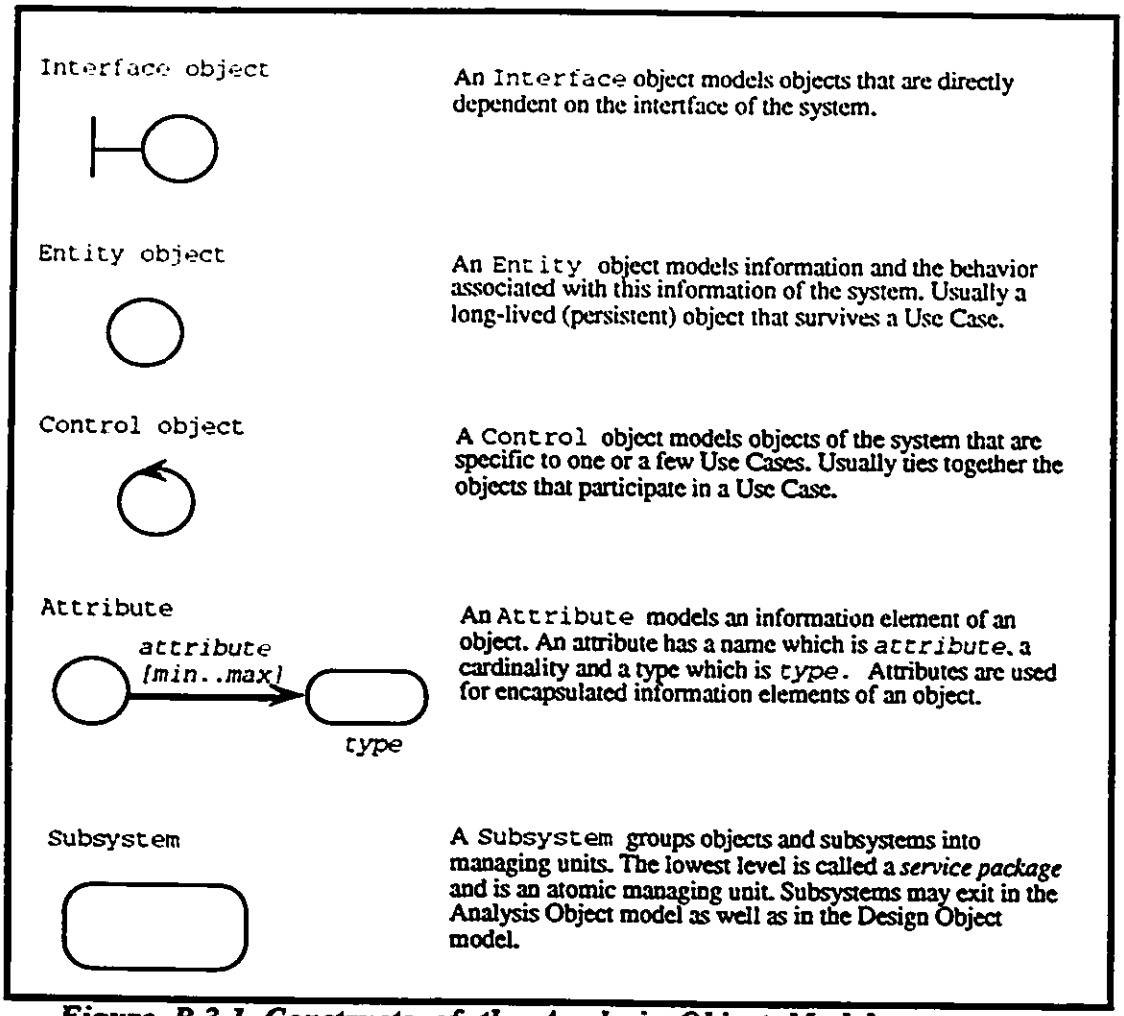


Figure B.3.1 Constructs of the Analysis Object Model

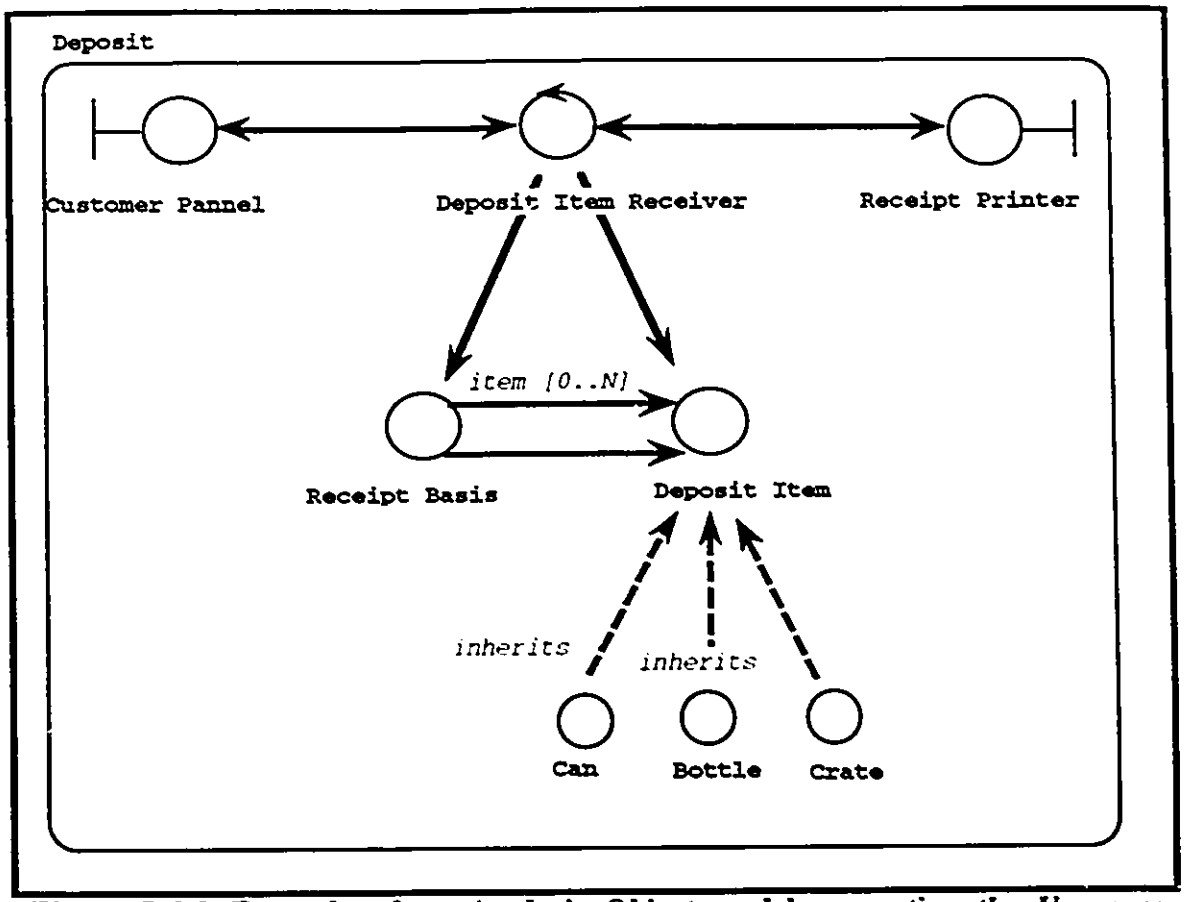


Figure B.3.2 Example of an Analysis Object model supporting the Use case `Returning Item [18]` *and describing the* `Deposit` *subsystem*

1.4 The Design Object model notation

The Design Object model [18], defines the architecture of the system in terms of Blocks (analysis objects) and their relations (class and instance associations).

Class association are described as depicted in Fig. B.2.1. Instance associations are described as depicted in Fig. B.2.2. Other specific constructs of the Design Object model are described in Fig. B.4.1. Fig. B.4.2 illustrates an example of a Design Object model.

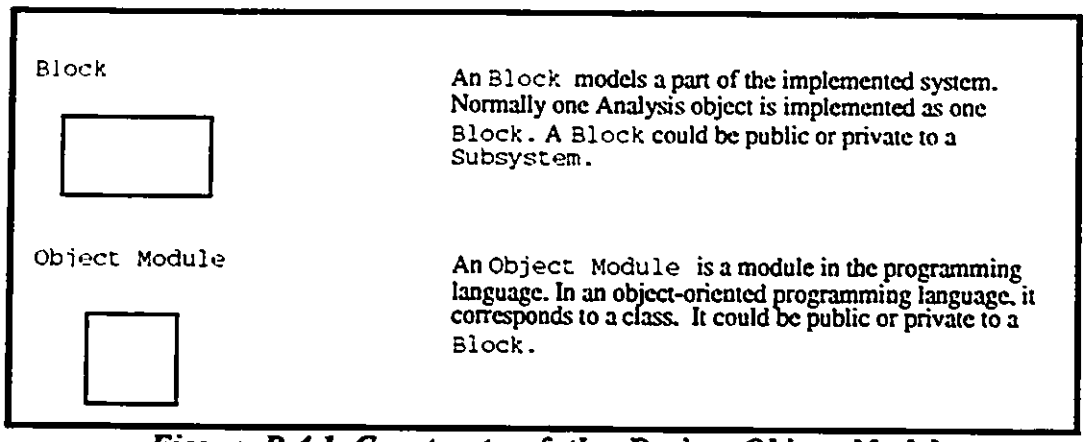


Figure B.4.1 Constructs of the Design Object Model

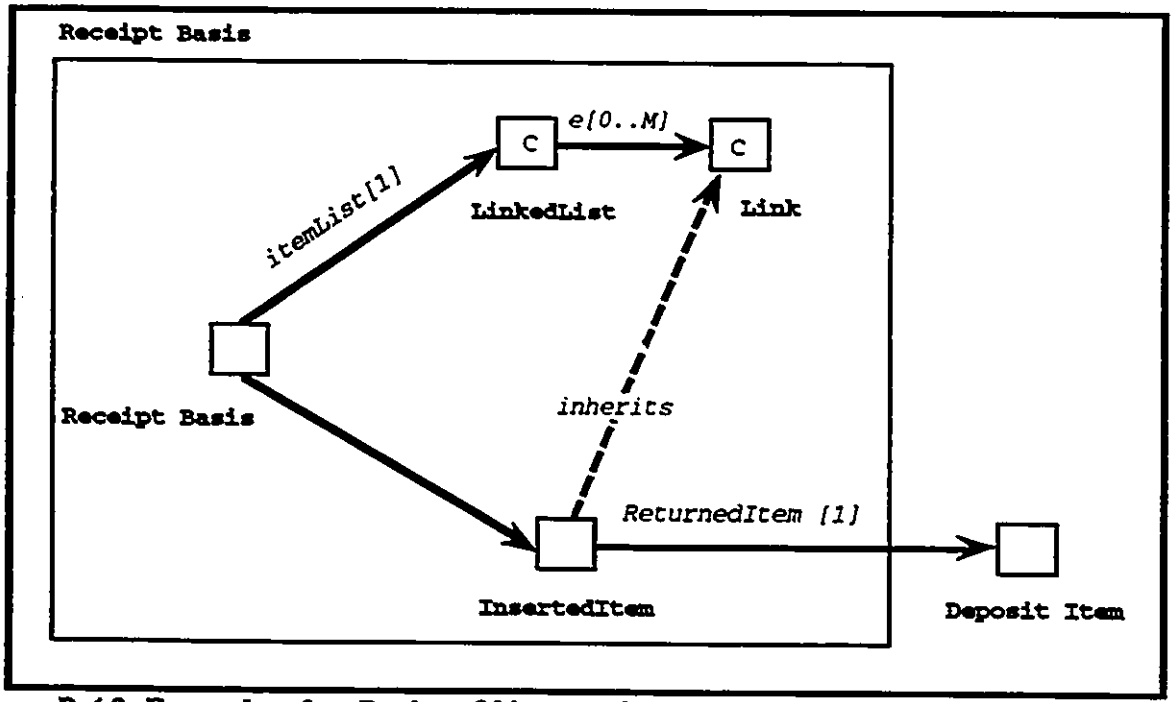


Figure B.4.2 Example of a Design Object model for the Receipt Basis block [18]

1.5 The Interaction Diagram notation

An Interaction Diagram [18], models the communication between objects. It shows how objects (blocks) communicate in terms of stimulus (operations) sent and received.

