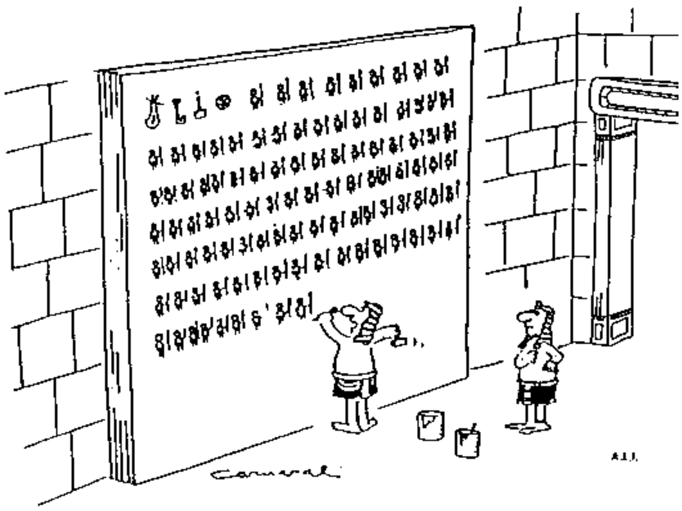
"To those who have borne
with us during our studies"

Acknowledgments

We would like to thank Prof. V. Ambriola, Prof. E. E. Doberkat and Prof. W. Schäfer for offering us the opportunity to gain this excellent experience at the University of Dortmund and for all the provided useful suggestions.
We also want to express our gratitude to Doris Schmedding for her advice and support throughout the development of our work, and for her helping hand at all times in private manners whether being big or small.
A special thank goes to Jörg Brunsmann and Wolfgang Emmerich for their suggestions and precious contribution, and for their readiness to reply to our regular outcry: *"hilfe"*. The first mentioned also corrected the last version of this thesis, for which we are truly grateful.
Others who have contributed to this work are J. L. Knudsen and E. Sandvan of the Mjolner Project. We wish to thank them because, during a face to face discussion, they offered us the opportunity to check the correctness of the theoretical part of this work.
Last but not least, we want to thank Werner Beckmann and our Dutch friends for their help in finding English mistakes.



Are you sure there isn't a simpler means of writing 'The Pharaoh had 10,000 soldiers?'

Contents

Chapter 1

Introduction

1.1 Motivations

In the past few years, Object Oriented techniques have finally made the passage from the programming-in-the-small island to the mainland of the programming-in-the-large. These techniques have found a natural application in different segments of computer science, from database systems to the representation of artificial intelligence and software engineering.

Object Orientation is a new way of thinking about problems using models organized around real word concepts. Object Oriented analysis, design and implementation are examples of areas associated with Object Orientation and they are often considered as constituting the most significant part of the software life cycle.

During the analysis phase the problem is understood. In this phase the relevant concepts and phenomena of the real word are identified and described. Then, the design phase is concerned with the construction of a precise description of these individualized concepts. During this phase the structure of the system to be developed is generated in terms of modules (classes) and the relationships between them. The resulting system is finally refined into an executable software system during the implementation phase.

The language used to express the result of the analysis phase is usually natural and/or graphical. Instead, graphical notations are often preferred for designing languages, while the implementation phase is supported by a variety of actually available Object Oriented languages. Object Orientation aims to integrate these three phases but some problems, depending on the differences between the used languages, still exist. For this reason many Object Oriented Analysis and Design (OOAD) methodologies have been developed in order to close the semantic gap between analysis and design, while the bridge between design and implementation is still an open problem.

Since it may be impractical to use a graphical notation without the support of a computer-based tool, a number of so called CASE tools have been developed to support the different OOAD methodologies. In general such tools allow the construction and manipulation of a graphical notation for analysis and/or design. In addition, such tools may support the generation of code skeleton in a programming language which may then be filled in to produce an executable software system. Unfortunately, in these tools it is difficult to recognize the original design in the code skeleton produced. This is due to the fact that usually there are some mechanisms in the graphical languages which are not directly supported by the programming languages.

Our work faces the problem to minimize the intellectual distance between real world problems and software solutions. This thesis shows how it is possible to integrate the three most important phases of the software development process and, in particular, how to close the gap between design and implementation.

The design methodology and the programming languages used are OMT [RBP+91] and Beta [LMN93]. A comparative study of several OOAD methodologies shows how OMT is the most suitable to be integrated with the Object Oriented language Beta. Our work realizes this integration by means of a mapping between the two languages. This mapping translates the most fundamental OMT concepts into those Beta concepts that are semantically equivalent. Moreover, the integration is concretely supported by the realization of a integrated Software Engineering Environment.

This Environment is created by developing a CASE tool complying a new approach that uses editor generators instead of common toolkits. The textual and graphical editor generators used to build the CASE tool are GENESIS [GOO94] and GraphProject [CI94].

By developing this tool, the bridge that closes the gaps between analysis and design, and between design and implementation does not remain only in a theoretical phase but became a useful and concrete solution. Moreover, by giving a description of the phases followed during this development, we also prove the utility of Object Oriented technology in the realization of an integrated Software Engineering Environment.

1.2 Structure of the Thesis

This thesis will further structured as follows.

In chapter 2 we give a brief introduction to the main features of Object Oriented languages. Moreover, we give an evaluation of Beta as Object Oriented language comparing it with the most known languages, and underlining those features that are Object Oriented and those that are not.

In the third chapter we give a rational for introducing a designing phase that supports the implementation in any Object Oriented programming language. In particular we point on the importance in using graphical notations during this phase.

In chapter 4 several OOAD methodologies are shown and compared, and their degree in supporting Object Oriented concepts is discussed. Four of these are investigated in more detail and the Rumbaugh methodology is chosen as the most suitable to be integrated with an implementation phase in Beta. Therefore the most fundamental OMT concepts and their notation are shown.

In chapter 5 we give a proposal of integration between the graphical language supported by OMT and the Object Oriented language Beta. Some solutions to simulate multiple inheritance in Beta are also given.

The following chapters discuss the realization of the tool.
In the sixth chapter the first two phases of the development of our tool are treated. We first give the analysis of the requirements our tool must satisfy and then a design in an OMT-like notation that follows these requirements.

In chapters 7 and 8 a comparative study of several approaches to build the graphical and the textual editor composing the tool is presented. These chapters show how GraphProject and GENESIS provide the means to build efficient and well integrated tools.

The implementation of the tool and its user interface are described in chapter 9. In this chapter the two editors and the realization of the integration mechanism between them are treated separately.

Finally, chapter 10 contains some considerations about the realized work underlining the experiences made in developing this thesis. Some suggestions for further works are given as well.

Chapter 2

Object Oriented Languages

In this chapter the characteristics of the Object Oriented languages (later called O.O. languages) are presented. Then the Beta language is introduced estimating it on the basis of these characteristics and a comparison with other well know O.O. languages. Finally, in order to increase the comprehension of the next chapters, the most important Beta constructs are explained.

Wegner Definitions

A widely accepted definition of the term Object Oriented is provided by Wegner [Weg92] who presents a necessary and sufficient set of criteria for O.O. languages. Wegner distinguishes three categories of languages:

- *Object Based* languages
- *Class Based* languages
- *Object Oriented* languages

We see now the definitions and the relation between the three categories.

Definition: *A language is* **Object Based** *if it supports objects as features.*

Definition: *An Object Based language is* **Class Based** *if every object belongs to a class.*

Definition: *An Object Based language is* **Object Oriented** *if every object belongs to a class, and if a class hierarchy can be defined by a inherit mechanism.*

Object Based languages are a proper subset of Class Based languages which in turn are a proper subset of O.O. languages. Ada is an example of non Class Based language and CLU [Kee89] is an example of non Object Oriented language.

Characteristics

The basic O.O. languages characteristic is that they are close to our own natural perception of the real world. They allow to model the real world by a set of *objects* -

computational entities each having a particular well defined behaviour, which communicate each other by message exchange. Moreover, objects are instances of *classes* which define the interface for the external world, and the objects behaviour by the definition of the operations. Each object in a class has a proper *state* that evolves independently from the other objects, and in particular from the other objects in the same class. The state depends from the value of the attributes. Moreover, classes can be arranged into hierarchies, offering in this way a natural and powerful mechanism to model the reality. Besides, allowing hierarchy organization of classes, O.O. languages give rise to strong constructs for supporting incremental program modifications - operations may be changed or added without changing the underlying model, and software developers are able to reuse existing software components when develop new one. For these reasons a large number of programmers practice Object Oriented programming. They use a number of different languages such as Simula, Smalltalk, C^{++} [SR90], Eiffel [Swi93] and CLOS. These languages have a common core of language constructs which to some extent makes them look alike.

2.1 Object Oriented Programming and Beta

Beta and the Mjolner Beta System

Object Oriented programming originated with the Simula language developed at the Norvegian Computing Center, Oslo, in the 1960s. Beta is a modern language in the Simula tradition. The Beta project was initiated in 1975 as part of the so called *The Joint Language Project*. People from Aarhus University, Aalborg University Center, and the Norwegian Computing Center, Oslo, participated in that project. The Mjolner Beta System is a software development environment supporting the Beta language which includes an implementation of the Beta language. In addition the system includes a number of other tools and a collection of libraries that provide a number of predefined patterns and objects. The Mjolner Beta system was originally developed as part of the Nordic Mjolner project with participants from Sweden, Norway, Finland and Denmark. Draft version of the Beta language we have seen and pre-releases of the Mjolner Beta System have been used for teaching Object Oriented programming as a second years course in programming language in two departments at Aarhus University. They have also been used, and they are actually used, for teaching at number of others places including the Universities of Copenhagen, Oslo, Bergen, Odense and Dortmund. Draft versions have also been used for Beta tutorials given in the OOPSLA '89, 90, 91' conferences, at the TOOLS '91 and '92 conferences, and at EastEurOOPe '91, [LMN93]. Although Beta is primarily intended for the Object Oriented style of programming, it contains comprehensive facilities for procedural and functional programming (is a *multi-perspective language*). As pointed out by others (Cox, 1981: Nygaard and Sorgaard, 1987), a programming language should support more than one style. Simula and C^{++} are multi-perspective languages too. We think that, even if in this case the programmer has more possibilities in writing programs, the use of a multi-perspective language in a teaching environment is not a good way if you want to learn Object Oriented methodology.

We discuss now the various features of O.O. languages that are supported by Beta, sometimes making some comparisons with others well known O.O. languages like C^{++}, Smalltalk-80 and Eiffel.

A Single Abstraction Mechanism: the Pattern

Beta replaces classes, procedures, functions, attributes and types by a single abstraction mechanism called the *pattern*. Pattern is a further generalization of the Simula class construct, and pattern attributes of a pattern correspond to interface operations in Smalltalk classes.

Encapsulation

By the use of this abstraction, Beta provides uniform access to both attribute and operations (as in Eiffel) but unlike them it does not provide encapsulation. In Beta, code associated with one class can directly access the attributes of another class without "asking for it" by invoking an operation of the object. Eiffel provides perhaps the finest control of encapsulation through its *export* statement, which lists attributes that can be read and operations that can be executed from outside. Many other languages (such as Smalltalk) forbid direct access to the attributes of another object or (as in C^{++}) permit attributes to be declared either public or private. Beta contains no such mechanism that allows to divide modules into interface and implementation modules. We think that it is a big lack for an O.O. language. Encapsulation is important because it prevents a program from becoming so interlaced that a small change has massive ripple effects. Without any facilities that ensure this property, it is a programmer task to ensure encapsulation by limiting the scope to any one method. He needs to exactly define the boundaries of visibility that each method requires.

Inheritance and Polimorphism

Beta contains facilities for inheritance, method resolution and, like C^{++}, the ability to override an operation in a subclass is only available if the operation is declared *virtual* in the superclass. Therefore, we can have objects that belong to distinct classes that react to messages with the same name to perform similar tasks even if the code that implements the methods can be different in the various classes. This situation reveals that we have *polimorphism*: we can access different implementations of the same operation using the same name. Virtual operations, superclasses and subclasses are generalized as patterns too. In particular, respectively as *virtual patterns*, *superpatterns* and *subpatterns*. Declaring virtual operations, the need to override a method must be anticipated and written into the origin class definition. Unfortunately, the writer of a class may not anticipate the need to specialize subclasses or may not know what operations will need to be refined by a subclass. This means that the superclass often must be modified when a subclass is defined. We think that this places a serious restriction to the ability of reusing library classes by creating subclasses.

How Beta Sees Inheritance

Although inheritance of 'code' is often considered the major benefit of sub-classing, in Beta language inheritance is mainly intended for hierarchical classifications of concepts (even if it may be used for code sharing as well). This is one of the major differences between the American and the Scandinavian school of Object Oriented programming [Coo88]. Moreover, O.O. languages differ in their implementation of inheritance. [KL88] discusses three independent dimensions for classifying inheritance mechanisms: static or dynamic, implicit or explicit, and per object or per group.
On the basis of this classification, Beta language is:

- *static*: in the sense that inheritance is bound at compiler time;
- *implicit*: the behaviour of the object depends on its class that cannot be changed;
- *per group*: inheritance characteristics are specified for a class and not for specific objects;

Multiple Inheritance

Unlike many O.O. languages like Eiffel, CLOS and C^{++}, Beta does not support multiple inheritance, due to the lack of a profound theoretical understanding, and also because the current proposal seems technically very complicated, [LMN93,p.107]. Multiple inheritance will be treated in more detail in the next sections.

Beta Is Strongly Typed

Beta is a strongly-typed language in the sense that attributes and references may be declared as belonging to a particular class or one of its descendents (in a Beta concept: they are *qualified*). The premise behind strong typing is that it is easier to detect and correct static semantic errors at compile-time rather than at run-time; the compiler will generate more efficient code, and the qualification improves the readability of the code. The price for this choice is, of course, less flexibility for the programmer. However, the use of untyped reference in Smalltalk-like languages has the benefit that a recompilation of a class does not have to take related documents of the program into consideration.

Membership of Classes

Moreover, Beta allows the test for class pattern membership even if this attribute is considered a bad programming style. This is allowed by the meaning of *pattern variable*. In Simula and in Smalltalk it is also possible to perform this test. In C^{++} they have been deliberately left out, since they are viewed as violating the advantages of Object Orientation. As we will see in the next chapters, for us pattern variable has been useful during our study of multiple inheritance. We have used this facility to simulate inheritance of features that in our opinion can be seen as another way of information hiding violation.

Composition and Block Structure

Beta also contains facilities for supporting *composition*. Composition is a means to organize phenomena and concepts as a composition of other phenomena and concepts. Beta supports three kinds of composition: *Whole-Part composition, Reference composition* and *Localization*. Block structure is the Beta mechanism to support Localization. This is a natural and powerful mechanism that gives many advantages as locality and defines the scope rules of the language. Despite of this, block structure is found for example in Simula but abandoned in Smalltalk-80. Moreover, in Simula the use of nested classes is limited by a number of restrictions; Beta does not have this restriction. Whole-Part composition and Reference composition will be explained later in detail.

Library

Like several other O.O. languages, Beta contains a standard class library as part of its environment. The availability of the class library means that many components need not to be implemented, expecially general purpose data structures. Classes implementing various kinds of associations should be also available in a class library [RPB^{+}91,p.319] but since Beta does not explicitly supports associations, it does not provide this facility.