Each participating objects is represented by a column. All behavior of an object is attached to the column representing the object. An Interaction Diagram also contains a column for the surrounding world; the column is referred to as the *system border* and represents the interface with everything outside the objects in the diagram, such as external actors and consequently it can correspond to different interfaces outside the system. The time axis in an Interaction Diagram is viewed as going down. At the left edge, to the left of the *system border*, sequences are described. A sequence corresponds to text (e.g. pseudo-code) describing a specific Use Case in terms of *operations*. Every *operation* belongs to an object and therefore it is drawn as a rectangle in the column for that object. Finally, Interaction Diagrams are controlled by *events*. A new *event* gives rise to a new *operation*. An event or stimulus is drawn as a horizontal arrow that starts in the column corresponding to the sending object and ends in the column corresponding to the receiving object.

Fig. B.5 illustrates an example of an Interaction Diagram.

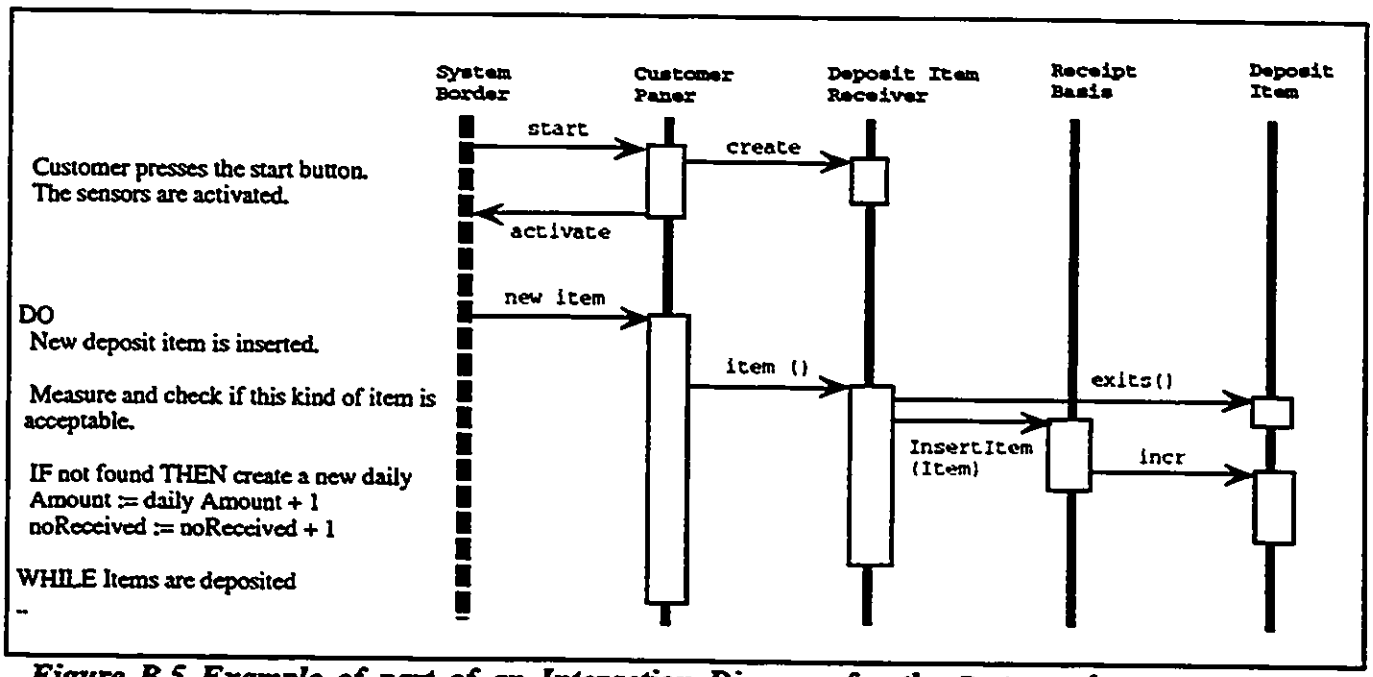


Figure B.5 Example of part of an Interaction Diagram for the Returning Item Use Case [18]

1.6 The Object Interaction Diagram and Data Flow Diagram notation

An Object Interaction Diagram [39], describes the message flow between objects. It corresponds to an Instance Diagram with message flow between instances. Messages correspond to operations invoked by an object and data flows as message arguments and return values. An Object Interaction Diagram also shows threads of control via a numbering scheme (i.e. message contain numbers reflecting their order).

A Data Flow Diagram [14,39], describes the effects of operations of objects. It shows the functional relationships of the values computed by a system, including input values, output values and internal data stores. It corresponds to a graph showing the flow of data values from their sources in objects through processes that transform them to their destinations in other objects. A Data Flow Diagram contains *processes* (i.e. operation of objects) that transform data, *data flows* that move data, *objects* that consume, produce and store data and *data values*.

A Data Flow Diagram is extracted from the Object Interaction Diagram. Therefore, it does show control information such as which paths are executed and in what order as well as the organization of values into objects.

Fig. B.6.1. illustrates an example of an Object Interaction Diagram. Fig. B.6.2 illustrates an example of a Data Flow Diagram extracted from a Object Interaction Diagram.

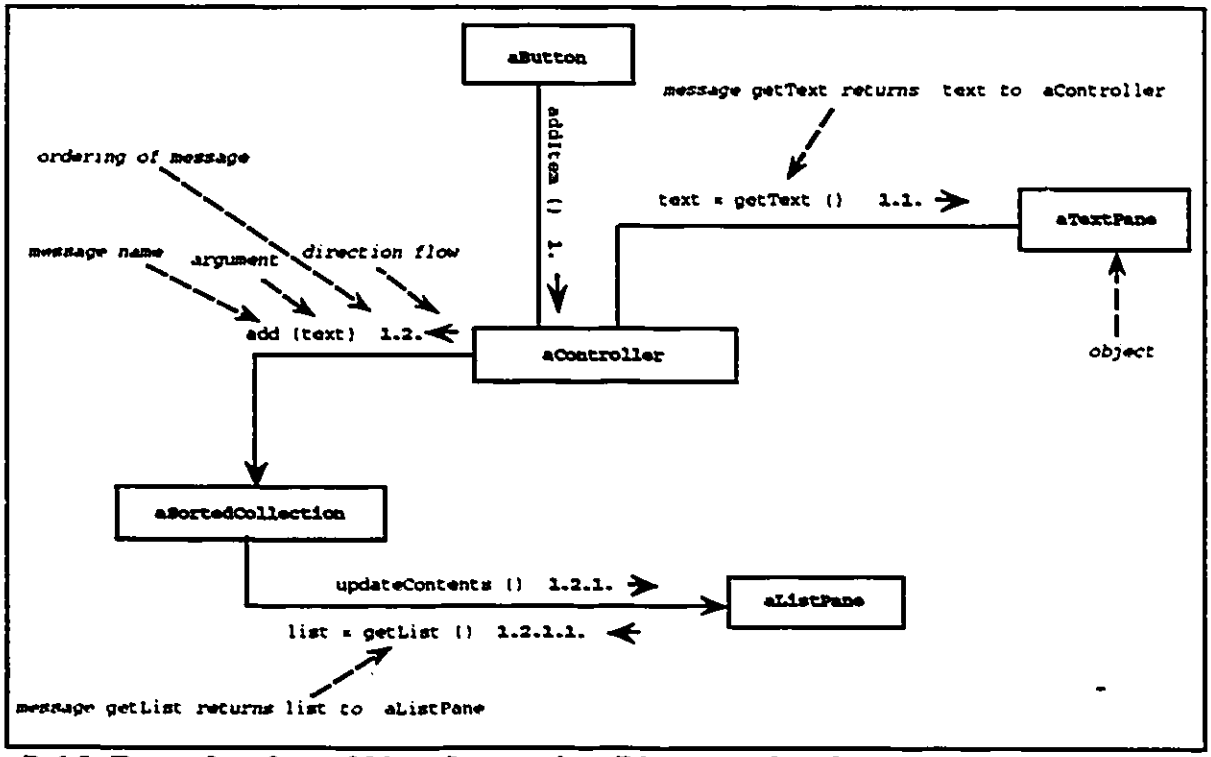


Figure B.6.1 Example of an Object Interaction Diagram for the add button of an editor [39]

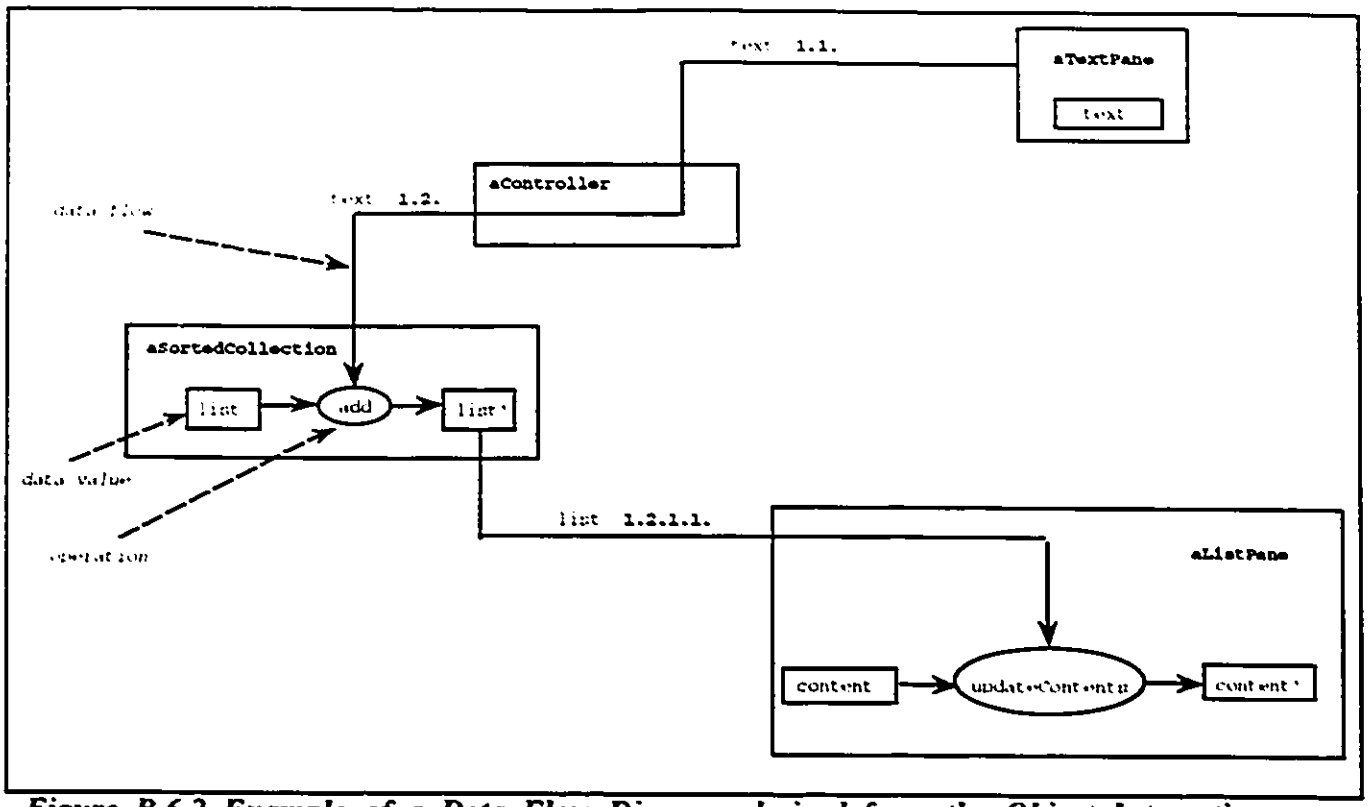


Figure B.6.2 Example of a Data Flow Diagram derived from the Object Interaction Diagram in Fig. B.6.1

1.7 The Domain Chart notation

A Domain Chart [24] depicts the different domains of the application to be developed as well as usage among them. "A Domain can be thought of as a separate world inhabited by its own conceptual entities, or objects. Hence, in an Automated Railroad Management System, a Railroad Operations Domain is concerned with trains, tracks and the like, while a User Interface Domain is involved with windows, displays, and icons" [24].

Each domain is represented as an oval. A connection between two domains indicates that the higher domain will make use of facilities provided by the lower domain in the implemented system. This is referred to as a bridge.

Fig. B.7 illustrates an example of a Domain Chart.

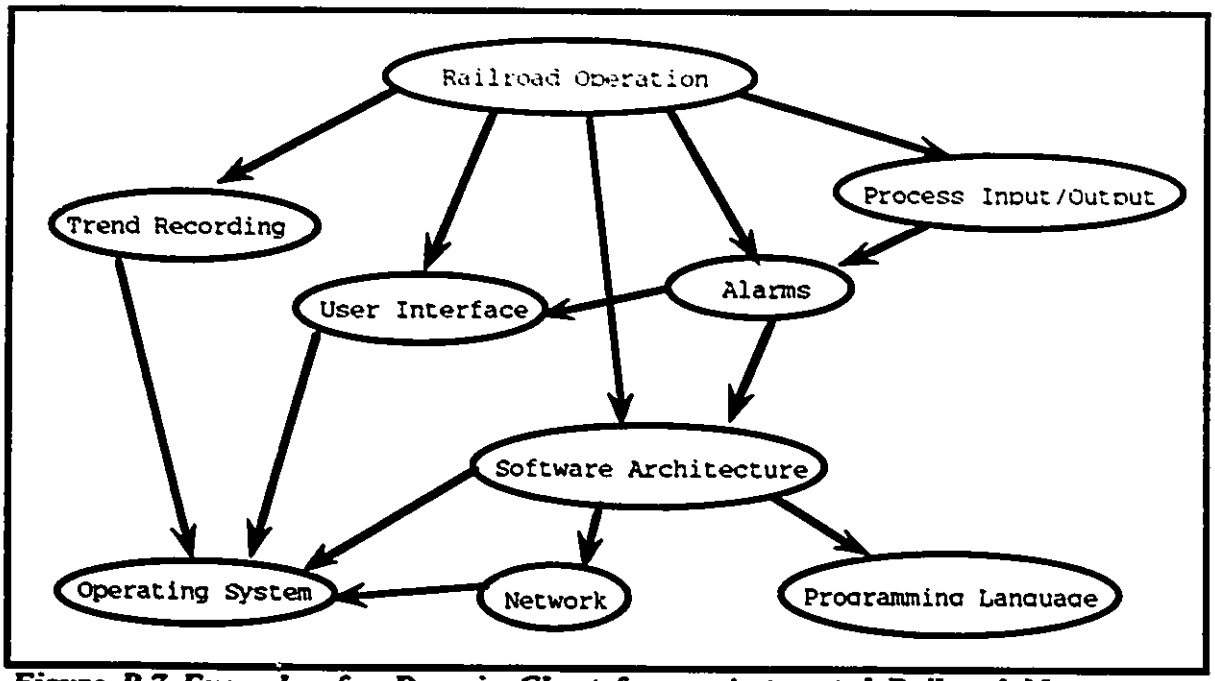


Figure B.7 Example of a Domain Chart for an Automated Railroad Management System

1.8 The Information Model notation

An Information Model [24] shows the information structure (i.e. objects, classes, their attribute descriptions and relationships to other objects). Fig. B.8 shows the major constructs of the model (i.e. *objects*, *attributes* and *relationships*) as well as an example.

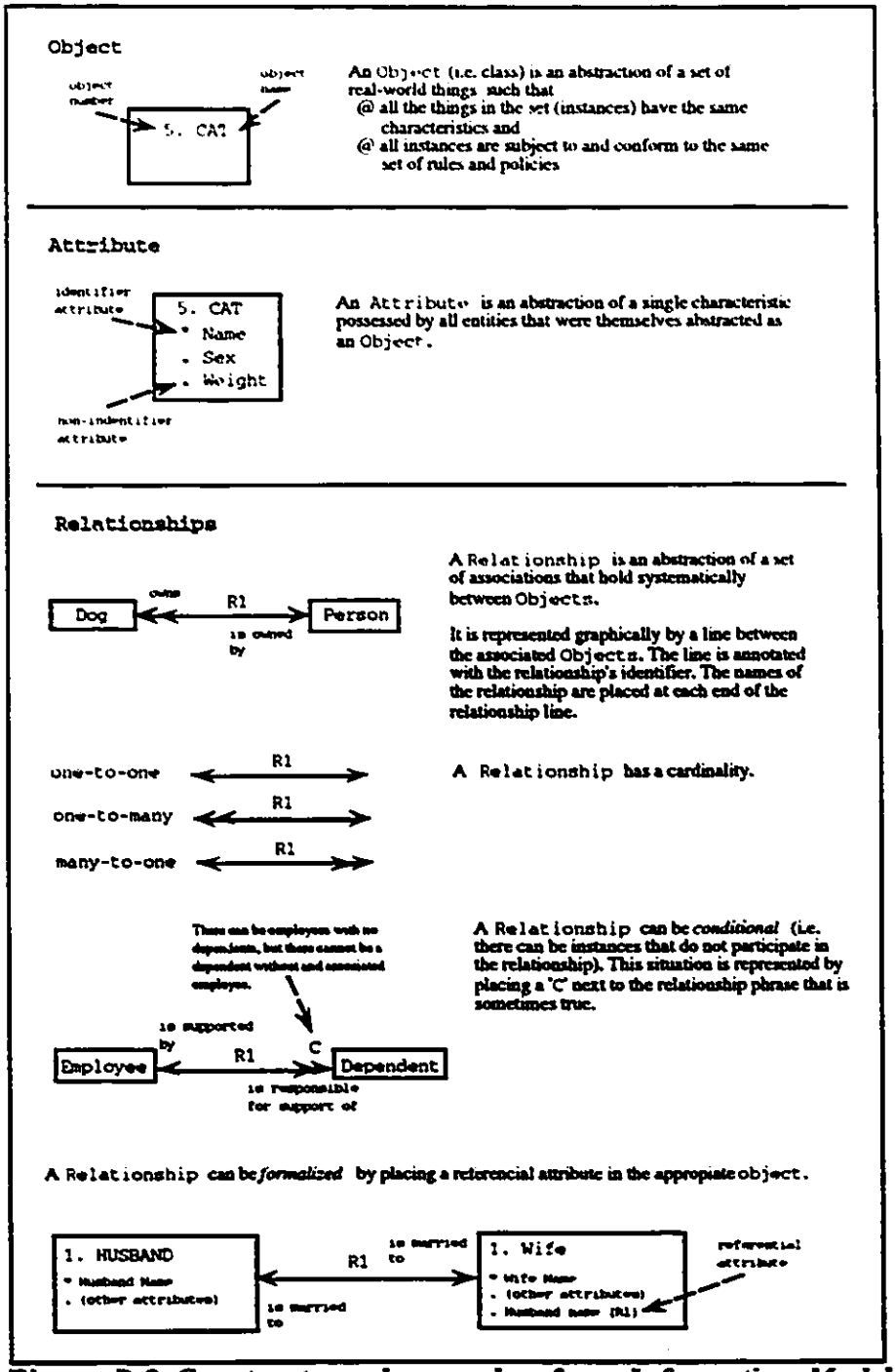


Figure B.8 Constructs and example of an Information Model

1.9 The Subsystem Relationship Model notation

An Subsystem Relationship Model [24], shows the relationships between Subsystems in Domains. A Subsystem is depicted graphically by a box and it represents the entire Information Model for that Subsystem. Relationships between Subsystems are represented by a line connecting the two Subsystems. The line is labeled with the identifier of the inter-subsystem relationships.

Fig. B.9 shows an example of a Subsystem Relationship Model.

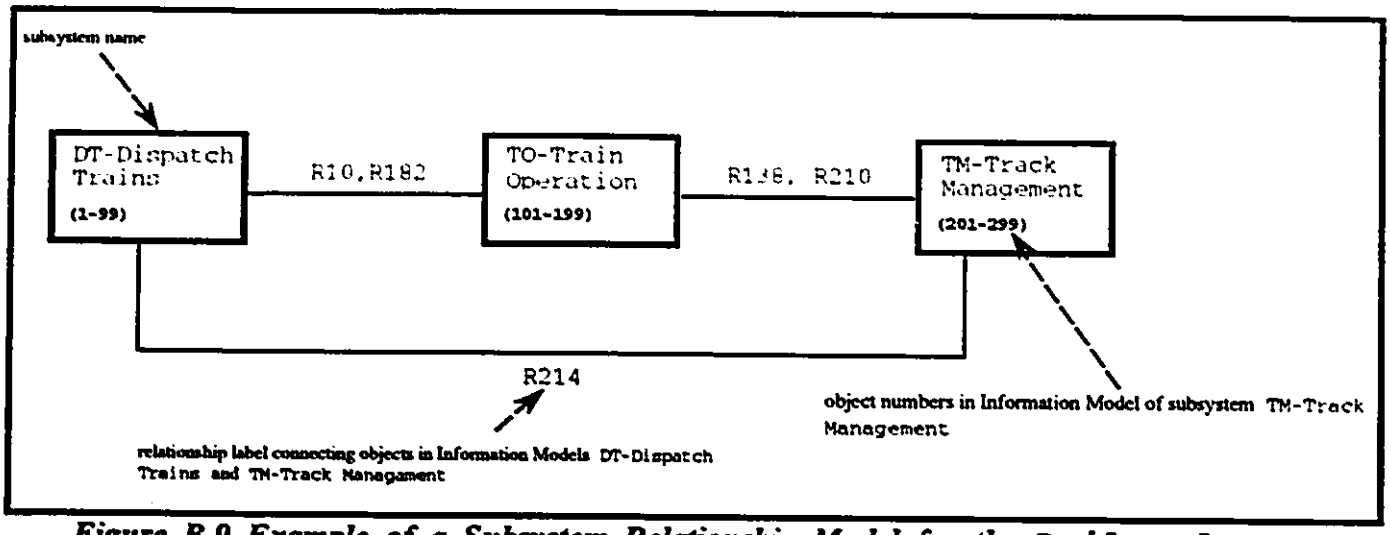


Figure B.9 Example of a Subsystem Relationship Model for the `Railroad Operation` domain [24]

1.10 The Object Communication Model notation

The Object Communication Model [14], shows the asynchronous communication between objects (i.e. classes) and external entities. "When a State Model generates an event, the target State Model receives the event some time after the action in which the event was generated is complete. This communication is termed asynchronous" [24]

Every flattened oval represents an object (the name of the oval is the same as the name of the State Model for that object). Each external entity (e.g. operators, physical devices, and objects of other subsystems) that can generate or receive an event is depicted by a box, know as a *terminator*. An event that is generated by one State Model or external entity and received by another is represented by an arrow from the generating component to the receiver. The arrow is annotated with the event and (optionally) event data.