Beta supports automatic memory management: objects that are no longer referenced are detected and the memory allocated to them is released without requiring (or allowing) any explicit deallocation. This approach relieves the programmer of the responsibility of deciding when to allocate memory and avoids the risk of dangling object references that can be heated by explicit deallocation. Automatic garbage collection is supported also by CLOS, Eiffel and Smalltalk, while C^{++} requires the programmers to deallocate unneeded objects allowing the programmer to define a *destructor* function for every class, automatically called when a variable goes out of the scope. Automatic Garbage Collection

2.2 Beta and Its Basic Language Mechanisms

In this section we introduce the most important constructs of the Beta language. This will be useful for the understanding of the next chapters, but readers familiar with Beta may skip this section.

A Beta program execution consists of a collection of objects and patterns. An *objects* is characterized by a set of *attributes* and an action-part. Patterns are used to represent categories of objects with the same properties [LMN93]. The *object descriptor* is the basic syntactic construct in Beta and may be used to describe a pattern or a singular object. An object descriptor has the following form: A Beta Program

(# Decl1; Decl2; ... Decln
enter In
do Imp
exit Out
#)

and its elements have the following meaning:

- *Decl1; ...; Decln* is the list of the attribute declarations. The possible kinds of attributes are further described below.
- *In* is a description of the *enter-part* of the object. The enter-part is a list of input parameters which may be entered prior to execution of the object.
- *Imp* is the *do-part* of the object. The do part is a list of imperative statement that describes the actions to be performed when the object is executed.
- *Out* is the description of the *exit-part* of the object. The exit-part is a list of output parameters which may be produced as a result of execution of the object.

Enter, do and exit-parts are called together *action-part* and each of these elements may be omitted. It is strange that an object has an action-part, for example Eiffel does not support this kind of facility, but it is due to the fact that Beta has a single abstraction mechanism (the pattern) that is used also for describe operations in which the action part is needed.

Example

The example below illustrates most of the language mechanisms defined in Beta language. It gives a definition of a pattern *Address*.

```
Address:   (# Street:@Text;
              StreetNo:@integer;
              Town:@(#  Name:@Text;
                        CAP:@integer
                    #);
              theCountry:^Country;
              whichCountry :(#   do  theContry.Display #);
              printLabel:< (#    do
                                  {print Street,
                                   StreetNo, Town};
                                   INNER;
                           #);
           #)
```

Figure 2.1:

Address is the name of the pattern and has the attributes *Street*, *StreetNo*, *Town*, *theCountry*, *whichCountry*, and *printLabel* but does not have an action-part. We show below the various kind of used attributes.

2.2.1 Object Attributes

In our example, the attributes *Street*, *StreetNo*, *Town* and *theCountry* are object attributes.

Part Objects

Street, *StreetNo* and *Town* are *part-objects* of an *Address* object in the sense that they denote the same object during the lifetime of the containing *Address* object. This kind of declaration is also called *static reference* since *Street*, *StreetNo* and *Town* always denote the same static objects. It is also possible to specify part-objects of any pattern, that is also of user-defined patterns. Consider a pattern *Person* defined as below. Here the pattern *Address*, that has been just defined by the user, is used to define a part-object of *Person*.

```
Person:  (#   Name:@Text;
              Addr:@Address;
         #)
```

Basic Patterns

The patterns of the part-objects *Text* and *Integer* correspond more or less to what in some languages would be standard, predefined simple (type) classes. In Beta this kind of patterns are called *Basic patterns*. A number of predefined basic patters for commonly used data types such as integer, boolean, char, text and real and their operations are available. Any instance of *Address* will have a *Text* object, an *Integer* object and a *Town* object as fixed parts. These objects are generated as part of the generation of *Address*.

Dynamic Reference

theCountry is a *Dynamic reference*, i.e. it is a reference to a separate object since this application keeps information on countries in a separate object. The attribute *theCountry* is said to be *qualified* by *Country*. In addiction such reference is variable in the sense that it may denote different objects (instances of the same class) at different points of time. A dynamic reference may be given a value by means of a *reference-assignment*. There are two different ways to use a dynamic reference. For example if you have declared the following static reference:

```
Coun:@Country;
```

a reference assignment of the form:

```
Coun[]->theCountry[]
```

implies that the object denoted by *Coun* is also denoted by *theCountry*.
It is also possible to dynamically create objects by the execution of actions. The following evaluation creates an instance of the pattern *Country* and the result of the evaluation is a reference to the newly created object that is assigned to *theCountry*.

```
&Country[]->theCountry[]
```

The symbol & means "new" and the symbol [] means that the reference to the object is returned as the result of the evaluation.

2.2.2 Pattern Attributes

whichCountry and *printLabel* are *Pattern attributes* used as procedures. A pattern attribute may be used as a template for generating objects that have a state that changes over time (*Class pattern*), or as a template for generating an action sequence.

Class pattern

A class patterns is a pattern describing a certain number of attributes. The content of the attributes defines the state of the objects (of this class) that can be changed by the other objects by invoking operations that change their values.

Procedure and Functional Pattern

Patterns used as templates for generating actions sequence are divided into *Procedure patterns* and *Functional patterns*. From a modelling point of view, a Procedure pattern is used to represent temporary state information when an object that belongs to the pattern is generated. A Functional pattern is a pattern that computes a value on the basis of a set of input parameters. The input values are entered in the enter-part, and the computed value is returned via the exit-part. The computed value depends solely on the input values and, in addiction, the computation of the value does not change the state of any other object (there is no side-effect) [LMN93,p.43]. For example, *T* is a functional pattern that takes as input two integers, and can be part of the following evaluation:

```
(1,2)->T->A
```

The execution assigns the integers 1 and 2 to the enter-part of the inserted item *T*, and causes the instance to be executed. Finally, the exit-part of the *T* instance is assigned to *A*. In this example an instance of *T* is generated as a permanent part of the object that computes the evaluation. In this case, *T* is called *inserted item* and it could be either a functional or a procedure pattern. In reality the difference between procedure and functional patterns is something vague states. We have not found a definition that clearly describes the differences between the two kinds of patterns, while we have seen examples of procedure patterns with exit-part and functional patterns that give in output values of an own attribute. In the example above, the pattern *Address* is a class pattern, while *printLabel* should be a procedure pattern.

2.2.3 Virtual Patterns

A *virtual pattern* is a particular kind of procedure or functional pattern. In the above example the pattern *printLabel* is declared as a *Virtual pattern* attribute. A virtual pattern can not be completely redefined, but only further *extended* in *subclass* patterns of *Address*. An example is the following:

```
AddressOfCompany:  Address
                   (#      Name:@Text;
                           printLabel::<(# ..{print Name}..#)
                   #)
```

Here *AddressOfCompany* is a subpattern of *Address* and *printLabel* is specialized printing the *Name* attribute. This declaration of *printLabel* is called *binding declaration.* Extending a virtual pattern implies to define it as a subpattern of the definition given as part of the virtual pattern specification. In subpatterns, attributes can be added and the execution of the special imperative *INNER* implies the execution of the actions of the subpattern. In this way a subpattern is a specialization of another pattern - the superpattern and the INNER imperative is used for specialization of actions. The enter-part for a subpattern is a concatenation of the enter-part of the superpattern and the enter-part specified in the subpattern, and similarly for the exit-part.

We describe now some other constructs and concepts supported by Beta that are not used in the above example.

2.2.4 Pattern Variable

A *pattern variable* is another dynamic concept of patterns. It may be assigned to different patterns during program execution. A pattern variable is defined as follows:

```
F:## T
```

where *F* is the name of the pattern variable and *T* its qualification. *F* may be assigned to any pattern which is *T* or a subpattern of *T*.
Consider the following subpattern definition:

```
T1:T(# ... #)
```

then:

```
T1##->F##
```

assigns *T1* as a pattern to *F*, that means that the entire structure of the pattern *T1* is assigned to *F*. *F* may be also used to create instances like in the evaluation of the following forms:

```
F        &F        &F[]
```

just as for ordinary patterns. *F* may be assigned to a new pattern or to another pattern variable.

One of the interesting use of patterns variable is for *testing pattern membership*. In fact it is possible to compare patterns variable like: Testing Pattern Membership

```
F## = T1##        F## < T1##        F## <= T1##
```

where = means the same pattern, < means that the left-side is a subpattern of the right-side, and <= means that the left-side is either equal to the right-side or a subpattern of the right-side. Patterns variable make patterns *first class* values in the sense that a pattern can be assigned to a variable, passed as a parameter to a procedure pattern, and returned as a result of a procedure pattern.

2.2.5 Composition

Composition is important in modelling real world. It is a means to organize phenomena and concepts in terms of components of other phenomena and concepts. There are a number of different ways for making composition. Beta supports three different approaches: Whole-Part composition, Reference composition and Localization [LMN93,p.308].

Whole-Part Composition

One important form of composition is the structuring of phenomena into wholes and parts.

Definition. *The part-of relation is a relation between a phenomenon and one of its part phenomena.*

Example

A *Microcomputer* may be considered as consisting of parts like *Mouse*, *Monitor*, and a *Keyboard*, i.e. *Mouse*, *Monitor* etc. are part-of a *Microcomputer*. The following example shows how Whole-Part composition is supported in Beta. The example shows a pattern describing the concept of a microcomputer:

```
Microcomputer:
        (#  theMonitor:@Monitor;
            theMouse:@Mouse;
            theKeyboard:@Keyboard;
        #)
```

The part-of relationship gives rise to a *part hierarchy* in which *Microcomputer* is the aggregate object and *theMonitor*, *theMouse* and *theKeyboard* are the part objects.

Reference Composition

A *reference* is a component of a phenomenon that denotes another phenomenon. Composition of references gives rise to a *has-ref-to* relation.

Definition: *The has-ref-to relation is a relationship between a phenomenon and one of its components, being a reference to another phenomenon.*

Example

The following example shows how reference composition is supported in Beta. The example describes the concept of a *Hotel Reservation*:

```
HotelReservation:
        (#  aPerson:^Person;
            aHotel:^Hotel;
            aRoom:^Room;
        #)
```

The components *aPerson*, *aHotel*, *aRoom* are reference components of a *HotelReservation*. The difference from the concept of Whole-Part composition is that, in this case, a person, an hotel, and a room are not parts of an hotel reservation, but they are only references that can be useful to take informations.

Localization

Localization is a means for describing the fact that the existence of phenomena is restricted to the context of a given phenomenon.

Definition: *The is-local-to relation is a relationship between a compound phenomenon and a locally defined dependent component phenomenon.*

An example of localization is a grammar symbol. A grammar symbol cannot exist apart to a grammar: it has not an independent existence. This example is translated in Beta with the following patterns: Example

```
Grammar: (#
         ...;
         GrammarSymbol:(# ... #);
         ...;
         #)
```

Chapter 3

Rational for Graphical Design

The designing strategy, as both a decomposition problem and system development paradigms, has made impressive inroads into the various areas of computer sciences. Substantially, it has been demonstrated that during the software development process, the introduction of a designing phase removes several barriers to programming [RS95]. Specifically, designing methodologies should not force users to:

- Build up desired program behaviour from low-level programming constructs as iteration and conditionals;
- Bridge the "semantic" gap between their conceptual model of the problem to be solved and the computation model of the program.

This chapter discusses how these barriers can be lowered through techniques with familiar, visible representations that let the user express software systems in terms pertinent to the problem to be solved. The most important thing is a way of thinking abstractly about a problem using real world concepts, rather than computer concepts. For this purpose some designing methods have been developed to inject some discipline in this abstraction process. Despite their differences, all these methods have elements in common: the notation, i.e. a language for expressing each model, the process followed during the construction and the tools that support the designing. In this process, we experienced that a graphical notation is more useful for a software developer in visualizing a problem in a more concise way and without prematurely resorting to implementation.

3.1 Software Design

Software Development Process

The software development process is a modeling process which involves the identification of relevant concepts and phenomena in the system being modelled (the *Referent system*) and the representation of these concepts in a model system. The (possibly informal) description of phenomena and concepts identified during an *analysis* phase has to be transformed into a formal description (*Model system*). The construction of this formal description, i.e. our physical model, is realized during the *designing phase* and can be later refined into an *executable program* during the *implementation* phase.

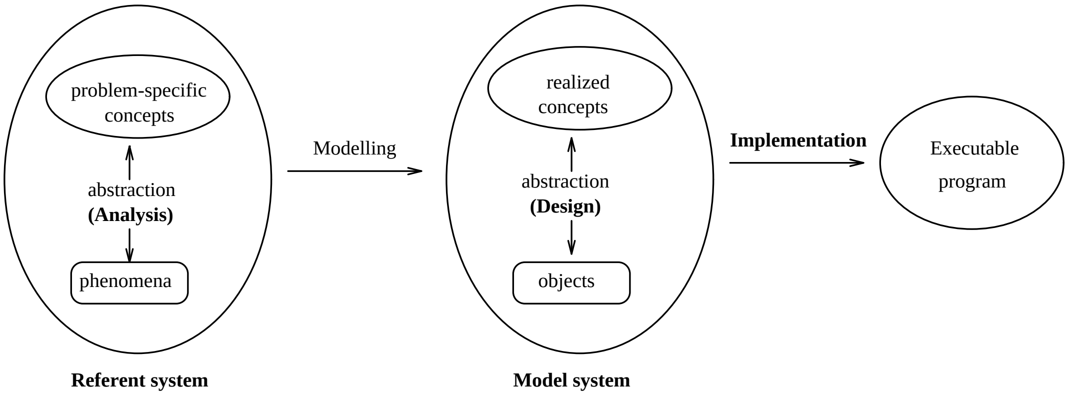


Figure 3.1: Software Development Process

Restrictions

The designing process is faced with the problem that not does the designer restrict the realism of his model by considering only a part of the word. Moreover, the modelling process *has* also to take into account the restriction imposed by the environment in which the model system is built. For example, in general the expressiveness is limited by the language used to describe the model.

Designing

For complex systems it is often difficult to individualize the whole structure and it is important to find out all facts, choices, assumptions and decisions, and make them explicit. In fact, designing means making decisions about the artifacts being designed: terminology, components, system structure, user interface and data structures. The result of this phase is a design. A design is a structure which includes a set of the components of the system, specifications of their interfaces and the way in which the components are related to each other.

A Good Design

A good design can be verified and analyzed with respect of its user requirements and resource constraints. Yet, a good design allows implementation freedom in the choice of the details of the components and sub-components. In fact the level of abstraction chosen by the user should be *expressive* enough to allow to state a solution for the problem, but at hight level enough to shield users from implementative decisions and details he does not want to be concerned with [RS95].

Importance of Designing

Design is one of the main issues in software engineering. However, other issues like requirements analysis, process metrics, coding, event reporting, cost control, testing, etc. are of great interest too [ZSG79]. Even if designing is useful to describe a variety of relationships among problem and/or solution concepts of interest, it has some limitations. In fact, it covers only the syntactic level of the analysis, making rigorous investigation of semantic properties difficult. Nevertheless, the *design system* however is the most important, because it determines the structure of the system and because of that it will affect all other aspects of the system development sooner or later.