Fig. B.10 shows an example of an Object Communication Model.

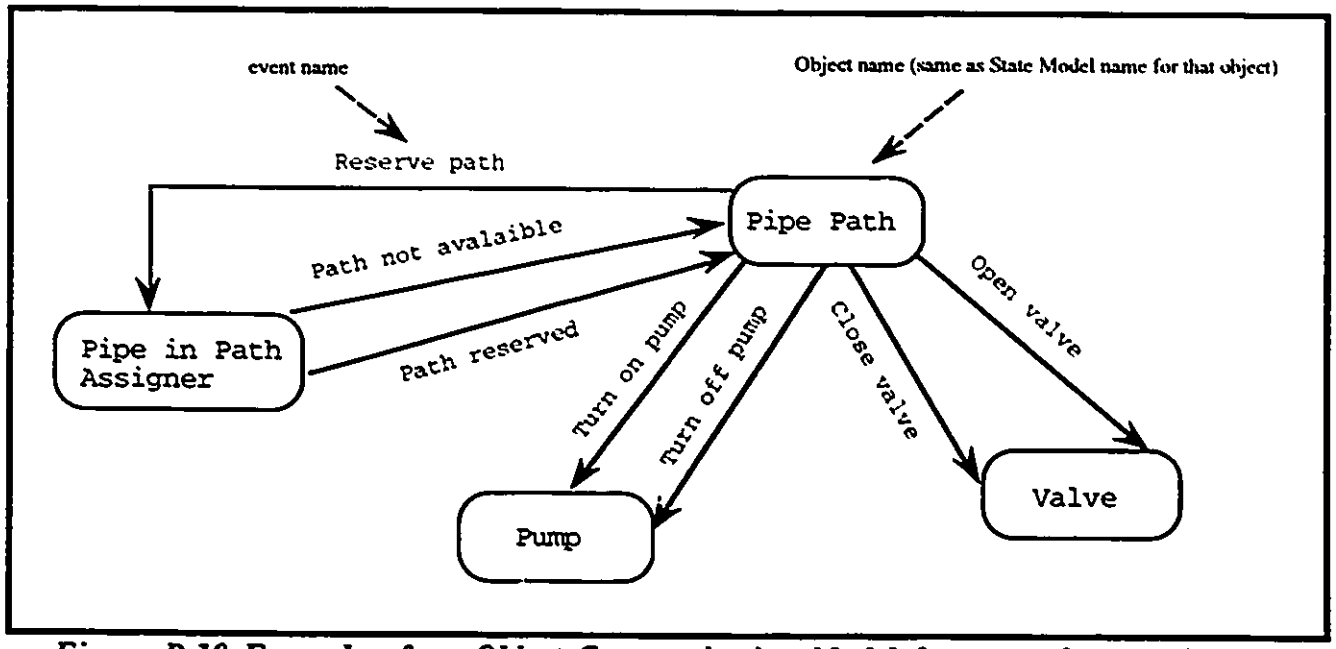


Figure B.10 Example of an Object Communication Model for part of a Juice Plant [24]

1.11 The Object Access Model notation

The Object Access Model [14], shows the synchronous communication between objects (i.e. classes). "When a State Model accesses the instance data of another object through an *accessor process* [38], the data access takes place during the time that the action is running. This kind of communication is said to be synchronous." [24].

Every flattened oval represents an object's data store and its State Model and it is named with the object's name.

"If a State Model (call it State Model A) makes use of an accessor assigned to another object (object B), an arrow is drawn from object A to object B. The arrow is labeled with the process identifier of the accessor.

Fig. B.11 shows an example of an Object Access Model.

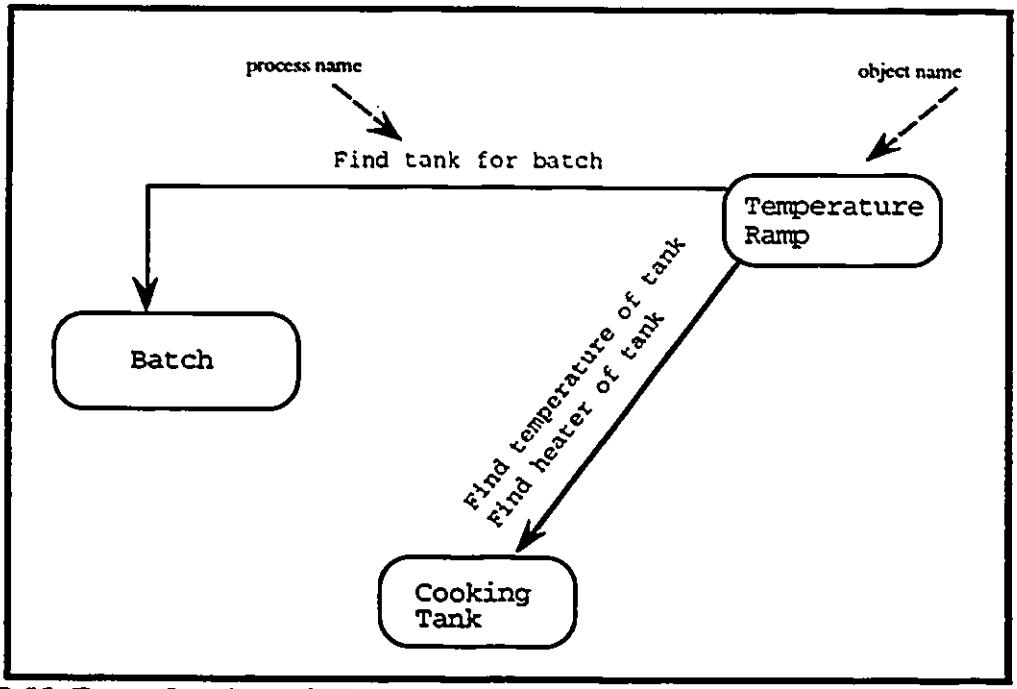


Figure B.11 Example of an Object Access Model for part of a Juice Plant [24]

[38] An *accessor process* is a process whose only purpose is to access data in a single object data store (data that describes all instances of an object class).The accessor may be one of the following:

Create Accessor: creates a new instance of an object class.
Read Accessor: reads attributes of a single object class.
Write Accessor: updates attributes of a single object class.
Delete Accessor: deletes an instance of an object class.

The Thread of Control Chart notation [14], shows the sequence of actions and events that occurs in response to the arrival of an event when the system is in a particular state. If an action along the thread of control generates more than one event, the thread of control splits so that two or more legs of the same thread of control are active at the same time.

The Thread of Control Chart notation glues the Object Communication Model notation and the State Transition Diagram notation. Graphically, it is represented as a succession of events and states occupied by instances that participate in a particular thread of control. Each instance appears separately as the string of states it occupies as the thread progresses. The states occupied by a single instance are connected by arrows, each labeled with the event that causes the transition to the next state. If an instance generates an event to another State Transition Diagram and that event causes a transition, an arrow is drawn from the state that generated the event to the transition of the receiving instance. The chart is laid out along a relative time axis, with the states placed on the chart in the order (top to bottom) in which they are occupied.

The Thread of Control Chart notation can be used to analyzed the time it takes the system to respond to an event. This is done in the following way. The time an instance occupies a certain state is made up of *action time* (time required to execute an action) and *dwell time* (time the instance remains in the state after the completion of the action).

Dwell time may be entirely determined by the instance and the state (e.g. in Fig. B.12, it took 10 minutes to serve `Customer9` and 6 minutes to serve `Customer10`. Alternatively, *dwell time* may be determined by interactions between State Transition Diagrams (e.g. in Fig. B.12, the length of time that `Customer10` needs to remain in the `Waiting for clerk` state is dependent on the availability of the `Clerk`; we will show that this time is 3.21 minutes).

To portray the timing of the thread of control on the chart.

* each state is annotated with its *action time*
* if the *dwell time* of a state is determined by the instance and state alone, the transition out of this state is annotated with the *dwell time*
* if the *dwell time* of a state is determined by interactions between State Transition Diagrams, a capacitor symbol is placed on the transition out of that state.

Fig. B.12 illustrates an example of a Thread of Control Chart.

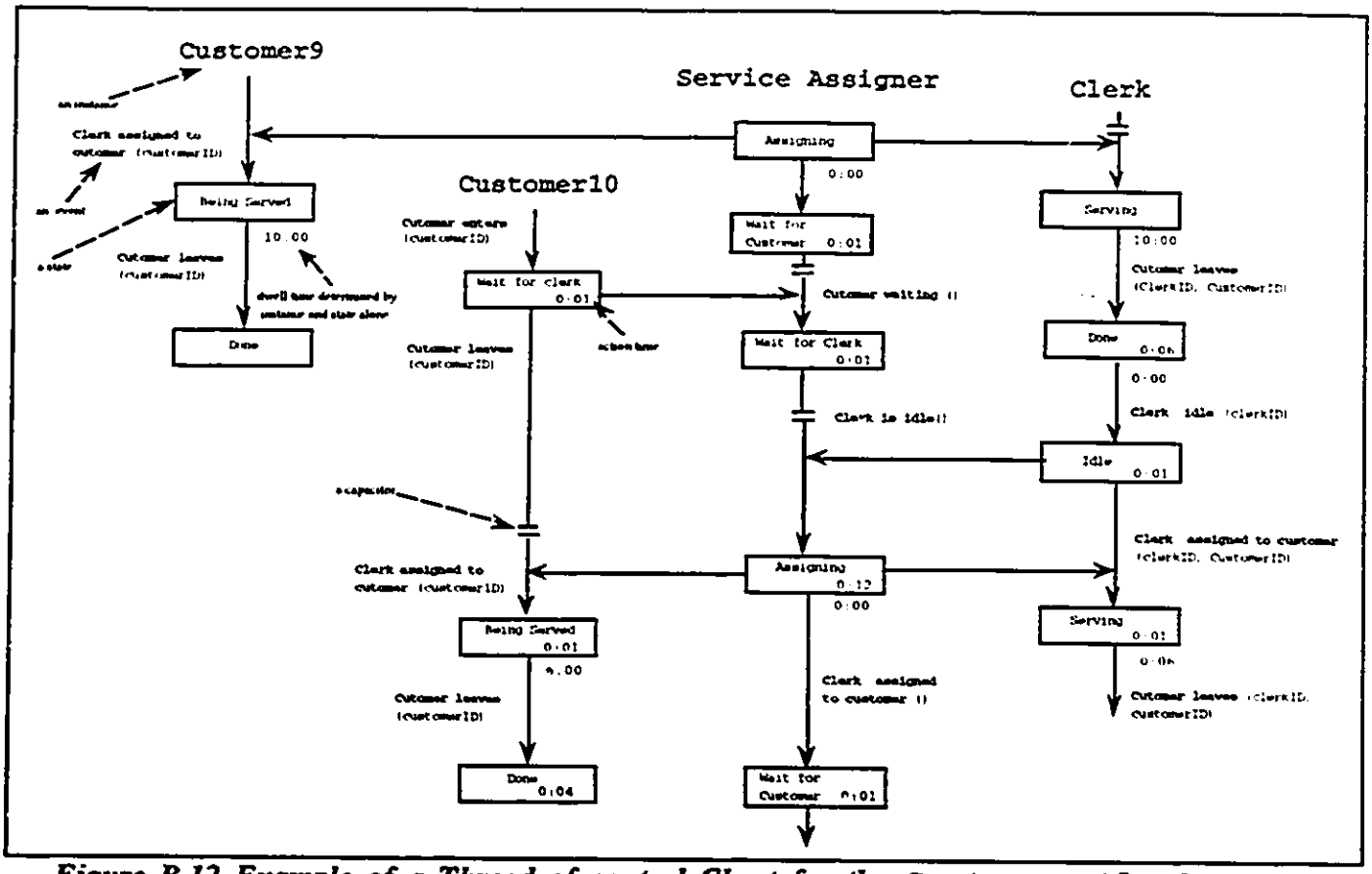


Figure B.12 Example of a Thread of control Chart for the `Customer-Clerk` problem [24]

Lets calculate the amount of time that `Customer10` has to wait before being assigned a clerk (assume `Customer10` arrived when the clerk had been serving `Customer9` for 7 minutes)

INSTANCE	ACTION	TIME COMPONENT	DURATION
Customer10	Wait for clerk	action time	0:01
Clerk	Serving	dwell time	3:00
Clerk	Done	action time	0:06
Clerk	Done	dwell time	0:00
Clerk	Idle	action time	0:01
Assigner	Wait for clerk	action time	0:01
Assigner	Assigning	action time	0:12
		TOTAL	**3:21**

Lets calculate the amount of time that `Customer10` is being served by the clerk

INSTANCE	ACTION	TIME COMPONENT	DURATION
Customer10	Being Served	action time	0:01
Customer10	Being Served	dwell time	6:00
Customer10	Done	action time	0:04
		TOTAL	**6:05**

1.13 The Action Data Flow Diagram notation

The Action Data Flow Diagram [14], describes the units of processing within an action and the intercommunication between them. The main constructs of the notation are *processes*, *data flows*, *persistent data*, *received events* and *generated events*.

A *process* corresponds to a separate unit of computation and is represented by a circle. It usually requires input data to carry out a function and produces output data as a result.

If a process requires input data, that data is shown being supplied to the process as a *data flow* directed to the process via a directed line. If a process produces data as output, the data is shown as a *data flow* directed away from the process via a directed line.

Persistent data corresponds to data that continues to exist after an action is complete. It is represented as a data store. If a data store provides data to (or receives data from) a process, the data store and the process are connected by a data flow.

A *received event* corresponds to an event received by a State Model that causes an action to be initiated. This is depicted as one or more *event data flows* (i.e. a data flow pointing into a process from nowhere). The event data flow is labeled with the names of the arguments that are carried by the event and are required by the process.

A *generated event* is event generated by a process. It is depicted as data flows directed away from the process. The data flow is labeled with the event name and arguments.

The Action Data Flow Diagram is closely coupled with the State Transition Diagram. Fig. B.13 shows an example of this.

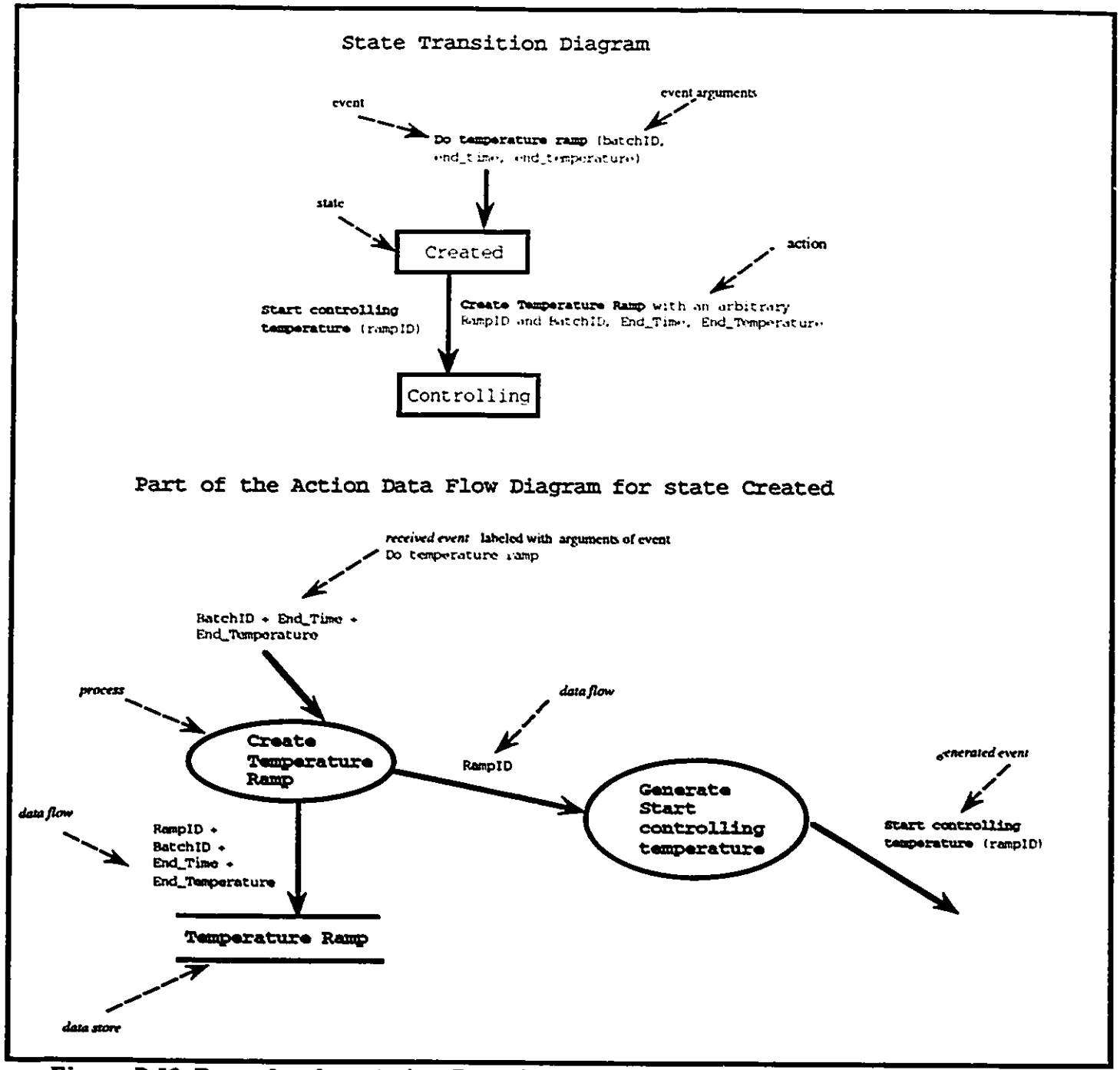


Figure B.13 Example of an Action Data Flow Diagram for state Created of Temperature Ramp [24]

1.14 The Class Structure Chart notation

The Class Structure Chart [14] notation describes the internal structure of a class as well as the flow of data and control within that class. It is usually used to show the internal structure of the code of the operations of the class. A separate chart is produced for each class.

It is based on the concept of a module, operation of a class, (i.e. a piece of code that is invoked and, when complete, returns control to the caller). Each module is represented by rectangle. A module may invoke one or more other modules. An invocation is depicted by an arrow from the calling module to the called module. Any data passed to or returned by a called module is represented by a wok[39]. An arrow attached to the work tell the direction of flow of the data. If input data is modified and then returned to the caller, the wok carrying that data will have two arrows attached (i.e. one showing the input direction, and one the output direction). If the input (or output) is produced only under certain circumstances, the wok is drawn with a dotted border.

Modules can be *primary* or *foreign*. A *primary module* represent the unit of code that receives control when an operation is invoked. They appear near the top of the Class Structure Chart as being invoked but with no external caller. A *foreign module* is a module that does not belong to the class being depicted. It is represented by the apron symbol.

A Class Structure Chart also shows, instance data of a class. polymorphic invocations and exceptions. Fig. B.14 illustrates an example of a Class Structure Chart.

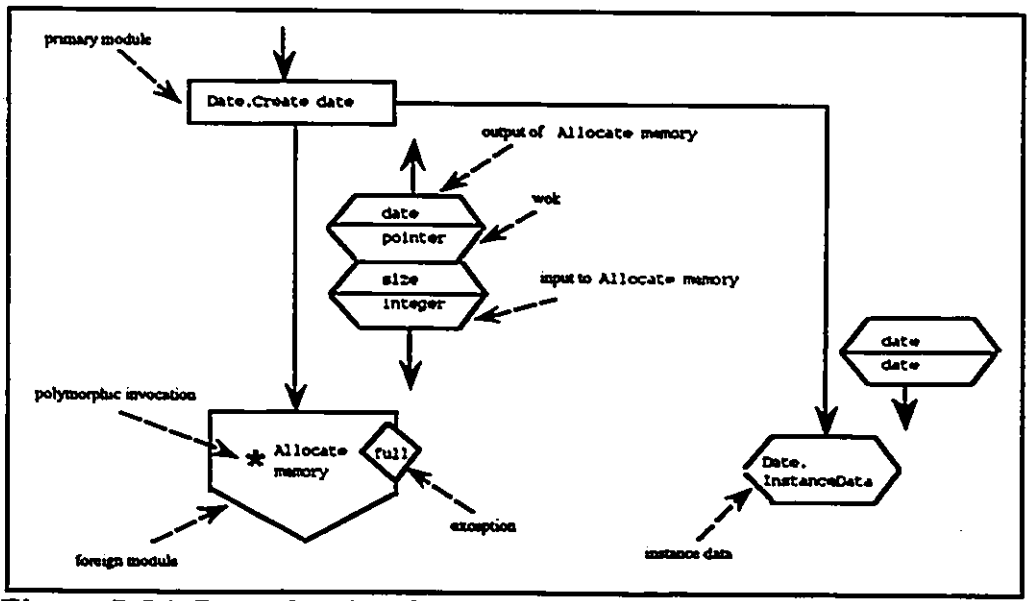


Figure B.14 Example of a Class Structure Chart for class `Date` [24]

[39] Small divided hexagon in a Class Diagram denoting logical component (i.e. data of a class).

1.15 The Dependency Diagram and Inheritance Diagram notation

The Inheritance Diagram notation [14], describes the Inheritance relationships between classes of a single program, library or environment. Each class is represented by a class box with all its internal details as in the Class Diagrams notation. Inheritance relationships are represented by a cross. Fig. B.15 shows an example of an Inheritance Diagram.

The Dependency Diagram notation [14] describes the relationships between classes other then Inheritance. There are two types of relationships (dependencies) between classes: *client-server* and *friends*. When a module of one class (call it Class A) invokes an operation of another class (class B), a *client-server* relationship is said to exist between the classes. The invoking class (Class A) is considered to be the *client*, while the invoked class is considered to be the *server*. *Friends* relationships are intended to reduce complexity and increase performance at the expense of encapsulation. When a module in class A either (1) invokes an internal operation of class B or (2) makes direct access of the data of class B, class A is said to be friend of class B. A Class is represented by a box with the name of the class. Friends relationships are drawn with a double directed arrow from the invoker to the object being invoked. Client-server relationships are drawn as a single directed line from the client to the server.