Graphical Notations

We share the view of many authors, such as [RBP+91, WN95, CY91], who propose using a graphical notation instead of a textual notation during the designing phase of systems. In most cases the difference between a design language and a programming language is just the use of a graphical notation instead of a textual notation. Graphical notations have been used for many years to support analysis and design and they are taking place in the teaching environment. For instance, also here at the University of Dortmund, during the Software Praktikum course, the students develop their graphical designs with the tool Opus. Opus is developed by STZ (Gesellschaft für Software-Technologie mbH) and it is composed of three editors: a graphical editor, to describe the structure of the system, and two syntax-driven textual editors for refining the code by specifying the module interfaces and their bodies. The Opus graphical notation [Lew88] in fact, structures the system in different kinds of modules, related only by means of a *use-relationship*. Hence, the language is not able to support designing for O.O. programming languages. This is one example of how the semantic gap between designing and implementation phases is still an actual problem to be solved. The next section explains the reasons why we have given a so big importance to the use of graphical representation during software development.

3.2 Graphical Representation

Graphical notation provides the most expressive power to capture the concepts of a design.

Why Graphical Representation

The reason for using graphical notation is that generally the best way of capturing descriptions and reasoning is with knowledge representation systems - a system in which everyone has its own representation of the knowledge [IJK90]. Although knowledge representation systems are usually applied to specific application domains, they also need to be applied to graphical systems. If the visual reasoning of the graphical system is described at the same higher-level language as the application domain reasoning, new ways of displaying complex ideas can be synthesized more flexibly and quickly.

Understand a Design

Usually a graphical system is a kind of representation used to easily describe complex problems in a natural notation. Alternatively, a user may use a standardized notation whose meaning is known by everyone in the culture, like an algebra in a scientific context. But, in either case, there is a specific association between visual objects and description for these objects. There are differences amongst individuals concerning the degree to which pictures are essential for them, and only few people do not need them at all. Therefore it is no surprise that many software development methodologies have certain pictorial representations built-in, and that some methods started as a graphical representation, with the semantics, methodology, and tools added later. The graphical representation and its semantics constitute the graphical notation of the method.

As for any other notation, a graphical notation as well is useful only if it is well defined. This is why we now want to individualize the most important features a good graphical notation should have. We also see how its organization is usually divided on the basis of the semantic framework.

Guidelines for Graphical Notations

For an easy usage, a graphical notation should be composed of simple symbols that can be easily sketched by hand or automated on a computer. The number of different basic symbols should be small. In fact the terms of the visual vocabulary should be enough to describe all the possible situations. On the other hand, the number should be small in order to let the developer free to design without the bore of repeatedly looking into the manual. Moreover the visual impact of the arrangements of the basic symbols should connote the semantics of situations they represent - the notation should be intuitive. As an element must be recognized as being in a certain semantic category, the overall notation must be unambiguous and the same symbol cannot be used to represent different features in the same context. But when semantic is analogue in different contexts, that analogy should be exploited by reusing the same symbol to visually reinforce the analogy.

Graphical Notation Subdivision

An increasing number of graphical notations has been developed in order to define expressive designing methodologies. Usually these methodologies have different semantic frameworks but most of them are structured into pictures in the same way (as described in [FLP]).

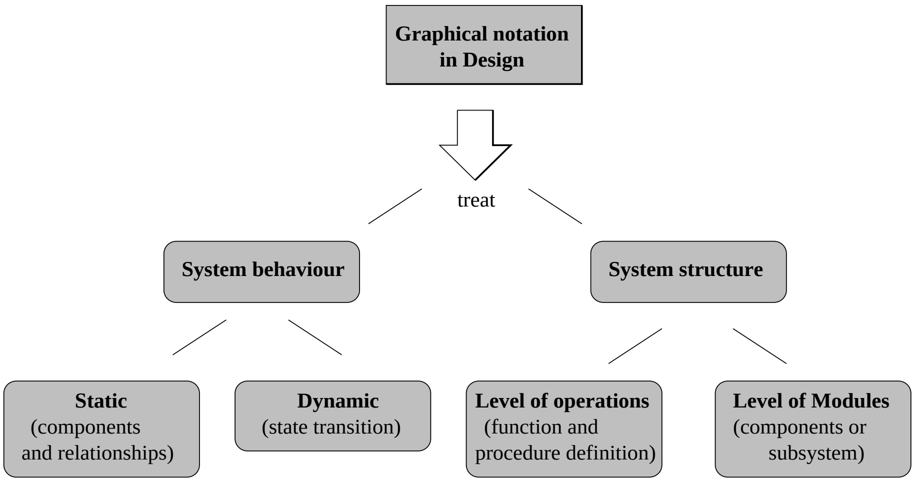


Figure 3.2: Graphical Notation Subdivision

Roughly speaking, we use to divide pictures on the basis of two options: either the picture represents some aspects of the behaviour of the system being designed, or the picture is about the *system structure* description. In the case it is concerned with the *behaviour* of the system, usually this behaviour is divided into *static* and *dynamic* behaviour. By "static behaviour" we mean all facts that apply to an individual state, or that apply in a static world view. This behaviour is usually represented by the definition of the system's components and the relationships between them. By "dynamic behaviour" we mean all aspects of system behaviour which are related to state transitions such as write operations on data structures, dynamic object creation, movement of physical devices, etc. As shown in fig. 2.2, if the picture is concerned with the *structure of the system*, usually we find a further subdivision depending on the level of detail represented in the picture. This can be the *level of the operations* (function and procedure definitions, but also assertion and expression), or the *level of modules* (components or subsystems). The reason for this division stands in fact that the abstraction level of the language component should match the user's conceptualization of the problem. In this way, the developer is not constrained by a deep and detailed description of the system too early.

3.3 Summary

In this chapter we have focused on the importance of graphical designing methodologies. In fact even if usually the current emphasis is on implementation rather than analysis and design, this leads to restricting design choices very early and to a product that is not of a good quality. Therefore, we think that it is only when the inherent concepts of the application are identified, organized and understood that the details of data structures and functions can be addressed effectively. Moreover, we have seen that using graphical notations instead of textual notations helps the developers to visualize the real world concepts they are going to model, and to express problems in terms of real world concepts on an abstract level. Finally we have seen the characteristics a good graphical notation should have and how the graphical notation is used by design methodologies to express aspects of the system being designed. These considerations provide the basis for a good choice of an OOAD methodology that graphically supports the designing phase. Several of these methodologies and some comparisons between them are presented in the next chapter.

Chapter 4

OOAD Methodologies

In this chapter we present a comparative study about several OOAD methodologies in order to find the most suitable to be integrated with an implementation in Beta. Four of these methodologies are treated in more detail, and the reasons to choose OMT as the methodology to adopt for our mapping are given. Moreover, the last section gives an overview of the most fundamental OMT concepts and their notation.

Structured Analysis and Design

The Object Oriented software development approach has complex and deep roots which it evolves from. In traditional software analysis and design, the most widely used methodologies are the Structured Analysis/ Structured Design (SA/SD), from Yourdon and De Marco, and the Jackson Structured Design (JSD) [Hsi92]. These traditional analysis and design methodologies use various notations as data flow diagram, process specification, data dictionaries, state transition diagrams, and Entity Relationships (ER) diagrams to logically describe the software system.

Why OOAD

However, in the last years, the emergence of Object Oriented technologies has changed the process and practice on how contemporary software systems are built. There is a growing evidence in literature that SA cannot be used effectively when subsequent design and implementation has to be done in an Object Oriented manner. Moreover, the same opinion has come out as a result from a panel discussion at the OOPSLA/ECOOP'90' conference. The reasons for this strong assertion are based on the fact that SA first characterizes and only subsequently derives the members in the data dictionary. In this way the identification of the classes is precluded; it does not exploit inheritance, and prevents encapsulation storage and behaviour features. Hence, to remedy these disadvantages and to accommodate the need for modelling the O.O. programming languages applications, research in new OOAD technologies has been developed in both academic and industrial community.

New OOAD Methodologies

Moreover, not only technologies but also new OOAD methodologies have been developed. Between them we have seen those that support a graphical notation in the design phase in order to choose the most valid to use in our work. These are Bailin [Bai89], Booch [Boo91], Coad & Yourdon [CY91], Edwards [Edw89], Shlaer & Mellor [SM89], Gibson [Gib90], Wirfs-Brock [WBW+90, MP92], Wasserman [WPM90], Rumbaugh [RBP+91], Kurtz [KWE91], Odell [OM], BON [WN95] and Page-Jones [PJC+90, MP92]. During our research, we have seen that Coad & Yourdon, Shlaer & Mellor, Booch, and Rumbaugh are the four dominant methodologies that have been used and practiced by various applications and user communities. As the methods of Bailin, Edwards and Gibson do not support inheritance, they can not be considered to be Object Oriented, while Kurtz and Odell are Object Oriented, but support an unusual freedom which allows classes to have multiple names and allows objects to change class membership. In the next section some of these methods will be classified by means of a table (fig. 4.1) complying a list of important components in OOAD based on the Colter [Col84] and Pressman [Pre87]. This will be useful to show what the methodologies have in common and where they differ. Unfortunately, because of the shortness and not well documented publications available for some methods, we have not been able to complete the table with all the methodologies seen. On the contrary, because of their characteristics, the most important methodologies, which are Coad & Yourdon, Shlaer & Mellor, Booch, and Rumbaugh, will be explained in details in the next section.

4.1 A Comparative Study of Nine Methodologies

In order to compare some of the OOAD methodologies we have listed above, we want to explain them basing on Colter and Pressman criteria. They describe a number of analysis and design methodology characteristics that can be applied also in the object arena. OOAD components include:

- *An OOA process:* A general domain analysis technique (i.e. an analysis procedure, the "how to's");
- *An OOD process:* A solution domain modelling technique (including specification of interface objects and other solution domain objects);
- *OOAD representations:* For structure, function and control at different levels of abstraction;
- *An OOAD complexity abstraction and management mechanism:* For portioning the problem and managing the complexity of the system.

Nine of the mentioned methodologies are classified in fig. 4.1 that shows a table that follows the Colter and Pressman criteria. The table contains static (structural) and dynamic (control) considerations in each of the four major areas. The first area is about the OOA Process. OOA models the problem domain by identifying and specifying a set of semantic objects that interact and behave according to system requirements. The second area is the OOD process. It models the solution domain, which includes the semantic classes and interface, applications, and base/utility classes identified during the design process. OOD should also be language independent in order not to restrict implementation choices which will be taken only during the physical design. These are the two primary components in OOAD. In addition an OOAD method must graphically or textually represent its results. Also OOAD representations are needed for static and dynamic views of the system. Static representation should portray objects, relationships, attributes, and methods. Dynamic representation should depict communications (message passing) and control. In addition, the constraints specified for a system may be supported by the notation. The last major component of OOAD is the complexity management. Complexity management is important from both the conceptual and the visual standpoint. Different views of a system have been defined. These views are all meaningful and can help in managing a large, complex design.

First Evaluation of the Table

In a first overall view of the table, it is obvious that the methodologies of Page-Jones and Wasserman provide only a representation, i.e. their focus is on visually representing a design and not on how to derive a particular design. For this reason, we think that these kinds of methodologies are not complete and they have been discarded from our choices.

Observation on OOA Process

Analyzing the first section of the table, we can see that most of the problem domain analysis components have been addressed by (at least) several authors. The identification of semantic classes (problem domain) and behavior has been extensively covered. However, it is interesting to note that identifying attributes has not been addressed as often as behaviour. A reason for this choice is that an object can be seen as an encapsulation of both structure and behaviour which can be accessed only via its interface. We agree with this point of view, even if we think that, at a conceptual level, the characteristics (attributes) are an important integral aspect of an object's semantic definition too. So we think that attribute names and any internal object should be specifiable.

Observation on OOD Process

OOD involves the same processes used during analysis, applied to different kinds of objects. Some authors address the base/utility classes that usually are new abstract data types which may be identified during design and added to the system library. Instead, surprisingly, interface objects are rarely addressed although we think that the user interface should be a part of any software analysis and design methodology. Another important point is the activity of refining classes. In fact, this involves examining the class structure for opportunities to abstract common behavior and attributes and find better hierarchies.

Observation on Representation

One of the first things to note is the large number of static representation models even if the used constructs and how they are emphasized vary greatly. Moreover, there are more O.O. static models than O.O. dynamic models most of which consist of state

Comparison of OOAD Representations and Processes

	PROCESS AND REPRESENTATION							REPRESENTATION ONLY	
	Bailin	**Booch**	**Coad & Yourdon**	**Rum-baugh**	**Shlaer and Mellor**	**Wirfs-Brock**	**BON**	**Page-Jones**	**Wasser-man**
1. OOA PROCESS Problem Domain Analysis (a) Identification of:									
Semantic classes	●	●	●	●	●	●	●		
Attributes		●	●	●	●				
Behaviour	●	●	●	●	●	●	●		
Relationships:									
Generalization		●	●	●	●	●	●		
Aggregation		●	●	●		●	●		
Other		●		●	●	●	●		
(b) Placement of:									
Classes			●			●			
Attributes			●						
Behaviour	●	●		●		●	●		
(c) Specification of:									
Dynamic behaviour		●		●	●				
(i.e., message passing)									
2. OOD PROCESS Solution Domain Design (a) Identification of:									
Interface classes			●				●		
Base/Utility classes				●			●		
(b) Optimization of classes		●	●	●		●	●		
3. REPRESENTATIONS (a) Static View									
Objects	●	●	●	●	●	●	●	●	●
Attributes		●	●	●	●			●	●
Behaviour	●	●	●	●	●	●	●	●	●
Relationships:									
Generalization		●	●	●	●	●	●	●	●
Aggregation		●	●	●			●		
Other		●		●	●		●		●
(b) Dynamic View									
Communication		●	●			●	●	●	●
Control/Timing		●	●	●			●		●
(c) Constraints									
On structures		●	●	●	●				●
On dynamic behaviour		●		●	●				
4. COMPLEXITY MGT.									
(a) For structural complexity		●	●	●	●	●	●		
(b) For behavioral complexity						●			
(c) Representation of:									
Static structure		●	●				●		
Dynamic behaviour						●			
No. of issues Addressed	5	21	19	20	14	15	18	5	8

Figure 4.1:

transition diagrams. Another important point regarding notation is the representation of relationships. Most representations support generalization relationships. Aggregation is supported but with a lesser extend, while other relationships are mentioned but rarely well supported by representation or accepted by many authors.

Observation on Complexity

Relatively few authors provide a conceptual grouping mechanism to manage the complexity of large design, but they do not provide many heuristics for identifying subsystem. Subjects and subsystems are supported by Coad and Yourdon's and Wirfs-Brock's representations while BON (Better Object Notation) offers clusters which allow to build this feature. Rumbaugh discusses modules as a way of grouping classes and relationships. However, he does not provide a notation for showing the module level. A solution for this problem may be the definition of various "abstraction layers" of the system. A model that synthesizes the static and dynamic features of a system at various levels of abstraction may be the most important need in OOAD research. For example, Rumbaugh and Shlaer & Mellor include object-diagrams, data-flow diagrams, and state transition diagrams. These diagrams, all together, represent three different important views of the system.

Last Evaluation of the Table

The last line of the table shows how many issues are addressed by the treated methodologies. The results show that the methodologies of Coad & Yourdon, Shlaer & Mellor, Booch, and Rumbaugh are within the most complete. Also the Wirfs-Brock's and BON methodologies have got a considerable number of points; despite of this, they are not so popular. A drawback for the Wirfs-Brock methodology is the lack in its graphical and textual representation of associations, aggregation and attributes. In addition attributes are not addressed also in the OOAD processes. We think, in fact, that in reality this method puts more emphasis on Analysis than on Design. Therefore, since this methodology is able only for getting an idea about the problem, no tools have been developed to support the designing phase and this restricts its usage. Instead, we deem that the principal reason of the unpopularity of BON is only due to the fact that this methodology is really new. It has the advantage of clusters, that allow to build subsystems. But though it supports dynamic models, they are too weak because of their little expressiveness. Another characteristic we consider an important lack is that it does not distinguish between attributes and methods that are all called *features*. Moreover, studying this methodology we have seen that it supports features, as interfaces, invariants and assertions. This makes BON very close to the Eiffel language but not useful to be mapped to a language as Beta which does not supports these concepts.