Fig. B.15 shows an example of an Dependency Diagram.

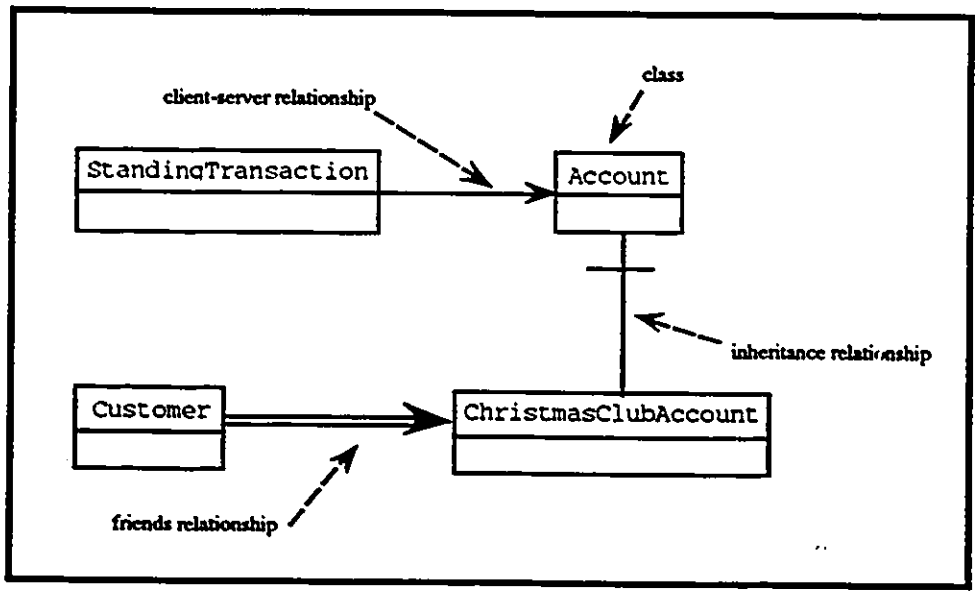


Figure B.15 Example of a Dependency Diagram and Inheritance Diagram [24]

1.16 The OAN Static notation

The OAN (Object Analysis Notation) static notation [15], capture the internal structure (e.g. attributes) and external structure (e.g. relationships) of objects. Fig. B.16.1 and B.16.2 illustrates the basic constructs of the notation as well as examples.

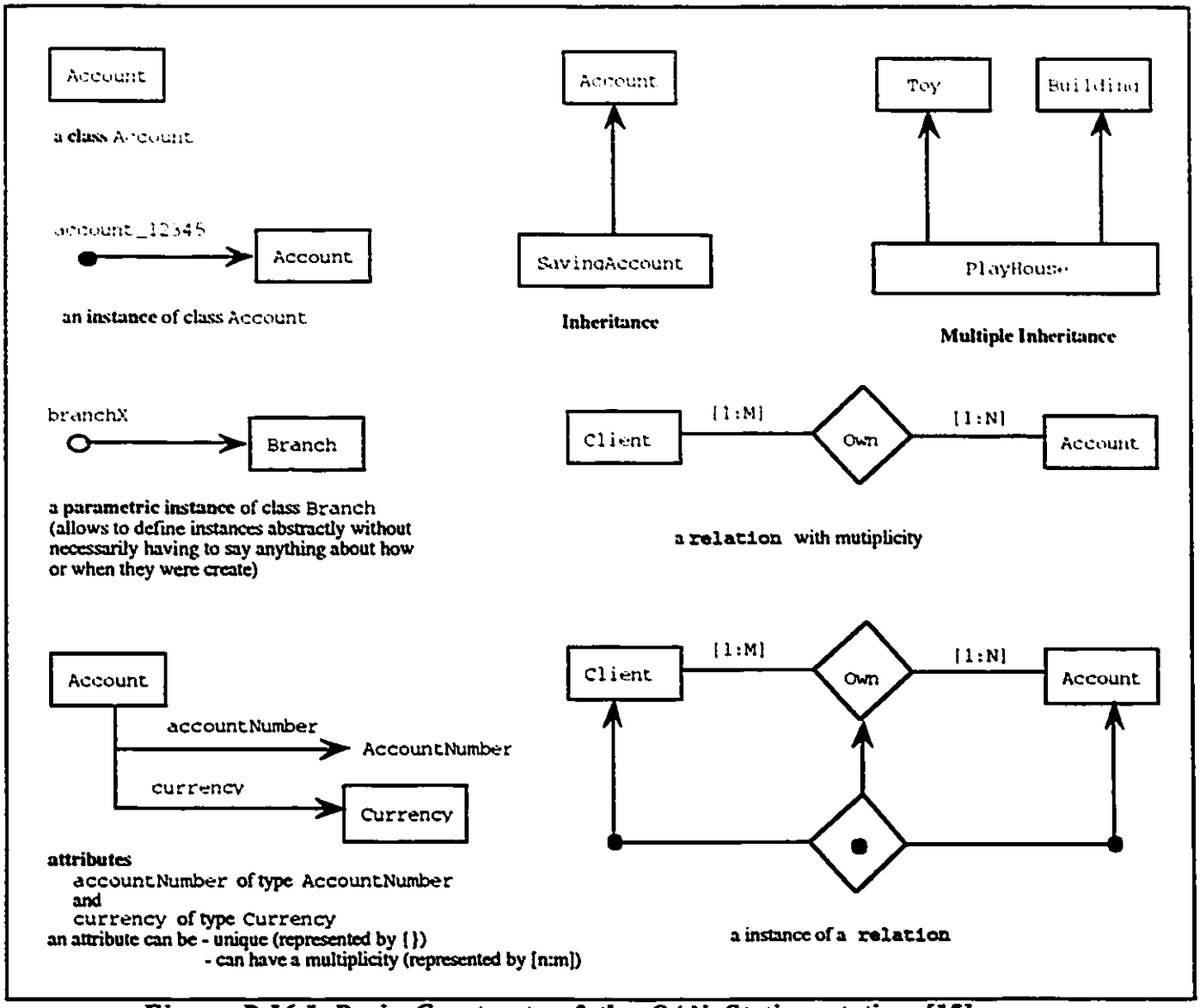


Figure B.16.1 Basic Constructs of the OAN Static notation [15]

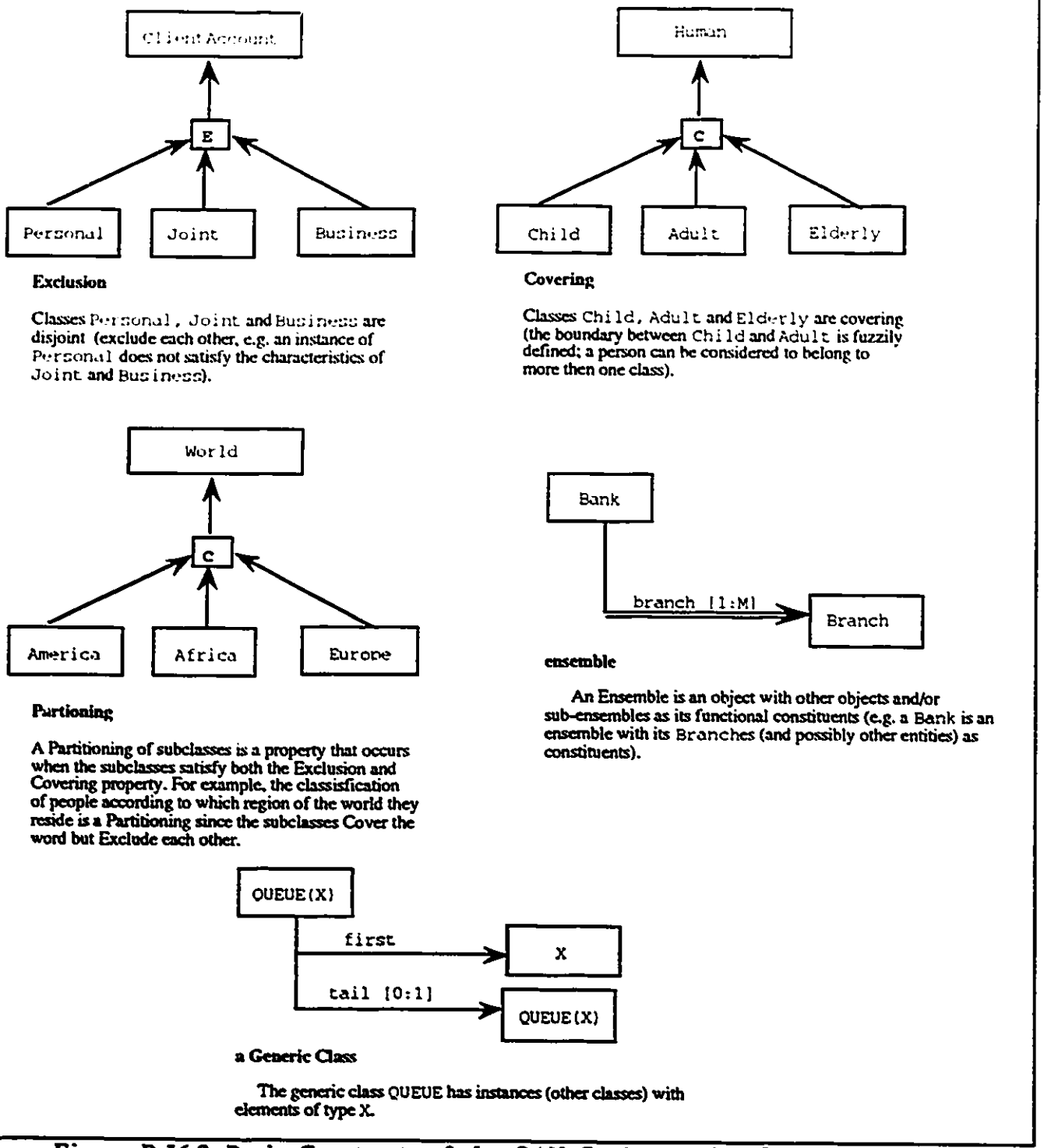


Figure B.16.2 Basic Constructs of the OAN Static notation [15] cont.

1.17 The OAN Dynamic notation

The OAN (Object Analysis Notation) Dynamic notation [15], describes the internal behavior (states, events and transitions) of objects. Fig. B.17 illustrates the basic constructs of the notation as well as examples.

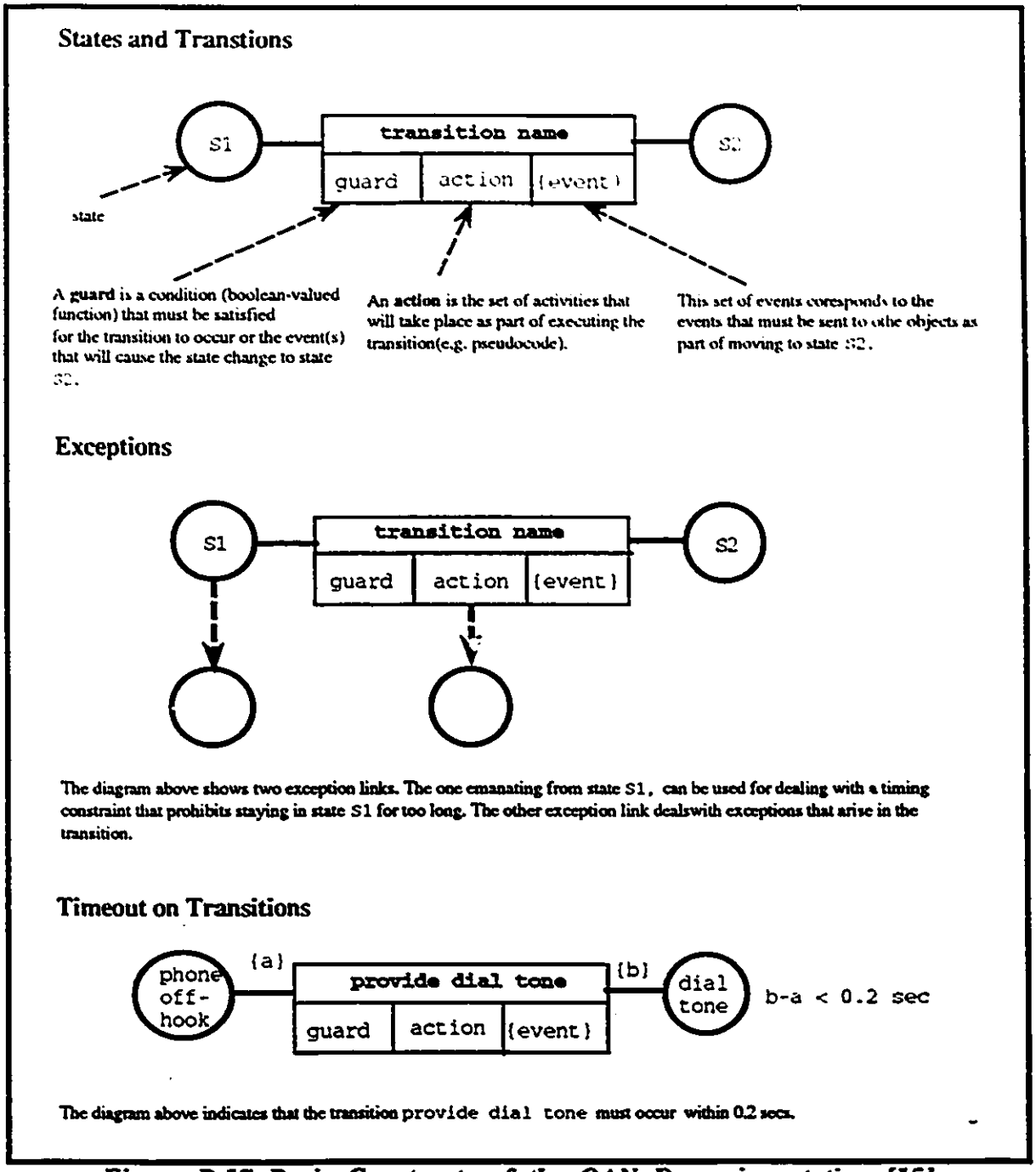


Figure B.17 Basic Constructs of the OAN Dynamic notation [15]

1.18 The OAN Object Interaction notation

The OAN (Object Analysis Notation) Object Interaction notation [15], describes the interactions among objects. Fig. B.18 illustrates the basic constructs of the notation as well as examples.

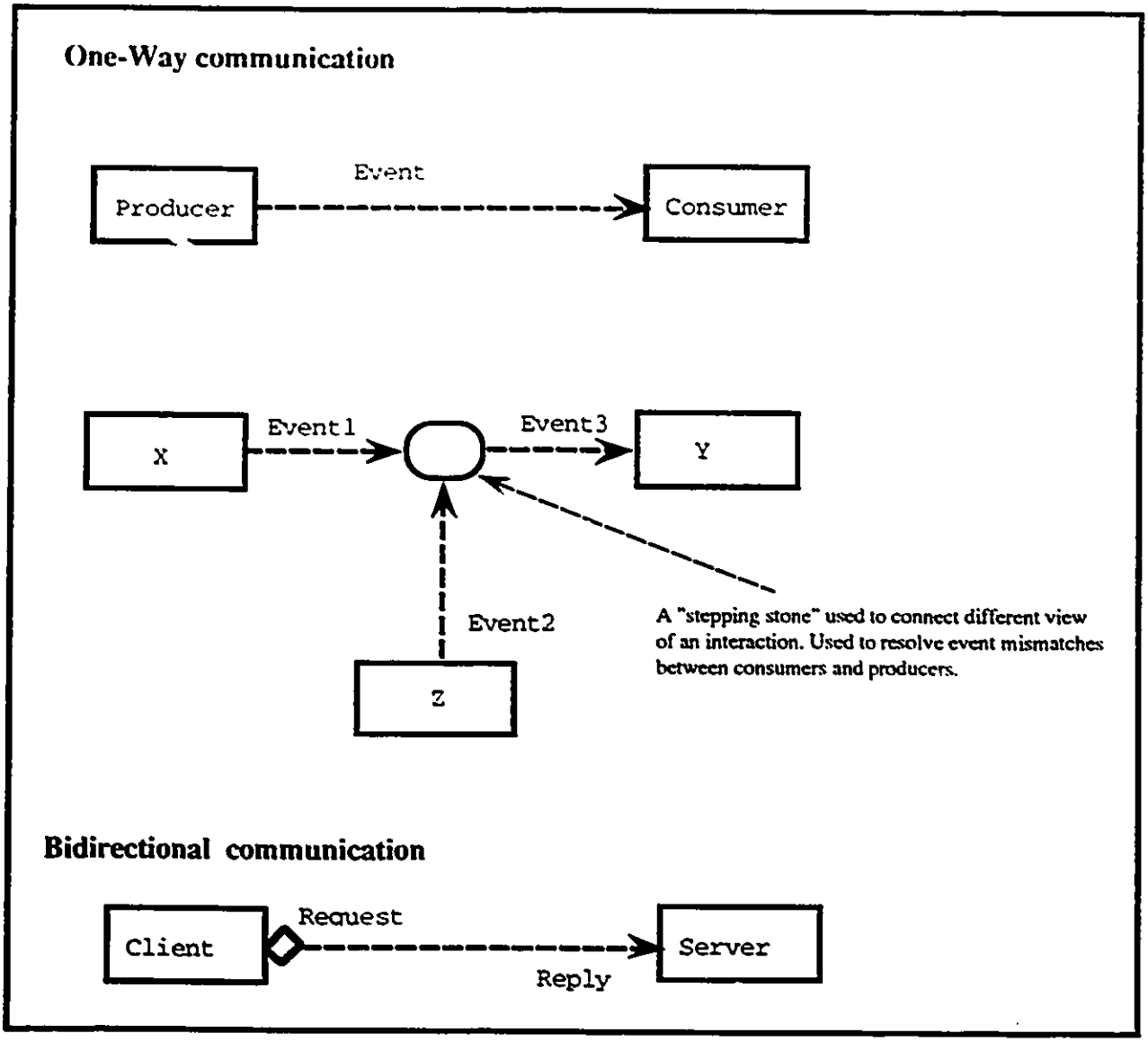


Figure B.18 Basic Constructs of the OAN Object Interaction notation [15]

1.19 The CRC notation

The CRC (Class, Responsibility, Collaboration) cards notation was introduce in [72] and used by several Object-Oriented methodologies as in [31].

It corresponds to an index card that represents a class of objects their behavior and their interactions. They are usually created in informal brainstorming sessions. The card usually contains the *class name*, their *responsibilities* (i.e. services that it provides) and *collaborators* (i.e. services required by the class to fulfill the responsibility).

The CRC card can also be used for Subsystems and Domains.

Fig. B.19 illustrates an example of a CRC card.

Class: BankAccount			
List of Superclasses: Account			
List of subclasses: CheckingAccount			
SavingAccount			
List of Responsibilities:	deposit	**List of Collaborators:**	Customer
	withdrawal		Teller
	balance		Bank

Figure B.19 Example of a CRC card for an Acc·· t

1.20 The Collaboration Graph notation

The Collaboration Graph notation [31], describes the collaboration between classes and subsystems. Collaborations are drawn as an arrow pointing to the contract[40] defining the behavior being requested. The constructs of a collation graph are shown in Fig. B.20.1. Fig. B.20.2 illustrates an example of a Collaboration Graph.

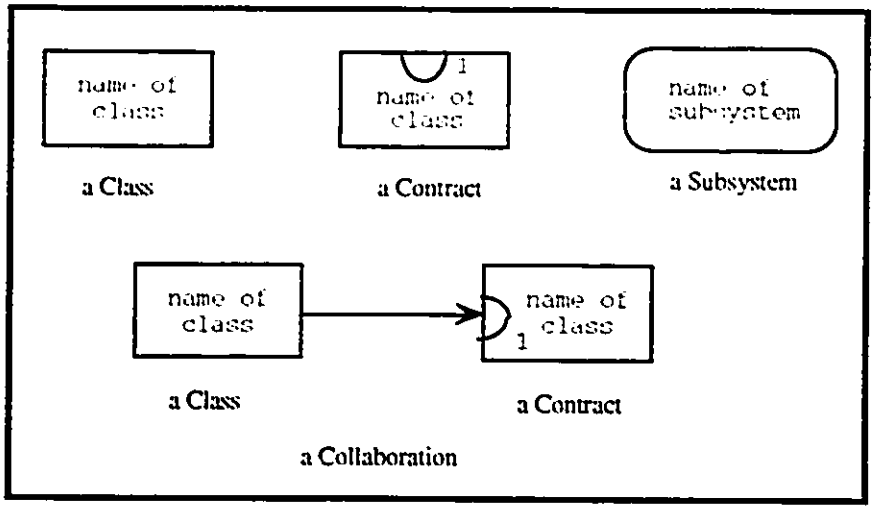


Figure B.20.1 Constructs of a Collaboration Graph [31]

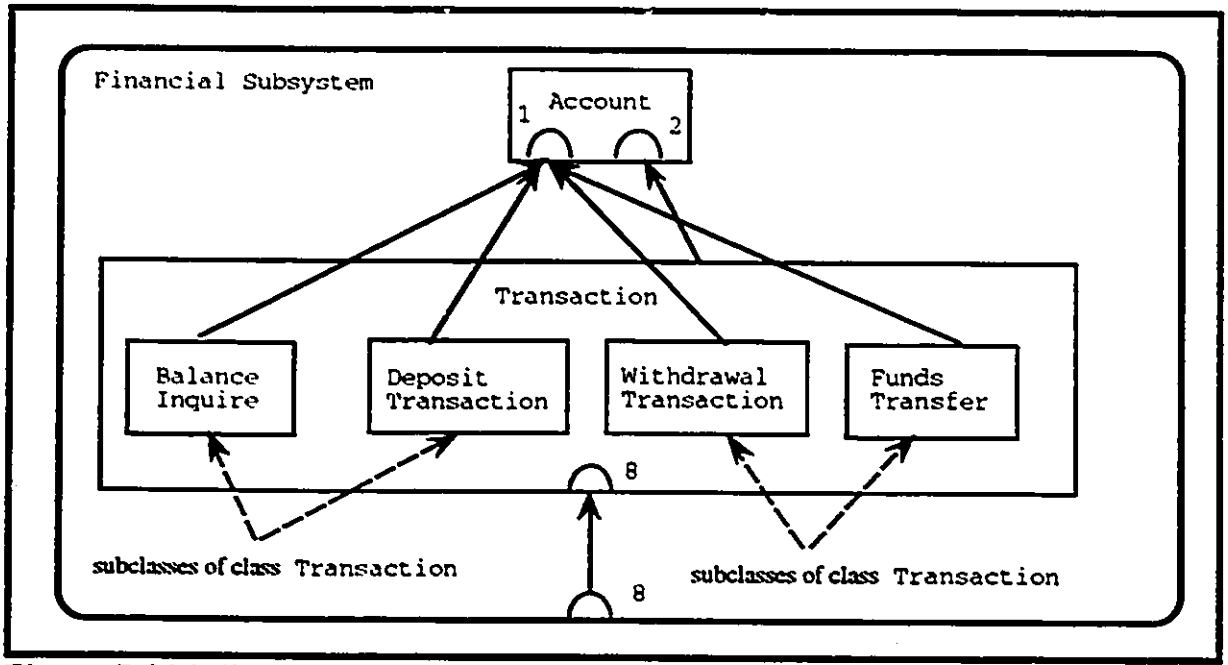


Figure B.20.2 Example of a Collaboration Graph for a Financial Subsystem of an ATM system [31]

[40]Contracts are defined in the next section

1.21 The Contract notation

The Contract notation [31,40], describes behavioral composition and the obligations on participating objects. A Contract can be defined as the list of requests that an object requesting services (i.e. client) can make of another object providing services (i.e. server).