Below the methodologies of Coad & Yourdon, Shlaer & Mellor, Grady Booch and Rumbaugh are treated in more detail.

4.1.1 Coad & Yourdon

Yourdon's traditional structured [CY91] design methodology is a composition of design strategies, evaluation aids, and graphical documentation techniques. The methodology emphasizes that both OOA and OOD are distinct disciplines - whether applied in sequence or in some intertwined fashion. A previously developed OOA model serves as input for the OOD process, and the same notation is used for both the OOA and OOD models. The Coad & Yourdon OOA is illustrative and easy to understand for novice users and beginners. However, experienced Object Oriented practitioners may find its notation and concepts overly simplified [Hsi92]. It also lacks rigorous definitions of both concepts and advanced modeling techniques such as abstract classes that have been proposed by other methodologies. So the simplicity of this methodology is a strength but also its biggest weakness. For example, for defining associations, it does not provide a precise definition and implementation. Another deficiency is in defining object attributes. According to the context of their book, their OOA methodology allows only primitive class types for object attributes. This is similar to the standard relational database model. The OOD is a simple, readable design cookbook. The methodology is helpful to those with little or no experience in the field. It essentially shows how to subdivide the design of an application into certain components. However, the practitioner may also be left a little hungry for details.

4.1.2 Shlaer & Mellor

The first step of this methodology [SM89] is the construction of an information model (or object model) in order to identify the conceptual entities of the world. Its information model formalizes knowledge about the world in terms of objects, attributes, and relationships (or associations). The information model is produced in two forms: one is a set of textual definitions and the second is a graphical representation which provides a global view of the world to be modelled. One lack of this step is that this methodology does not support the concept of aggregation as a relationship. The second step is to model the dynamic behaviour of the conceptual entities and associations. The methodology assumes that all objects and relationships have their life cycles. Life cycles are formalized in state models with *states, events*, and *actions*. The third step is the construction of a set of *Process Models*. In this step a separate data flow diagram is constructed for each state in every state model. The data flow diagram for a state graphically depicts the actions processes associated with that state. The strength of Shlaer & Mellor OOA is its ability to provide a comprehensive dynamic model for developing critical Object Oriented software applications. However, the current methodology only supports the analysis phase and it is also weak in supporting semantic-rich applications due to its primitive object relationship modeling mechanism.

4.1.3 Booch

Booch's OOAD [Boo91] is the oldest among the discussed methodologies because of its first appearance in [Boo] where he proposed a method for using some of the features of Ada in Object Oriented style. Now in [Boo91] this methodology includes definitions for Object Oriented analysis and design concepts, object model, classes and objects, objects classification, and methods. It also defines Object Oriented modeling notations which include: Class diagrams, State transition diagrams, Object diagrams, Timing diagrams, Module diagrams and Process diagrams. Booch distinguishes three roles of objects: *Actors, Servers* and *Agents*. A fairly consistent graphical notation for the relationships between classes is proposed, which uses different types of lines to indicate use, inheritance and other relationships. Also a notation for object creation and destruction is introduced for the design phase; on the contrary, attributes and operations in classes do not have a graphical notations. The granularity of objects is one of the key design decisions. Booch recommends a form of layering so that classes are categorized into "categories" containing several related classes, similarly to the concept of "module". The dynamic behaviour of Booch's methodology is accomplished in two ways. A state transition diagram shows the dynamic behaviour of classes. The instance level dynamics shows by timing diagrams borrowed from the field of hardware design. Although Booch's OOAD methodology covers a wide spectrum of topics and concepts that are needed for Object Oriented practitioners to follow, his methodology is more descriptive than prescriptive. For example, many of its concepts and guidelines can be interpreted and practiced differently by individual users. This results in the lack of precision that many Object Oriented practitioners look for.

4.1.4 Rumbaugh's Object Modelling Technique

Rumbaugh's OOAD [RBP$^+$91] methodology proposes a set of Object Oriented concepts and a language independent graphical notation, called the Object Modeling Technique (OMT) that can be used in throughout the entire software development process. OMT supports the entire software life cycle using a full Object Oriented approach. One key feature, we found to be unique about OMT, is its formal definition of the inter-objects relationships, or object associations. OMT defines various kinds of object association semantics to the same level of the classes. This feature eliminates the ambiguity which we have found to be a common deficiency in other methodologies, and makes OMT the most precise OOAD methodology currently available. It also supports sufficient dynamic modelling capabilities based on events, states, and concurrency. OMT also clearly defines the boundary between Object Oriented analysis, design, and implementations phases. Note that other methodologies also draw boundaries between different phases, however OMT makes strong emphasis on this point. Moreover, as mentioned before, OMT uses three different kinds of diagrams (Object Model, Functional Model and Dynamic Model) which also represent three important different abstraction levels in the description of a system. The only drawback is that they are not well integrated.

4.2 Rational for Choosing OMT

This comparison between OOAD methodologies has been done in order to choose a good methodology to follow in the construction of a design which has to be translated in Beta.

During the analysis of the methodologies taken in consideration, some choices have been made very soon. For example, we have discarded those methodologies we have thought not complete, such as the Page-Jones's and Wasserman's, which consider only the representation without treating the phases of analysis and design. Other methodologies, such as Kurt's and Odell's, have been discarded because they support some freedom we consider out of the Object Oriented mentality or, straightway, because they do not support inheritance, as for Bailin's, Edward's and Gibson's methodologies. For the other methodologies the evaluation has been done on the basis of more refined characteristics. Basically, these methodologies adopt similar base Object Oriented concepts in their earlier analysis phase and almost all of them support a graphical notation for developing the two phases. However, they vary to large extend in their support of advanced object modelling concepts. Moreover, each methodology emphasizes on different target users and practitioners.

Coad & Yourdon methodology [Coa91a, Coa91b, Coa91c, Coa91d] is more suitable for novice professionals who start practicing OOAD. However as users become more experienced, they may want to reference other methodologies to gain more precision.

We also deem that Shlaer & Mellor's methodology provides a very practical solution to the world of OOAD. Although, we consider their methodology weak in supporting formalized object associations. For example, it misses support for the concept of aggregation. Moreover, another lack is that it does not support the designing phase which, as we have said, is fundamental.

Although it is within the most complete, Wirfs-Brock's methodology has been discarded because of its lack in graphical representation. We think this is a very important issue a methodology should support. Moreover, some concepts as attributes are not represented in a textual form.

By far Booch's and Rumbaugh's methodologies are the most complete. The similarities between the approaches are more striking than the differences, and both approaches complement each other. A major distinction between Booch's approach and the OMT approach is the emphasis that the second places on associations. Hence, in order to choose between this two methodologies we have thought about the language we are going to treat - Beta. This has been useful to decide which is the most suitable to be implemented in this language. Looking for the representation used by each methodology, we have seen that Booch provides means of documenting the meanings of each graphically defined class in the Object Diagram. This is realized textually by means of class templates which, in particular, allow the developer to define the interface of each class filling a private, public and protected part of the template. As we have seen, Beta neither supports encapsulation nor defines interfaces for objects as C^{++} does. So we have decided that the most suitable method for the Beta language is the OMT method since it puts both attributes and operations at the same level of access. Instead we consider Booch as the most suitable methodology for implementing C^{++} applications. Moreover, we feel that Rumbaugh's methodology contains the highest precision and completeness practitioners can follow without lost or confusion. It also supports a clear and well defined graphical notation that covers both the analysis and design phases of the software development process. The unique lacks that we have found in this methodology has been the bad supported integration between the Object, Functional, and Dynamic models and the absence of a graphical notation allowing the organization of an Object Model in *subsystems*. As we will see in the next section, the first problem turns out to be not very important for our aim because we will decide to consider only the Object Model.

4.3 OMT Basic Concepts

This section presents the OMT methodology and a language independent graphical notation for expressing Object Oriented models. We give an overview of the most important OMT concepts and their notation. These concepts constitute the guideline for the next chapter in which we give a description of the mapping from OMT to Beta. The graphical notation adopted is an extended version of the notation proposed in [Rum87] and showed in [RBP+91].

The OMT notation uses three kinds of models to describe a system [RBP+91,p.6]: Uses Three Kinds of Model

- the *Object Model* describes the static structure of the objects in a system and their relationships using a graph called *Object Diagram*;
- the *Dynamic Model* describes the interactions among objects in the system, i.e. by a state diagram it describes the aspects of the system that change on the time;
- *Functional Model* describes the data transformations of the system by a data flow diagram.

The three models are orthogonal parts of the description of a complete system, they are related but give three different viewpoints, each capturing important aspects of the system.

The Object Model is the most fundamental, however, because it is necessary to describe *what* is changing or transforming before describing *when* or *how* it changes. It provides the essential framework into which the Dynamic and Functional models can be placed. Therefore, of the three OMT models we will treat only the Object Model while we will omit the Dynamic and the Functional models. Object Model Is the Fundamental

An *Object Model* is graphically represented with object diagrams. *Object diagrams* provide a formal graphic notation for modelling objects, classes, and their relationships. An Object diagram is a graph whose nodes are *object classes* or *objects instances* (later called classes and objects respectively) and whose arcs are *relationships* among them. Classes are arranged into *hierarchies* sharing common structure and behaviour, organized in *aggregations*, and *associated* with other classes. Object diagrams are concise, easy to understand and work well in practice. Object Diagrams

The formal notation for an object diagram will be described in the following. Here only the basic concepts will be shown, while some advanced concepts will be treat in the next chapter.

4.3.1 Classes and Objects

The OMT symbol for a class is a box with the class name in bold face, while the OMT symbol for an object is a rounded box with the name of the class to which it belongs between brackets and in bold face.

Figure 4.2: Symbols for Classes and Objects

Classes define the attributes carried by each object instance and the operations that each object performs or undergoes. For this reason the class box is divided in at most three regions. The regions contain, from the top to the bottom: class name, list of attributes, and list of operations. Each attribute name can be followed by optional details such as type and default value. Each operation name may be followed by optional detail such as argument list and result type. Attributes and operations may or not may be shown; it depends on the level of detail desired [RBP+91,p.26].

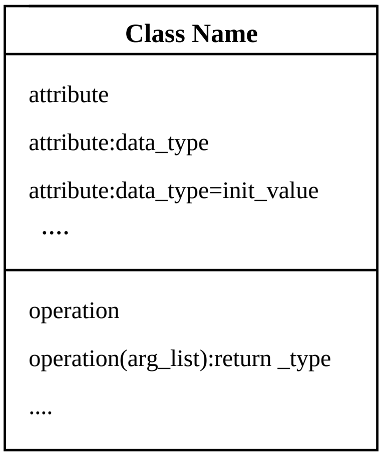


Figure 4.3: Class with Attributes and Operations

4.3.2 Relationships

Generalization

In OMT the notation for *generalization* is a triangle connecting the superclass to its subclasses. The superclass is connected by a line to the apex of the triangle and the subclasses are connected by lines to an horizontal line attached to the base of the triangle.

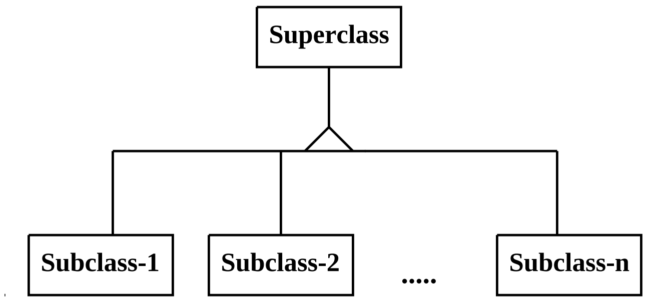


Figure 4.4: Generalization

Generalization provides the means for refining a superclass into one ore more subclasses. The superclass contains features common to all classes; the subclasses contain features specific to each class. Inheritance may occur across an arbitrary number of levels and each object accumulates features from each level of the generalization hierarchy.

Association

An *association* describes a group of links between classes with common structure and common semantics. The notion of association is certainly not a new concept. Associations have been widely used through the database modeling community for years.

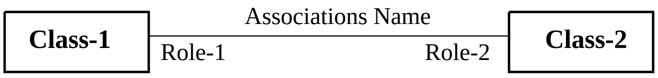


Figure 4.5: Association

Since OMT uses the natural language to specify associations, in an abstract sense, associations are inherently bidirectional and it is only the name of the association that establishes a direction.

If the association is binary, than it can be traversed in both directions and in reality both directions of traversal are equally meaningful and refer to the same underlying association. In this case usually *role names* are used to uniquely identify one end of the association [RBP$^+$91,p.34]. The use of role names provides a way of traversing the association from one object to another. Role Names

The number of related objects in an association is constrained by *Multiplicity*. Object diagram indicates multiplicity with special symbols at the end of the association line. In fig. 4.6 the various type of multiplicity supported by OMT are shown. Multiplicity

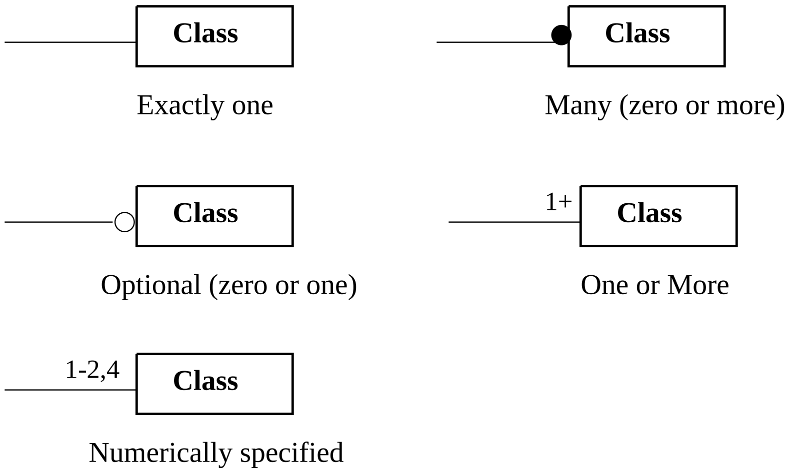


Figure 4.6: Types of Multiplicity

However, the most important multiplicity distinction is between *Exactly one* and *Many* that are also the most used.

Aggregation

The OMT notation for *aggregation* is similar to generalization but it has a diamond instead of a triangle.

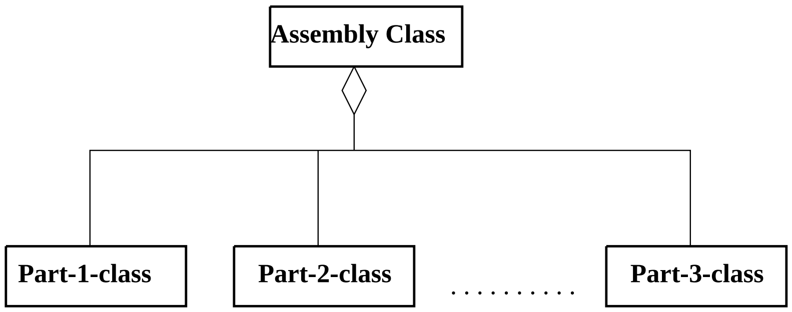


Figure 4.7: Aggregation

In reality, aggregation is a strong form of *association* between an aggregate object and its component parts with extra semantics:

- *Transitivity*: if A is part of B and B is part of C, then A is part of C.
- *Antisymmetry*: if A is part of B, then B is not part of A.

The aggregate object is semantically an extended object that is treated as a unit in many operations, although it is physically made of several lesser objects. A single aggregate object may have several parts connected with the aggregate by mean of a *part whole relationship.* Each part whole relationship is treated as a separate aggregation in order to emphasize the similarity to the association. Like association, also aggregation supports the concept of the multiplicity with the same notation. The use of multiplicity gives rise to three different kinds of aggregation:

Three Kinds of Aggregation

- FIXED: *In a fixed aggregation a fixed aggregate has a fixed structure: number and types of subparts are predefined.*
  For example, Car may be naturally viewed as consisting of parts like a "Motor", a "Body", and "Wheel", i.e. "Wheel" is a part-of a Car, "Body" is a part-of a Car, etc.
- VARIABLE: *A variable aggregate has a finite number of levels, but the number of parts may vary.*
  For example a 'Company' may be composed of many 'Division', and each Division of a Company may be composed of many 'Departments'. In this example a company is a variable aggregate with two-level-tree structure.
- RECURSIVE: *A recursive aggregate contains, directly or indirectly, an instance of the same kind of aggregate. The number of potential levels is unlimited.*
  For example a program may be composed by blocks nested in arbitrary depth.

Instantiation

Instantiation is a relation between a class and an object that is an instance of this class. This is the OMT notation:

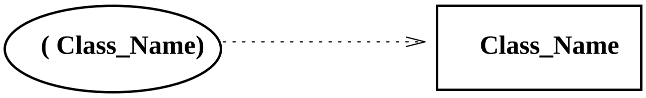


Figure 4.8: Instantiation

Usually this kind of relationship is not useful during the design of an object diagram. Relationships between objects are used in another kind of diagram, the *Instance diagram*, that we are not going to treat. Instance diagrams are useful for discussing examples. Since an infinite number of objects may belong to a class, infinite instance diagrams belong to an Object diagram.

Chapter 5

Mapping from OMT to BETA

This chapter discusses how to take a general OMT Object Model and implement it with the O.O. language Beta. Our goal is to produce a Beta skeleton for a software system corresponding to this model. The realization of this chapter would be a proposal to integrate the OMT design language with the O.O. language Beta. This part constitutes the theoretical basis for the development of our tool that we will describe in the chapters 6 and 9. The OMT concepts shown in the previous chapter will be treated and translated in Beta code. Moreover, some other concepts, in [RBP+91] called *advanced concepts*, will be introduced and translated during this "mapping". We will see that, because of the characteristics of Beta, it will not be possible to directly translate and, unfortunately, either simulate some of the OMT concepts. One example is multiple inheritance of which we give some proposals to simulate all those aspects that make it useful.

5.1 OMT Versus Beta

The following considerations apply when implementing an Object Oriented design in an O.O. language:

1. Class definition
2. Creating Objects
3. Calling Operations
4. Using Inheritance
5. Implementing Associations
6. Aggregation

This is the schema followed by this section to describe our mapping.

5.1.1 Class Definition

Object Class

An *object class* describes a group of objects with similar properties (attributes), behaviour (operations), common relationships with other objects and common semantics [RBP+91,p.22]. The first step in implementing an Object Oriented design is to declare object classes. Each attribute and operation in an Object diagram must be declared as a part of its corresponding class. Furthermore, it is a good practice to carry forward the names from the design diagram and assign data types to attributes. For example, to represent the concept of a bank account, in which you have the balance and the operations of deposit and withdraw, in OMT we can use the following class:

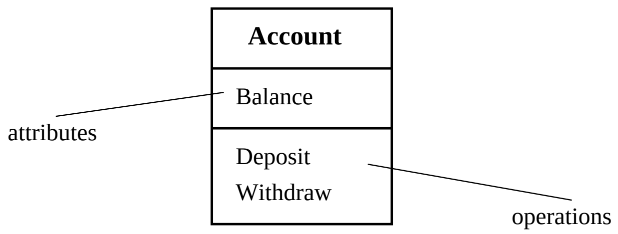


Figure 5.1: Class

Implementation with Class Pattern

In Beta the concept of class is mapped into the concept of *class pattern*. So, in Beta language the declaration of the class *Account* can be represented by the following pattern *Account* with *Balance*, *Deposit* and *Withdraw* as attributes.

```
Account: (#  Balance: ......;
             Deposit: ......;
             Withdraw: ......;
         #)
```

Now we will see in particular how to translate class attributes and operations.

Attributes

OMT Definition

An *attribute* in OMT is a data value held by the objects in a class. An attribute should be a pure datavalue, not an object, because, unlike objects, pure data-values do not have identity. Moreover, its name must be unique within this class.
In OMT an attribute is defined in the following way:

attribute_name:data_type (=data_type_value) where we assume that *data_type* may be one of the *basic pattern* included in Beta or a *repetition* of a basic pattern, and the sentence between brackets is the optional default value.

This declaration will be translated in Beta in one of the following attribute patterns: Translation with Pattern Attribute

attribute_name:@data_type

if *data_type* is a name or a Beta basic pattern or simply a name:

attribute_name:[eval]@basic_pattern_name

if *data_type* is a repetition of a basic pattern.
In this case the basic attribute will be initialized to its default value, but if in OMT the attribute is initialized to a particular default value, in Beta, in the do_part of the pattern containing the attribute, it should be an assignment of the value to the attribute. Moreover, since Beta does not support redeclaration of attributes in the same pattern, it is not possible to translate an object with more than one attribute with the same name.

5.1.2 Operations

An *operation* in OMT is a procedure or a function that may be applied to or by an object in a class. All objects in a class share the same operations. Each operation has a target object as an implicit argument and the behaviour of the operation depends on the class of its target. In fact an object knows its class, and hence the right implementation of the operation. Remember that a method is the implementation of an operation in a class and, when an operation has methods on several classes, it is important that all methods have the same signature (member and type of arguments, type of the result value). As a rule, an OMT operation has the following syntax: OMT Definition

operation_name(arg1:Type1; ...; argn:Typen):result_Type

where we assume that *Type1, ...Typen*, are names of Beta basic patterns, repetitions or a name of a structured data type and the result type may be a list. We also assume that *Type1, ..., Typen* may be pointers to a particular object class. This is useful for the user when he wants to declare operations to handle associations (see section *Association*). The result type should not to be omitted, because it is important to distinguish operations that return values from those that do not. Operations can be translated in Beta by means of procedure or functional pattern.

Suppose to have an OMT operation that does not return values like the one shown below: Translation with Procedure Pattern

operation_name(arg1:Type1, ... argn:Typen)

in Beta this operation can be translated in the following *procedure pattern*:

```
operation_name:
        (#  arg1:Type1; ... argn:Typen;
            ...{other useful declarations}...
            enter(arg1,..., argn) {enter-part}
            do
            ...
            {do-part}
            ...
        #)
```

From the modelling point of view, a *procedure pattern* is used for generating an action sequence that implements the method of the operation. To represent temporary a state information during this action sequence, an instance of this procedure is generated. (See also section *Calling Operations*).

Translation with Functional Pattern

On the contrary, if an OMT object has an operation that returns values:

operation_name(arg1:Type1, ..., argn:Typen):result_Type1, ..., result_Typem

in Beta this can be translated in the following *functional pattern*:

```
operation_name:
        (#  arg1:Type1; ...; argn:Typen;
            result1:result_Type1; ...; resultm:result_Typem:
            ...{other useful declarations} ...
            enter(arg1, ..., argn)
            do
            ...
            exit(result1,..., resultm) {exit-part}
        #)
```

A *functional pattern* means a pattern intended for computing a list of values on the basis of a set of input parameters. The result values are computed in the *do-part* of the object descriptor which implements the method of the operation. In addition the computation of the value should not change the state of any other object. Unfortunately Beta does not support encapsulation and, since the method of an operation is independent from the design, it is not possible to prevents the user to produce these side effects.

5.1.3 Derived Attributes (Advanced Concept)

During modelling it is useful to distinguish operations that have side effects from those that merely compute a functional value without modifying any objects. The later, in OMT, is called *query*. Queries with no arguments except the target object may be regarded as *derived attributes*. A derived attribute is like an attribute in the way that it is a property of the object itself, and when computed it does not change the state of the object. An Object Model should generally distinguish "independent" basic attributes from "dependent" derived attributes. The choice of basic attributes is arbitrary but should be made to avoid the overspecification of the state of the object. Definition:

For instance if you have two classes, *Person* and *Current Date*, as shown in fig. 5.2 , age provides a good example of a derived attribute. Example

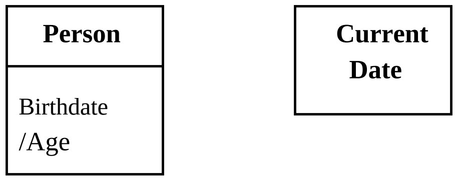


Figure 5.2: Derived Attribute (Age = CurrentDate - Birthdate)

Age can be derived from birthdate and the current date. As you can see, the OMT notation of a derived attribute is a slash, or a diagonal line, in front of the name of the attribute.

The concept of derived attribute may be compared with the Beta concept of *measurable property*. A measurable property is a property of a class that does not have substance in the real or imaginary part of the world being modelled. For this reason it is better to implement it as a pattern which represents the measurement of the property rather than directly as a basic pattern. Because of this consideration, the derived attribute *Age* may be implemented with the following functional pattern: Implementation

```
Age:  (#  V:@integer;
          do
          {here the age is computed on the basis of
          the Birthdate attribute and Current date values}
          exit V
      #)
```

5.1.4 Creating Objects

Definition

In the previous section we have seen how a class defines the interface for a set of objects. These are called *instances of the class* and usually can be created dynamically by a "new" or "create" operation. The OMT notation for an object appears in an Object Model together with the concept of instantiation. Although classes and instances can appear in the same OMT diagram, in general it is not useful to mix classes and instances. However, we think that it is important to show how to generate objects corresponding to class objects.

Two Way for Creating Objects

In Beta there are two different ways to generate objects. An object may be created statically by the declaration of a static reference, or created dynamically by a "new" imperative. These different ways of creating objects give rise to two categories of objects respectively called *static* or *dynamic* objects.

Static Objects

A *static object* (or part object) is generated by a declaration of static reference like:

```
X:@T
```

where X is the name of the reference and T is a pattern; or by a *singular* static/part object which is declared in the following way:

```
Y:@(# ...... #)
```

The OMT concept of instantiation of a class (T) corresponds to the declaration of a static object (as shown before) made at the same level of the declaration of the classes composing the Object Model. We will see later that static objects are also useful in modelling part hierarchies, i.e. objects which consist of part objects.

Dynamic Objects

It is possible to create objects dynamically by the execution of actions. The following evaluation creates an instance of the T pattern and the result of the evaluation is a reference to the newly created object (X).

```
&T[]->X[]
```

Dynamic generation of objects is used to describe systems where new objects are generated during program execution, as it is often the case when modeling real life phenomena. For example, from a technical point of view recursive data structures give rise to the dynamic generation of objects. For this reason objects that are created dynamically are not shown in an OMT diagram.

5.1.5 Calling Operations

We have seen that objects which are instances of the same class have the same behaviour because of the same operations. So, after having seen how to create an object, in this section we will see how it is possible to "generate" its behaviour.

How to Call an Operation in Beta

In Beta, all OMT operations are methods associated to a pattern and, as we have seen, in Beta all arguments and variables of operations are objects. In the *do-part* of an object we can call an operation invoking the pattern that implements the operation as a procedure or functional pattern by the evaluation of:

&T

where T is the name of the pattern that implements the operation; or generating a *singular object* by the execution of

&(# ....#)

In this way, the invocation of a pattern as a function or a procedure gives rise to the generation of an object for representing the action sequence being generated by executing the procedure/function.

Inserted Item for Storage Management System

These ways of "calling operations" lead to the generation of a large number of small objects. These objects have to be generated and removed by the storage management system, which may be quite expensive. For this reason, it is possible to declare that such procedure/functional objects are generated as a permanent part of the object invoking the pattern. This can be done by the generation of an *inserted item* in one of the following ways:

E− > T− >A

E1− >(#.....#)− >A

The T object (the inserted object) will be an integral part of the enclosing object. This inserted item will then be executed when control reaches the evaluation statement. The state of T will be undefined before each execution. Apart from the allocation, the execution of the inserted item is like the execution of any other object. The motivation of inserted items is a matter of efficiency, since the compiler may compute the storage requirement of the calling object. Inserted objects are similar to static objects in the sense that they are allocated as part of the enclosing object and they cannot be used to describe recursive procedures, since this will lead to an infinite recursion. (Remember that static items cannot be used for describing recursive data structures).

As we have said in the first section, an object class also describes the relationships with the other objects. In the next sections we will see the implementation of the most important relationships, such as *inheritance*, *association* and *aggregation*. These are important because they show how objects can communicate.

5.1.6 Inheritance

As we have said in chapter 2, Beta classifies inheritance in *static*, *implicit* and *per group*. Also OMT has this kind of view of inheritance. In fact, in OMT the developer draws the Object Model during the design of the system he wants to develop, and the features that are inherited or overridden are also part of the Object Model. Moreover, an Object Model has classes as nodes in the part that describes inheritance. In the following, we will use the words *generalization* or *specialization* to refer to the relationships among classes, and the word *inheritance* to refer to the mechanism of sharing attributes and operations using the relationship given above.

Example

We would like to be able to model situations that you could find in a travel agency that handles reservations of several kinds including flight and train reservations. The common properties and the way in which these reservations differ give rises to a simple classification hierarchy. Fig. 5.3 shows the situation described above.

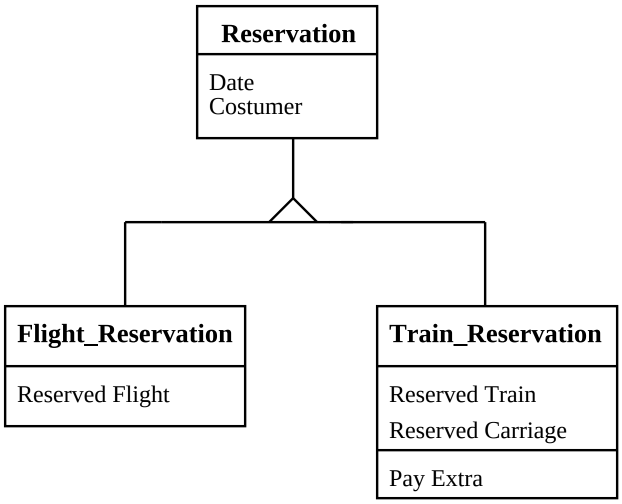


Figure 5.3: Inheritance

Remember that in OMT generalization and inheritance are transitive across an arbitrary number of levels. Each subclass not only inherits all the features of its ancestor but adds its own specific attributes and operations.