Contracts can be defined informally as in [31]. In Fig. B.20.2 (in the previous section), the *server* of a contract corresponds to the class being pointed to by a collaboration; the *client* of a contract corresponds to the class that originates a collaboration. Thus in contract

1. the clients are `BalanceInquire`
   `DepositTransaction`
   `WithdrawalTransaction`
   `Funds Transfer`

   the Server is `Account`

Fig. B.21.1 shows the informal contract for Fig. B.20.2.

Contract 1

Server: Account

Clients: `BalanceInquire`
`DepositTransaction`
`WithdrawalTransaction`
`Funds Transfer`

Description: This contract defines a way an account can be accessed and modified.

Contract 2

Server: Account

Clients: `Transaction`

Description: This contract supports committing account changes to the database.

Contract 8

Server: `Transaction`

Clients: `ATM`

Description: This contract supports executing financial transactions.

Figure B.21.1 Example of a Contracts for Fig. B.20.2 [31]

Contracts can be defined formally as in [40]. "A *contract* defines the behavioral composition of a set of communicating participants. Each contract specifies the following important aspects of behavioral composition. Firstly, it identifies the participants in the behavioral composition and their *contractual obligations*. *Contractual obligations* consists of *type obligations*, where the participants must support certain variables and external interfaces and *causal obligations*, where the participants must perform an ordered sequence of actions and make certain conditions true in response to messages. Through *causal obligations*, contracts capture the behavioral dependencies between objects. Secondly, the contract defines *invariants* that participants cooperate to maintain. It also defines what actions should be initiated to resatisfy the invariant, which as a matter of course during program execution will become false. Lastly, a contract specify preconditions on participants to establish the contract and the methods which instantiate the contract" [40].

There are two important operation on contract: *refinements* and *inclusion*. These provide a way of expressing complex behavior in terms of simpler behavior. *Refinement* allows for the specialization of contractual obligations and invariants of contracts. *Inclusion* allows contracts to be composed from simpler sub-contracts.

"As an example, consider the behavioral composition in which a *Subject* object, containing some data, and a collections of *View* objects, which represent that data graphically, say as a dial, histogram and counter, cooperate so that at all times each *View* always reflects the current value of the *Subject*.." [40].

Fig. B.21.2 show the contract for the behavioral composition between `Subject` and objects of type `View`. The language supports the following:

- `P ---> M` upon reception of message `P`, `M` is sent
- Δ `v` the setting if instance variable `v`
- $\langle$ `o v:c:e` $\rangle$ repetition of an expression `e` separated by the operator `o` for all variables `v` which satisfy `c`.
- ordering or actions can be explicitly given by the operator `;`, an `if-then-else` construct, or be left unspecified by the operator `||`

"Contract `SubjectView` requires certain obligations of its participants `Subject` and the set of `Views`. Type obligations require a certain interface and data: `Subject` for example is required to support a variable of the unspecified type `Value` and the method `Notify ()`. Causal obligations require that receipt of a message leads to certain behavior. For example, the expression `Draw ()--->Subject--->GetValue()` specifies that each `View`, on receipt of a `Draw()` message, is required to behave in a way which leads to the sending of a message `GetValue()` to `Subject`.

Causal obligations are the essential feature of behavioral compositions. Through them we can infer that sending `Subject` the `Notify ()` message will lead to the sending of a `Update()` message to each `View`.

Participants in behavioral compositions often cooperate to maintain some invariant. However, during execution, this invariant can become false, requiring its resatisfaction. The invariant along with the action which lead to its satisfaction appears in the contract in the **`invariant`** section. For `SubjectView`, the expression *`Subject.SetValue(val)`* preceding the `--->` symbol signifies the action that will lead to the satisfaction of the invariant "for each *`View in Views: View`* **`reflects`** `Subject.value`".

Finally, the **`instantiation`** statement specifies that to initiate this contract between a set of `View` objects and a `Subject` object, the methods `AttachView()` and `SetValue()` must be executed with the appropriate argument." [40].

```
contract SubjectView
Subject support [
     value: Value
     SetValue (val:Value) ---> △ value (value = val) ;  Notify ()[41]
     GetValue () : Value  ---> return value
     Notify () ---> ⟨ || v : v ∈ Views : v ---> Update () ⟩
     AttachView (v : View) ---> (v ∈ Views)
     AttachView (v : View) ---> (v ∉ Views)
    ]
View : Set (View) where each View supports [
       Update () ---> Draw ()
       Draw () ---> Subject ---> GetValue () (View reflects Subject.value)
       SetSubject (s:Subject) ---> (Subject = :)
    ]
invariant
  Subject.SetValue(val) --->⟨ ∀v: v ∈ Views: v  reflects Subject.value⟩
instantiation
  ⟨||v:v∈ Views:⟨ Subject--->AttachView(v)||v ---> SetSubject(Subject)⟩⟩
end  contract
```

Figure B.21.2 Example of a formal Contract between classes `Subject` and `View` [40]

1.22 The ObjectChart notation

The ObjectChart [71] notation, describes the internal behavior of an object. It corresponds to the StateChart [52] formalism extended by augmenting states with attribute information. "ObjectCharts are extended StateCharts in which the effect of state transitions on attributes are specified" [71].

The lifecycle behavior of an object class may is expressed as a StateChart. States represent the various stages that an object of a class may go through.

Transitions are labeled with either the state changing services (i.e. operations) provided by the class or services required of other objects. A service required of another object is prefixed by the name of the object providing the service. Not all services change the state of an object (e.g. some report on the values of attributes). These services are called *observers* . In general, request for a service may not be allowed in all states. For example, it is possible to request the top of a stack only when the stack is not-empty. ObjectCharts show this type of information by adding the name and type of the allowable observers to each state.

ObjectCharts also allow to describe parameters of services. For example, *set (t:time)* is a parameter of service *set*). Parameters are partitioned into input and outputs. Input parameters of a service may only be read by the object providing the service. Output parameters are the values returned by the providing object. All output parameters of a services are listed after a "|". Thus *C. time(|t:time)* has output parameter *t*, and the value of *t* is set by object *C* that provides the service *time*.

Not all attributes of an object need to be visible. These attributes are treated as observers and are added to states, but depicted by writing them is square brackets to denote that the are ***hidden***.

Fig. B.22 shows an example of an ObjectChart. In Fig. B.22, when the alarm rings, it opens a window; when the alarm is not ringing, the window is closed; the alarm can stop ringing because a timer expires or can it was stopped by the user.

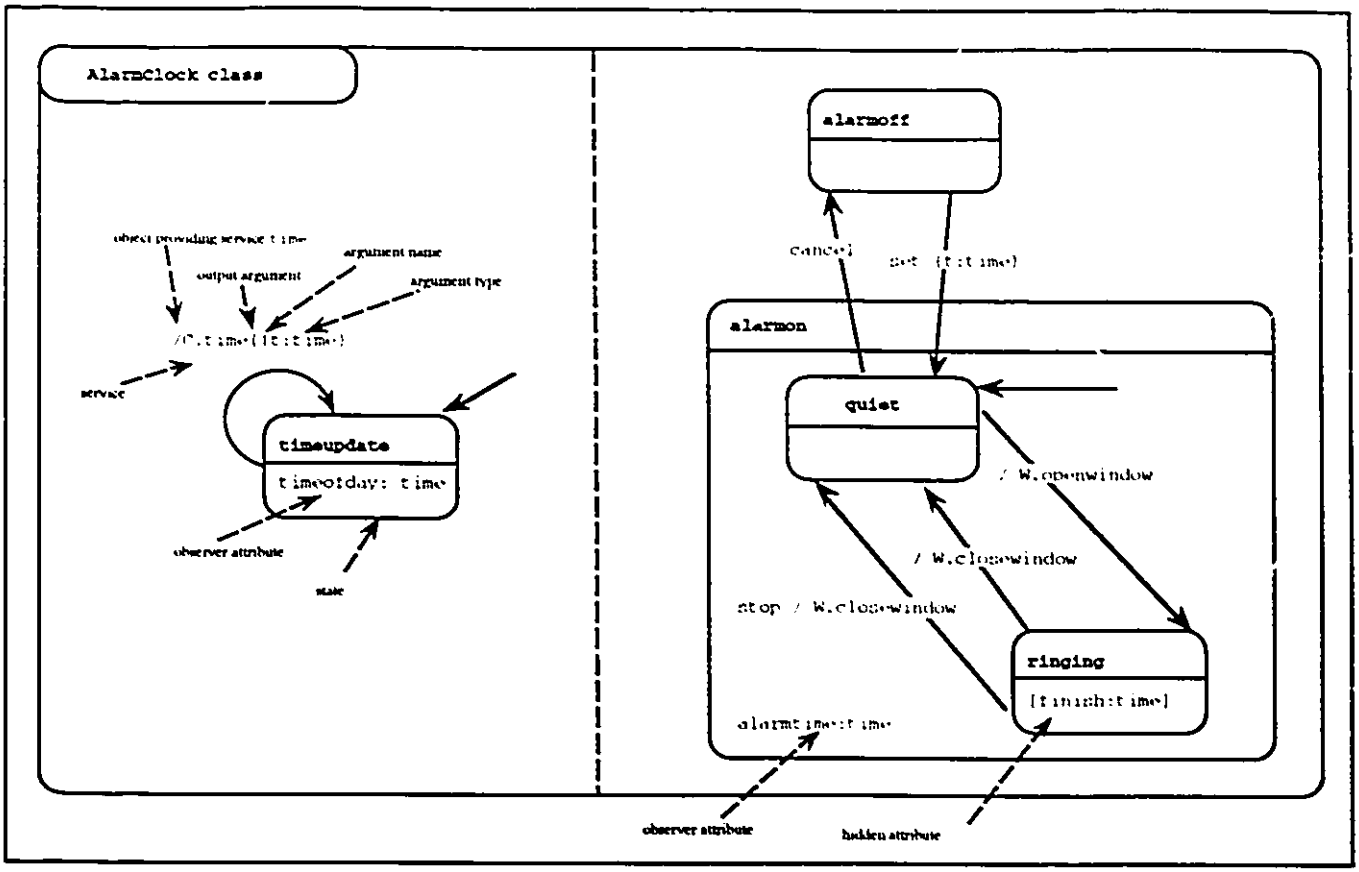


Figure B.22 Example of an ObjectChart for part of an Alarm Clock [71]