Translation with Subpatterns and Virtual Patterns

Generalization and specialization are directly supported in Beta by *subpatterns* and *virtual patterns*. The example described in fig. 5.3 can be translated in Beta in the following patterns and subpatterns:

```
Reservation:                     (#   Date: ...;
                                      Customer: ...;
                                 #);

FlightReservation: Reservation   (#   ReservedFlight: ....;
                                 #);

TrainReservation: Reservation    (#   ReservedTrain: ...;
                                      ReservedCarriage: ...;
                                      PayExtra: ...;
                                 #)
```

Here *TrainReservation* and *FlightReservation* are subpatterns of *Reservation* and *Reservation* is the superpattern. (Note that in both OMT and Beta inherited attributes need not to be repeated). The objects created as instances of the three patterns will form three disjoint sets (have disjoint extensions). Subpattern features can be accessed from any subclass but features from any subpatterns cannot be accessed from the superclass.

Overriding Operations

Moreover, OMT allows subclass to override a superclass feature by defining features with the same name in the subclass. You may override methods of operations and default values of attributes but you should never override the signature or form of the features. Inherited features can be renamed in a restriction. Suppose that in our travel agency we want to print the attributes of the reservations. In OMT this could be represented in the following way:

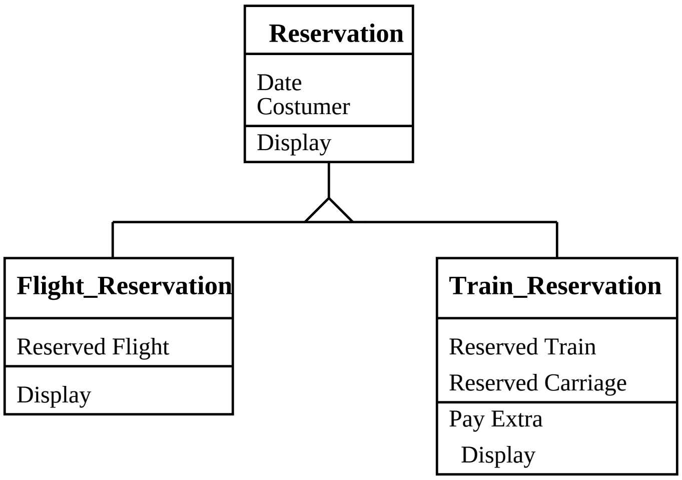


Figure 5.4: Overriding the Operation Display

where the operation *Display* in *Reservation* displays the attributes *Date* and *Customer* while *Display* in *FlightReservation* specializes the namesake operation of *Reservation* displaying its proper attributes. In Beta, if a method is overridden by a subpattern, the pattern representing the operation must be declared as *virtual pattern* in its first appearance in a superpattern. The subpattern and superpattern's action-parts are combined using the *INNER* statement.
The example in fig. 5.4 can be translated as follow:

```
Reservation:
    (#...{attributes};
       Display:< (#  enter  (...)
                     do
                       {display Date and Costumer}
                       INNER
                     exit  (...)
                 #)
     #);

TrainReservation  :  Reservation
     (# ...{attributes}
        Display  ::< (#  do
                           {display ReservedTrain
                           and ReservedCarriage}
                     #)
     #);

FlightReservation:  Reservation
     (# ...{attributes}
        Display  ::<  (#  do
                             {display ReservedFlight}
                      #)
     #);
```

Figure 5.5: Implementation of the Operation Display

As for attributes, operations declared in the superclass are also inherited. Methods that override inherited methods must be declared as binding in the subclass. Inherited (and not overridden) attributes need not to be repeated. To ensure that a signature of an operation cannot be overridden, the enter and the exit part (if exist) of the operation should be defined only in the first appearance of the virtual pattern implementing the operation. The way in which the INNER works ensures that the user can override only the methods while he has the same enter and exit part for all the specialized operations.

Since Beta does not support pattern renomination, it is not possible to override the default value of attributes; instead, if in a superclass you have an attribute A, that you want to rename with B in a specific subclass, in Beta you can have the following code:

Override Default Attribute Value and Attribute Renomination

```
Superclass:        (#   A: ...;
                        ...;
                   #);

Subclass:Superclass
                   (#   B:@(# enter A exit A #);
                        ...;
                   #)
```

Renomination of operations is allowed using patterns and subpatterns in the following way:

Renomination of Operations

operation:(# ... #);

new_name:operation (# ...#)

Where *operation* is declared in the superclass and *new_name* in the subclass.

The subpattern mechanism only supports tree-structured classification hierarchies. Therefore Beta does not support *multiple inheritance*, i.e. it does not support the possibility of a class to have more than one superclass. We have tried to simulate multiple inheritance with others Beta concepts, in particular using part objects, but we have been able to treat only the principal aspects of it without finding a unique simulation able to capture all of these. The various aspects we have seen are:

Multiple Inheritance

- How to have more than one classification hierarchy for the same class of objects;
- Overriding inherited operations (multiple override criteria);
- Code reuse;

This argument will be explained in more detail in section 5.2 (Multiple Inheritance).

5.1.7 Implementing Associations

Associations are the "glue" of our Object Model, providing access paths between objects. An association describes a group of links between classes with common structure and common semantics. In this section we formulate a strategy for implementing associations of an Object Oriented Model in Beta language. An association is a logical construct of which a pointer is an alternative implementation. Most existing Object Oriented programming languages ([Cox86], [GR83], and [Mey88]) lack the notion of association and require the use of pointers. Associations are also translated by distinct association objects. Since Beta does not explicitly support associations, we had to implement them for some kinds of association. Moreover, whichever implementation strategy you choose, a good translation should hide the implementation using access operations to traverse and update the associations. We have implemented associations trying to satisfy the constraints, but we will see that for some associations, that have been implemented with single pointers, it has not been possible. In this section the various implementations will be classified on the basis of the number of instances involved in each instance of the relationship [LMN93,p.310].

One_to_One Association

Definition: *In a One_to_One association, in each instance of the relationship at most one instance of a class can be in relation with at most one of another.*

This kind of association can be a *one way* or *two way* association, depending on the way in which the association is traversed.

One Way Association

Example

In OMT an example of a one way association can be the following:

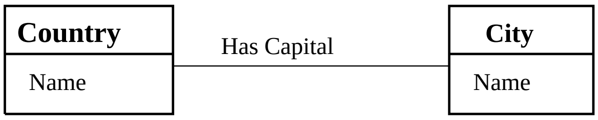


Figure 5.6: One Way Association

Each country has a capital city so that you have a "reference" from the class *Country* to the class *City*. This kind of association can be implemented as a simple pointer - an attribute that contains an object reference.

Implementation

In Beta the pointer can be implemented by a *dynamic reference attribute*. The previous example could be translated as follow:

Country:

```
        (#   Name: ...;
             Has_Capital:^City;
        #);
```

City:

```
        (#   Name: ...;
        #)
```

Two Way Association

In practice many associations are traversal in both directions. An OMT example for this kind of association is:

Example

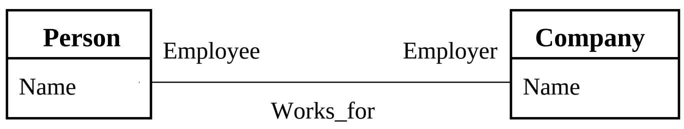


Figure 5.7: Two Way Association

In this relation, a person assumes the *role* of employee of a company and a company assumes the role of employer. These role names may be useful in Beta translation. In fact, they can be used as the names of the references that implement the pointers in both directions.

Use of Role Names

The example in fig. 5.7 can be translated in Beta in the following way:

Person:

```
        (#   Name: ...;
             Employee:^Company;
        #);
```

Company:

```
        (#   Name: ...;
             Employer:^Person;
        #)
```

Since role names are used to distinguish among objects directly connected to a given object, all role names must be unique [RBP+,p.35]. Although the role name is written next to the destination object on the association link, it is really a derived attribute of the source class and unique within it. Since we translate role names in pattern names, it is impossible to use more than one role with the same name because Beta does not support multiple declaration of attributes inside a pattern. If role names are not used, then the name of the association may be used as the names of the references implementing the association instead of roles.

Inconsistent Links

This approach in implementing associations allows fast access but, unfortunately, associations cannot be simulated by attributes on classes without violating encapsulation of classes because the paired attributes composing the association are not independent. Updating one pointer in the implementation of one association implies that the other pointer must be updated as well to keep the link consistent. The individual attributes should not be made freely available externally because they must not be updated separately. C^{++} allows limited relaxation of encapsulation using the *friend* construct, and Eiffel provides export of features to selected classes but, for example, there are no clean ways to encapsulate associations as attributes in Smalltalk as well. Unfortunately, Beta does not support this kind of features too. For this reason, and because of other characteristics of the language, it is not possible to find a way to ensure this consistency. Note, however, that this problem is not a real problem for our final purpose because we will be able to ensure the consistency of the link at the level of the syntax tool.

One_to_Many Association

Definition: *In this kind of association, in each instance of the relationship at most one instance of a class can be in relation with many instances of the other.*

For example, the class *Company* of the previous example in reality assumes the role of employer for many persons. This new situation could be the following:

Figure 5.8: One_to_Many Association

Now a pointer from *Company* to *Person* is not sufficient. We need a set of pointers. A set may be implemented using an appropriate available data structure - often a linked list or an array. An hash table or a binary tree may be used as well for greater efficiency.

Implementation Using Set

The two way relations can be seen as in fig. 5.9 that suggests a Beta translation using *part object*:

```
Person:
        (#   Name: ..;
             Employer:^Company;
        #);

Company:
        (#   Name: ...;
             Employees:@Set(# Element::Person#);
        #)
```

where *Set* is a pattern defined in the Beta basic libraries. The pattern *Set* also defines the operations that can be used to handle the association.

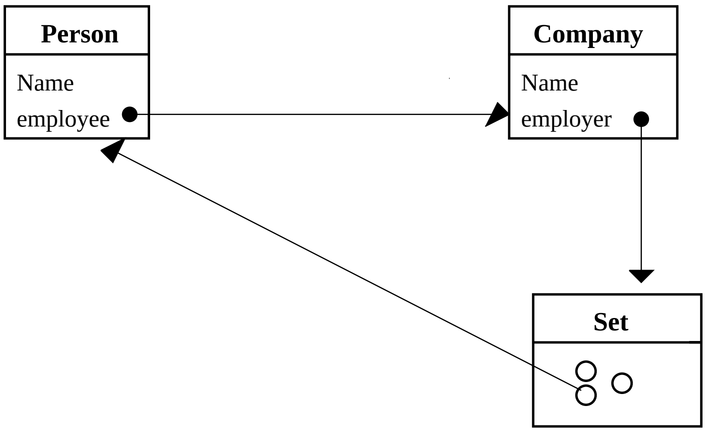


Figure 5.9: How to See One_to_Many Association

Implementation Using Repetition

The translation can also be done using *repetition* of references (see fig 5.9a). In practice this representation may be inconvenient because the dimension of the data structure is fixed.

```
Company:
        (#   Name:...;
             Employees:[...]^Person;
        #)
```

Figure 5.9a

Fixed Multiplicity

Instead, repetition of references may be convenient to represent One_to_Many relation in which the multiplicity is fixed. For example, if the company engages only seven persons, the example is the following:

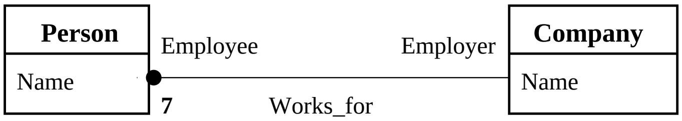


Figure 5.10: Fixed Multiplicity

Company:
```
(#   Name:...;
     Employees:[7]^Person;
#)
```

Figure 5.10a

Association Fixed and Ordered

An advanced concept of the OMT allows the objects on the many side of the association to have an explicit order that must be preserved (See the example in fig. 5.11 and the corresponding Beta translation in fig. 5.11a).

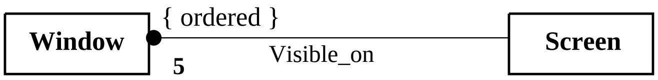


Figure 5.11: Ordered Association

Window:
```
(#   Visible_on:^Screen;
#);
```

Screen:
```
(#   Visible_on:[5]^Window;
#);
```

Figure 5.11a

This example shows a workstation screen containing five overlapping windows. The windows are exactly ordered, so that only the topmost window is visible at any point of the screen.

Association Ordered and Not Fixed

If the multiplicity is not fixed, repetitions may be used but, to hide the implementation of the association, some operations on repetition must be redefined to support the expandibility of the structure. Otherwise, the association can be implemented with a linked list using the structure *List* predefined in the Beta basic library.

Many_to_Many Association

Defition: *In a Many_to_Many association, in each instance of the relationship, many instances of a class can be in relation with many instances of the other.*

For example, a Many_to_Many association may be useful to represent a relation between files and users. The OMT notation for this kind of relation may be the following: Example

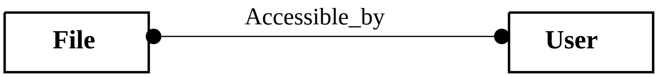


Figure 5.12: Many_to_Many Association

One file is accessible by a lot of users and a user can access a certain number of files. This Many_to_Many association can be seen in fig. 5.13.

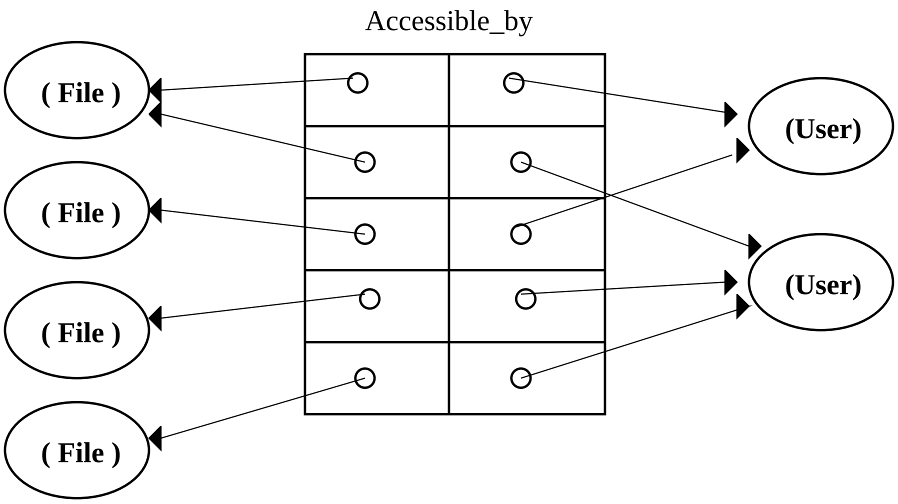


Figure 5.13: View of Many_to_Many Association

Here, the association is a set of pair of associated objects stored in a single variable size object. For efficiency, this can be implemented using two dictionary objects, one for the forward direction and one for the backward direction. Access is slightly slower than with attributes pointers, but if an hash table or a binary tree is used, then access is still constant time.

Although Hash Table is defined in the Beta basic libraries, we have decided to implement the structure shown in fig. 5.13 with a binary tree (called *Tree*) in which each node represents a link between two objects. As for the pattern *Set*, we have also implemented some operations on the structure that can be used by the user to handle the association. Implementing the association with a binary tree means to have a vari- Implementation able size structure on which you can define efficient operations. By using hash table it would have been more complex to define and implement the operations needed by the developer. In Beta the implementation of the association shown in fig. 5.12 is a pointer to the structure *Tree* defined in the patterns *File* and *User* as shown below:

```
File:  (#  Accessible_by:^Tree;
           ...;
       #);

User:  (#  Accessible_by:^Tree;
           ...;
       #);
```

and in the main program, an instance of the binary tree must be declared and specialized with the type of objects involved in the association:

```
Accessible_by:@Tree  (#  Type1:<File;
                         Type2:<User;
                     #);
```

With this implementation, an instance of a link is an instance of the class implemented by mean of the Beta pattern *Tree*. This kind of association, like the One_to_Many, has been modelled as a class.

Higher Order Association

Associations may be binary, ternary or of higher order, but in practice the most used are binary associations, only few ternary association are exploited and higher order are never needed [RPB+91,p.28]. Moreover you have to beware of ternary associations that sometimes can be restated as two binary association.

Example

An example of *ternary association* is the following:

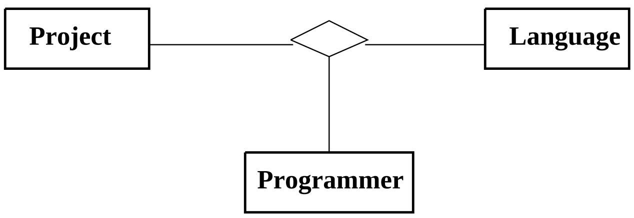


Figure 5.14: Ternary Association

In this association, programmers use computer languages on projects. This association is an atomic unit and cannot be subdivided in two binary associations without losing information.

We have decided to implement ternary associations as the Many_to_Many association, i.e. we have used a binary tree where each node has three pointers, instead of two, to the related objects (see fig. 5.15). This kind of binary tree is called *Ternary*. Implementation

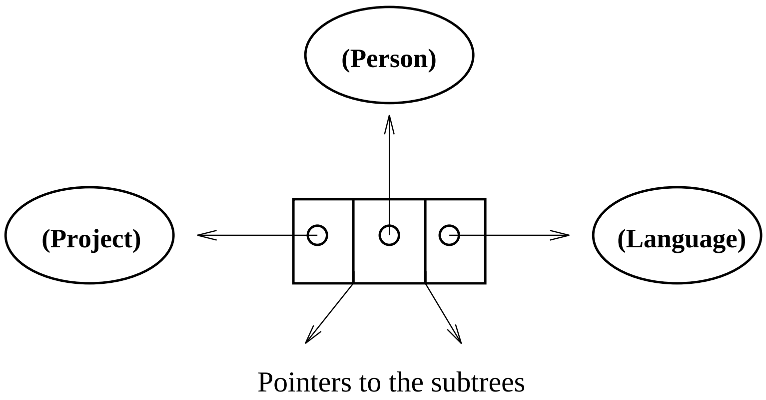


Figure 5.15: Node in Ternary Association

The declaration that must be generated in the Beta code is almost the same: you must declare a pointer to the structure *Ternary* in each class involved in the association. The link is an instance of the structure *Ternary* specialized with the three kinds of objects involved in the association; in our case:

```
Ternary_ass:@Ternary   (#   Type1:<Project;
                            Type2:<Person;
                            Type3:<Language
                        #);
```

Link Attributes (Advanced Concept)

Sometimes it is useful to use *Link Attributes*. As an attribute is a property of an object in a class, a Link Attribute is a property of the link in an association.

In fig. 5.16, for example, an OMT example is shown in which a Link Attribute is used: *Access Permission* is an attribute of the association *Accessible_by*. Example

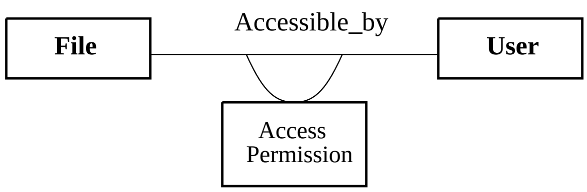


Figure 5.16: Link Attributes

The OMT notation and the fact that each Link Attribute has a value for each link, emphasizes the analogies between attributes in an object and Link Attributes.

Implementation

A Many_to_Many association provides the most compelling rational for Link Attributes. Such attribute is unmistakably a property of the link and cannot be attached to either object without losing information. For this reason, the implementation of associations with attributes depends on the multiplicity. If the association is One_to_One, the Link Attributes can be stored as simple attributes of either object. If the association is One_to_Many, the link attributes can be stored as attributes of the object at the many-end, since each "many" object appears only once in the association. If the association is Many_to_Many, the best approach is usually to implement the association as a distinct class, in which each instance represents one link and its attributes.

Many_to_Many Association

For this reason Link Attributes in Many_to_Many associations have been implemented declaring a class *Attributes* that contains all the Link Attributes in the tree implementing the association. A pointer to an instance of this class is inserted in each node of the tree. In this way, each node represents a link between two objects with its attributes. Fig. 5.7 shows how a node can be seen and the Beta code that must be generated to translate the association shown in fig. 5.16.

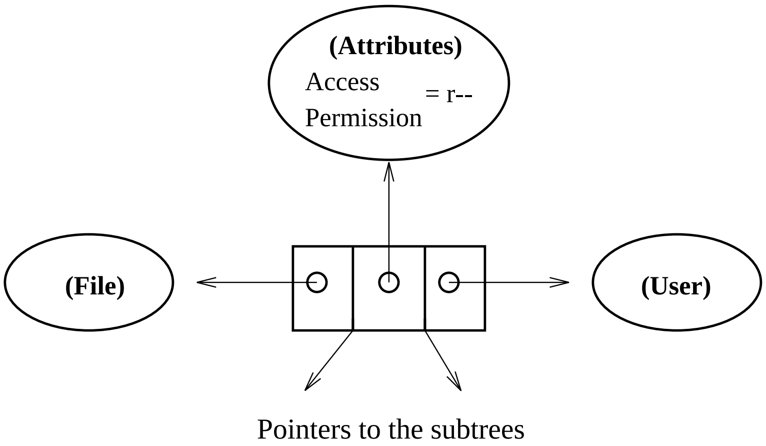


Figure 5.17: Node for Link Attributes

```
File:   (#   Accessible_by:^Tree;
             ...;
        #);

User:   (#   Accessible_by:^Tree;
             ...;
        #);
```

```
Accessible_by:@Tree   (#   Type1:<File;
                           Type2:<User;
                           Attr:<Attributes
                      #);

Attributes:   (#   AccessPermission: ...; #);
```

We have also implemented the operations that allow the user to get all the Link Attributes. We would have liked to allow the user to access a single Link Attribute, but this feature cannot be allowed without showing him the internal representation of the association.

One_to_One and One_to_Many Association

For a One_to_One association or a One_to_Many association, the translation is obvious. It is sufficient to add the Link Attributes as simple pattern attributes in the appropriate class pattern/s that implement the related objects, as explained above. Unfortunately, with this kind of implementation, in the binary association the consistency of the link attributes is not ensured. So it will be a task of the user to pay attention to this kind of inconvenients.

5.1.8 Aggregation

Aggregation is a strong form of association in which an *aggregate* object is made of *components*. Components are *part of* the aggregate. Aggregation is inherently transitive and an aggregate may have parts which in turn may have parts. In OMT, this gives rise to an aggregation tree composed of object instances that are all parts of the composite object. In Beta, aggregation is supported by *composition*. We think that, of the many ways for making composition, aggregation is directly supported by *Whole-Part composition* while *Reference composition* shows that aggregation is really a special form of association and not an independent concept. In OMT, aggregation may be *fixed*, *variable*, or *recursive*. In this section we show how the various types of aggregation should be implemented. At the end we will see that these implementations give rise to some problems.

Fixed Aggregation

Example

As we have seen in the last chapter, an example of fixed aggregation can be a car. A car may be naturally viewed as consisting of parts like a *Motor*, a *Body*, and *Wheel*, i.e. *Wheel* is a part-of a *Car*, *Body* is a part-of a *Car* etc. The following figures show how this example can be expressed in the OMT notation and how this example is translated in Beta:

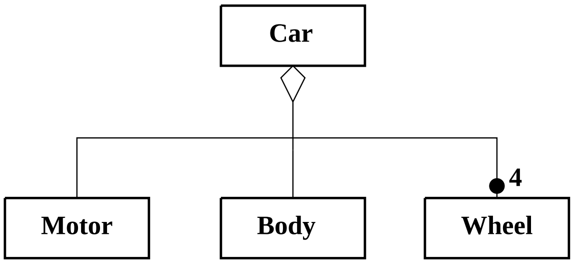


Figure 5.18: Fixed Aggregation

Implementation

```
Car:  (#   CarMotor:@Motor;
           CarBody:@Boby;
           Wheels:[4]@Wheel;
      #);

Motor:  (# ... #);
Body:   (# ... #);
Wheel:  (# ... #);
```

Declaring the component objects as *part_object* inside the pattern representing the aggregate you can generate the instances of all the parts of the composed object that compose the aggregation tree.

Variable Aggregation

In the OMT example in fig. 5.19, *Company* is a variable aggregate with a two-level-tree structure.

Example

Figure 5.19: Variable Aggregation

There are many divisions per company and many departments per division. This example may be implemented in Beta in the following way:

Implementation

```
Company:  (# Composed_of:@Set1(# element::Division#);
             ...
          #);
```

Division: (# Composed_of:@Set1(# element::Department#);
...
#);

Department: (# ... #)

Where *Set1* is a structure similar to set which implements a set of instances instead of a set of references (pointers). This example and the example in fig. 5.18, show that some kinds of aggregations are implemented with the same idea of One_to_Many association. This emphasizes the similarity between these relations. This kind of implementation of fixed and variable aggregations maintains the most significant property of aggregation that is the transitivity. Aggregation is also antisymmetric and this property is preserved by the implementation.

Recursive Aggregation

We have seen that in a recursive aggregation, the number of potential levels is unlimited.

Example

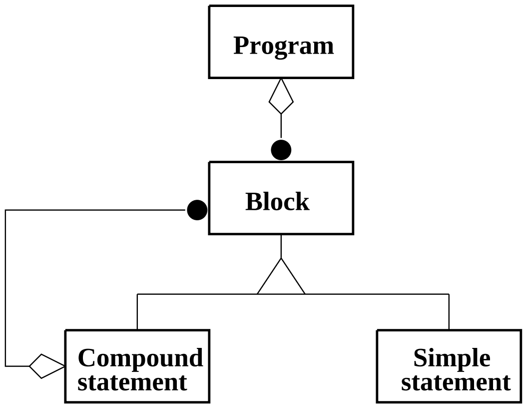


Figure 5.20: Recursive Aggregation

Fig. 5.20 shows an example of recursive aggregation in which a computer program is composed of blocks that can be nested in arbitrary depth.

The Beta translation of this example is the following:

Implementation

Program: (# Composed_of:@Set1(# element::Block #);
...
#);

Block: (# ... #);

CompoundStatement: Block
(# Composed_of:@Set1(# element::Block #);
...
#);

SimpleStatement: Block (# ... #)

PROBLEMS:

As we said before, we have found some problems in implementing aggregation.

Repetition of Instances

We have implemented fixed aggregation by means of repetition of instances. This implementation should work, but up to now, repetition of instances of non-basic patterns has not been implemented yet. We plan to solve this problem by declaring an attribute for each instance needed. The number of the attributes is defined by the multiplicity of the aggregation.

Set of Instances

In implementing variable aggregation, we used set of instances similarly to the set of references available in the Beta basic libraries. We are not able to implement this kind of set and we think that it is not possible. In fact, because of some limitations of Beta, it is not possible to handle instances using statements similar to those that are defined for references. This lack does not allow an implementation of a set of instances similar to the set of references. However, we do not want to use set of pointers even if aggregation is a particular kind of association and in Beta composition may be viewed also as Reference composition. We think that using pointers a real association is realized losing the additional meaning of the part-of concept and some properties of the aggregation like, for example, transitivity. For these reasons we have decided that is not possible to translate variable aggregation in Beta.

Propagation of Operations (Advantced Concept)

Definition

Another advanced concept of the OMT notation is the *propagation of operations*. Propagation is the automatic application of an operation to a network of objects when the operation is applied to some starting object. For example, moving an aggregate means to move all its parts; the *Move* operation propagates from the aggregate to the parts. Moreover, propagation of operations to parts is often a good indicator of aggregation. Fig. 5.21 shows an OMT example of propagation.

Example

A Man is composed of a *Body*, two *Legs*, two *Arms* and a *Head* which in turn are composed of other objects. If you move a man, you move every significant part of it. As we can see, in the OMT notation the names of the propagated operations are written near an arrow up to the classes in which they are propagated. The *Move* operation can be implemented by invoking a corresponding *Move* operation in the parts.

A Beta implementation of the *Move* operations in the aggregate *Man* (fig. 5.21) is shown in the figg. 5.22 and 5.23.

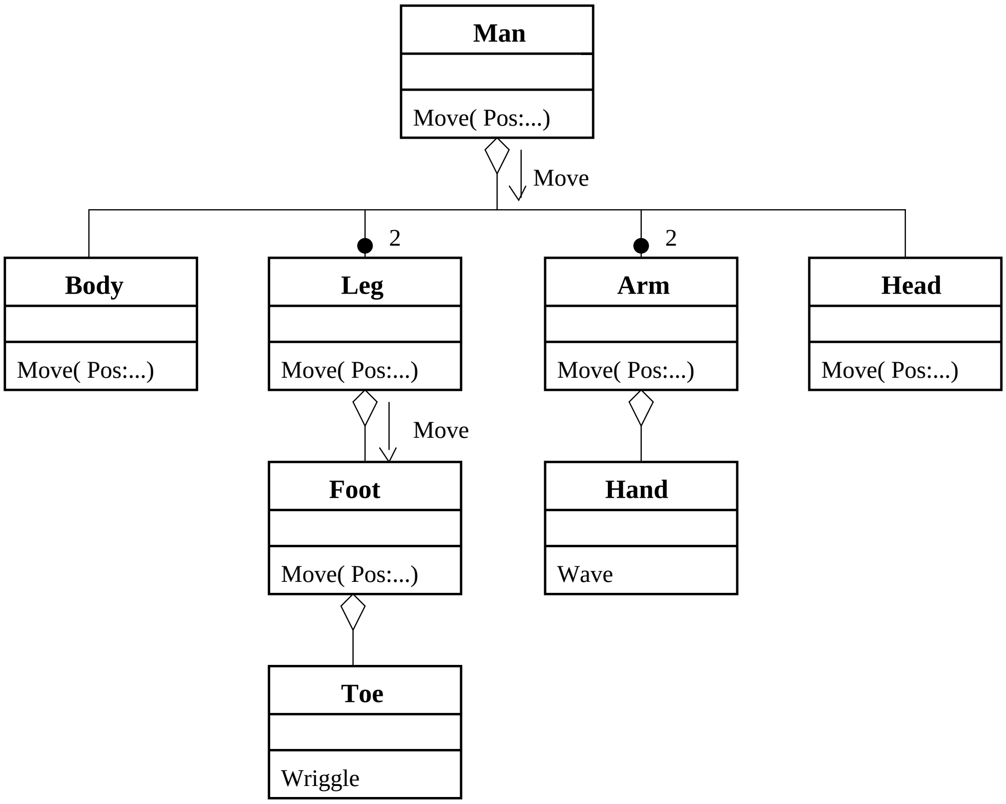


Figure 5.21: Propagation of Operation

Implementation

```
Move:
   (# Pos: ....;
     enter  Pos
    do      Pos->Manbody.Move;  Pos->Manhead.Move;
         ( for    i:Arm.Range repeat   Pos->Arm[i].Move  for );
         ( for    i:Leg.Range  repeat    Pos-> Leg.Move   for);
    #);
```

Figure 5.22: Move in the Man Attribute

```
Move:
   (#  Pos: ...;
    enter  Pos
    do
       ...{Implementation of the
          operation Move} ...
   #);
```

Figure 5.23: Move in the Foot Attribute

If the operation is propagated, the aggregate implements the operation calling the corresponding operation on its parts while, if the operation is no more propagated, in the do-part of the pattern that implements the operation, the real method of the operation is implemented.

5.1.9 Summary

In this section we have considered the most important OMT concepts proposed in the [RBP+91] and some of the advanced. We have translated this subset in the Object Oriented language Beta. Unfortunately, because of some lacks of the Beta language and of its compiler, we have not been able to translate some of these concepts in a complete correct way. Of the six points treated in our mapping, we have translated all the concepts related to class, object, and operation, while we have some problems for a complete translation of the concepts of inheritance, associations and aggregation.

We decided not to implement variable aggregation because in Beta it is not possible to implement a set of instances of a class, and because we think that the implementation with pointers is not the proper translation.

We had some problems in implementing bidirectional associations since we cannot ensure the consistency of the link, even if this problem is overcome at the level of the tool. Moreover, in One_to_One association, we are not able to ensure the consistency of Link Attributes, if this feature is used.

Multiple inheritance is supported by the OMT notation but it is not possible to translate this concept in Beta. Some aspects of this concept that can be translated have been given even if this problem is dealt in more details in the next section. Moreover, also in the single inheritance it is not possible to override default value of attributes because Beta does not support renomination of attributes inside a class.

Other Advanced Concepts

The OMT notation proposed in [RBP+91] also supports other advanced concepts that are not treated in this section. These concepts are not to treated because in our point of view some of them are optional in the design of a system, especially during early stages of modeling. Below we briefly show below these concepts and our motivations.

In the OMT notation, it is possible to add constraints on classes and associations. Constraints improve the odds that the behaviour of a class or of an association matches the expectations of its client and are written in natural language. They can be viewed as a way of expressing declaratively what might otherwise has to be written as procedural code. Unfortunately, Beta has no features like *assert* macro in C or the *assertion* in Eiffel in which these constraints could be mapped, and there is no way to translate whichever natural or mathematical languages in a Beta code that could be inserted in the do-part of a pattern. Also the concept of *ordered* association and the multiplicity are constraints, in this case on arcs. They have been implemented because their definitions are not so generic even if for the multiplicity we decided to translate only the one, the many and the fixed because we think that are the most important differences.

A One_to_Many association can be qualified. In the association shown in fig. 5.10 , for example, you can decide that each employee is qualified by a number and draw the appropriate notation. This is a way to transform a One_to_Many association in a One_to_One because, in this way, each person is referred by its own number. We have not implemented this kind of association because, following our implementation for the One_to_Many association, we have seen that in Beta it is not possible to define Set of pointers accessible by a string or a number. So the only possibility was the fixed multiplicity, but since from the OMT diagram is not possible to know the singular value of the qualificator, the result of the translation would have been the same that you have in the normal case with a repetition of pointers or a number of unnamed pointers.

5.2 Multiple Inheritance

Classification and Composition

When modeling phenomena and concepts from the real world, *classification hierarchy* and *composition* are fundamental methods of organization for abstracting the real world. A modeling language should have direct support for classification and composition, in particular O.O. languages have support for classification by means of the class/subclass mechanism. Moreover, some languages only support tree structured classification whereas others support non-tree structured hierarchies by means of the so-called multiple inheritance. On the other hand, in most O.O. languages there is little direct support for composition. This is usually supported directly through instance variables, or it is simulated using multiple inheritance. Beta does not support multiple inheritance because it provides inheritance by the use of super/sub-patterns that give rise to tree structured classification hierarchies. Instead, Beta directly supports composition and, as we have seen, contains facilities for three kinds of composition – Whole-Part composition, Reference composition and Localization. Now we want to see how the lack of facilities for directly support multiple inheritance is not a so big lack, since all the different uses of multiple inheritance may be simulated by means of part objects and virtual patterns.

Definition

Multiple inheritance has come up as a generalization of single inheritance. In case of single inheritance, a class may have at most one super-class, whereas multiple inheritance allows a class to have several super-classes. In this way, the subclass will have direct access to the attributes of the superclasses and virtuals of the superclasses may be redefined in the subclass in order to express adaptation to special needs.

When It Is Useful

In our experience in using multiple inheritance, we have seen that multiple superclasses are useful for several reasons. In fact, modeling the real world we often find more than one classification hierarchy for the same class of objects, or we need to apply the so-called *multiple override criterion*. Moreover, multiple inheritance is often used to combine unrelated classes for the purpose of reuse of code, but this (mis-)use of multiple inheritance often leads to complicate inheritance relationships. Later a way for simulating these various aspects of multiple inheritance by the use of the available Beta mechanisms is shown.

5.2.1 More Independent Classification Hierarchies

When It Is Used

In practice one often ends up with a classification hierarchy that is not tree structured. This may be the case if one is classifying the same phenomena according to independent properties. In this case, a non tree structured hierarchy can always be made tree structured, which, therefore, often gives rise to clumsy hierarchies [LMN93,p.305]. For this kind of classification, it is also possible to use part objects instead of multiple superclasses as shown in [CG90].

Example

For example, consider two possible classification hierarchies for persons: one classifies them according to which kind of sportsman they are (e.g. *TennisPlayer*, *GolfPlayer*,...)

and another classifies them into how they are students (e.g. *FullTime*, *PartTime*). Each of these are represented by subclasses (of e.g. *Sportsman* and *Student*, respectively). The class of persons being both a *Sportsman* and being a *Student*, *SportyStudent*, may obviously be defined as a class with *Sportsman* and *Student* as superclasses. In this way, however, this class is excluded from being specialized to e.g. a person that is a specific kind of sportsman and a specific kind of student. The superclasses are fixed and may not be redefined in subclasses of *SportyStudent*. This example is taken from an example found in [CG90], but in the following it is just indicated how part objects may be used instead of multiple superclasses. By the use of part objects and virtual classes we can realize this kind of classes organization in the following way:

```
SportyStudent: (#  TypeOfSportsman:< Sportsman;
                   TypeOfStudent:< Student;
                   TheSportsman:@ TypeOfSportsman;
                   TheStudent:@ TypeOfStudent;
                   ...
               #);
```

Also in this implementation, the constraint of the virtual classes implies that *TypeOf-Sportsman* may only be extended to one of the subclasses of *Sportsman*, while *TypeOf-Student* may only be extended to one of the subclasses of *Student*. A sporty student being, for example, a tennis player and part time student will then be defined by the following subclass:

```
TennisPlayerPartTimeStudent : SportyStudent
                              (# TypeOfSportsman::TennisPlayer;
                                 TypeOfStudent::PartTimeStudent;
                              #)
```

Figure 5.24:

5.2.2 The Multiple Override Criterion

In [Car91] and [Gui91] we can find one example of using multiple inheritance where the so called multiple override criterion is applied.

Suppose to have the following two classes: Example

```
A: (#  f1:< (# ... #);
       f2:< (# ... #);
       f3:< (# ... #);
   #)
```

```
I:  (#  f: (# ... #);
        g: (# ... #);
        h:< (# ... #);
    #)
```

Now we want a class *R*, derived from both *A* and *I*, which has to map *A*'s virtual functions into external facilities offered by *I*. In the example above *f1* is supposed to be mapped into *f* and *f2* into *g*. The class *I* may also impose some behaviour in *A* which is satisfied by *R*. In the example *h* is supposed to be mapped into *f3*.

Implementation Using Part Object and Binding Declaration

Using part objects and binding declarations the class *R* may be expressed as follows:

```
R:A  (#  x:@ I(# h:: (# do f3 #) #);
         f1:: (# do x.f #);
         f2:: (# do x.g #);
         f3:: (# do ... #);
     #)
```

R has as superpattern *A* and as a part object an instance of *I* (*x*) where the virtual procedure *h* is mapped into *f3* that is inherited by *A*. The virtual procedures *f1* and *f2* are mapped into *x.f* and *x.g* respectively. Note that even though *I* is not a superclass pattern of *R*, a redefinition of the virtual *h* still has access to attributes of *R*: *f3* is visible in the extension because the redefinition is defined in the scope of *R*.

5.2.3 Code Reuse

When It Is Used

There are of course also objects and classes that are used purely for implementation purposes and which may not represent phenomena and concepts from the application domain. In this case, probably, it is more correct to refer to *code sharing* rather than to *inheritance* even if it is often realized in this way. For this reason many authors distinguish between *inheritance of specification* and *inheritance of code*. Moreover, trying to combine unrelated classes, often leads to complicated inheritance relationships.

Subtype Relation

When inheriting a specification, there is a *subtype* relation between a subclass and its superclass. If a subclass is viewed as a subtype of its superclass, then the subclass should be applicable whenever the superclass is applicable. When inheriting code a subtype relation does not have to exist. It may be difficult (and it turned out to be difficult) to design *one* language mechanism that supports inheritance of specification and inheritance of code equally well.

Implementation

As proposed in [RL89] code reuse does not have to be obtained solely by inheritance, but may also be obtained by part objects. For this purpose he introduced a new Beta constructs for renaming of patterns. Since this construct is not supported by the Beta language, we have tried to implement his ideas using the available mechanisms of pattern variable and part object.

Example Using Part Object and Pattern Variable

Suppose we want to define a pattern *Stack* and to have already defined a pattern *Deque* (double queue). *Stack* can be defined as a subpattern of *Deque*, but it would work only by using some of the features of *Deque*. Therefore there will be no a sub-type relation between *Deque* and *Stack*. So this is not an appropriate use of subclassing since in Beta subclassing is supposed to be subtyping. We may alternatively describe *Stack* using inheritance by means of part object and rename the desirable properties as follows:

```
Stack:  (#  d:@Deque;
            push:##d.enterInFront;
            pop:(# ... #);
            intOfStack:@(# enter intOfDeque; exit intOfDeque #);
            ...;
        do
            d.enterInFront##->push##;
            ...;
        #)
```

In general in Beta the attributes of *d* are remotely accessible. So *Stack* can use all the properties that are in *Deque* using remote accesses, rename procedure patterns or attribute as for *push* and *intOfStack* respectively, or define its proper features as for *pop*. Renomination of procedure or functional patterns is done by the use of *pattern variables*. *push* is defined in the declaration-part of *Stack* as a pattern with the same structure of the procedure pattern *enterInFront* of *Deque*. Then in the do-part the pattern has to be assigned to push.

In the above example, if push is not only a renomination, but a specialization of the *enterInFront* of *Deque*, then this implementation gives rise to some problems since renomination with specialization as suggested by [RL89] (push:d.enterInFront(# ... #)), has to be reviewed in one of the following ways: If Also Specialization Is Needed

1) d:@Deque(# enterInFront::<(# ... #) #);
   push:##d.enterInFront;

declaring *enterInFront* as virtual pattern in *Deque* and specializing it in *Stack*. However, the compiler does not accept the declaration of patterns variable of virtual patterns;

2) d:@Deque;
   push:##d.enterInFront(# ... #);

modifying the pattern at the level of the pattern variable, but also in this case the declaration of a pattern variable qualified by a modified pattern is not accepted by the compiler. So, reuse of code of specialized operations cannot be simulated in Beta.

In Multiple Inheritance

The reuse-of-code-by-part-objects approach above may easily be generalized also to cover the need for multiple inheritance. Suppose we have two patterns A and B and that T may be described by inheriting from A and B. Using part objects T may be declared in the following way:

```
T: (#  a:@A;
       b:@B;
   #)
```

in which the same considerations on attributes and operations of the example before may be done. Note that this form of multiple inheritance resembles the one implemented for C^{++} [Sto86].

Name Conflicts

In order to resolve name conflicts and in order to avoid the compound identifiers, renaming of attributes (as introduced before) may be used. In case there are only a few name conflicts, then singular part objects with renaming specified will be the solution. For example if there is a conflict with the attribute z in the patterns A and B, a solution can be the following:

```
T: (#  a:@A;
       b:@B(# bz:(# enter z; exit z #) #)
   #)
```

In this way, the attribute z in T will be the one from A, since the attribute z in B has been renamed with bz.

5.2.4 Summary

In this paragraph we have demonstrated that some of the various uses of inheritance may be provided by the use of part objects and other features as pattern variables and/or virtual patterns. Using virtual patterns and part objects, we can declare classes that depend on more independent classification hierarchies. By specifying singular part objects that are specializations of some general classes, it is possible to add attributes and to redefine virtual procedures/functions. So, for each part object a specialization may be obtained realizing the multiple override criterion. Code reuse by renaming attributes and operations has been realized by the use of part objects and patterns variable but we have not been able to realize renomination of operations with specialization. Moreover, the provision of multiple subtyping is not covered by part objects. This is a relevant drawback for a programming language, like Beta, that mainly intends inheritance for hierarchical classification of concepts. For this last reason and for the fact that we have not been able to find an unique implementation able to cover all the aspects of multiple inheritance, we conclude that in Beta it is not possible to simulate the concept of multiple inheritance.

Chapter 6

Tool Modelling

What we have to construct is a tool able to be inserted in an Object Oriented software development environment in which Beta is the used programming language. Since Beta is a multi-perspective language, this allows the programmer to have some freedoms during the implementation of a system that makes the implementation realized not in a pure Object Oriented style. For this reason we have chosen OMT as the Object Oriented methodology the user has to follow in drawing an Object Oriented Diagram, which will be implemented in a textual notation. So, what we want is a tool composed of a graphical and a textual editor. The user should be able to draw an OMT Object Model with the graphical editor and to complete a correspondent Beta skeleton with the textual editor. Before implementing our tool we had some interactions with students who study Beta and use the Mjolner Beta system to program with this language, in order to know the requirements that they would like to have in a tool as we ware going to construct. Moreover, we had some experiences with the tool "Software Through Picture" [IDE93] - an OMT graphical tool that is available in this university - to evaluate the characteristics that could be important for our graphical editor. This chapter shows the requirements we think are important for the construction of our tool and the specification of the tool made in a OMT-like formalism.

6.1 Requirements

Realization of the Mapping from OMT to Beta

The first requirement is that the tool should be able to realize the mapping of the OMT concepts in the Beta concepts how it is explained in section 5.1. The graphical editor should allow the user to draw all the concepts that can be translated maintaining the original OMT notation as much as possible. The textual editor should provide the user with an automatically generated Beta skeleton following the mapping and being always consistent with the graphical notation.

Expandibility

Since we have chosen to translate only the Object Model from the three models offered by OMT, we have decided that the tool should be expandable for future implementations in which other developers could decide to treat also the Functional and the Dynamic Models.

Contemporary Use of the Graphical and Textual Editor

We require our tool let the user to have the freedom to implement his system working contemporaneously with the graphical and textual editor without waiting for the complete drawing description of the system. However, we have decided that our tool should not require *reverse engineering*. Reverse engineering is considered to be the problem of going back to design diagrams from the code [LMN93,p.50], but we only allow the user to change the implementation of his system changing the graphical version. Moreover, the user should not have the possibility to textually modify the generated code correspondent to the drawing. This is to subtract the user the freedom of the Beta language that allows to write a not Object Oriented program, and constraints a right implementation that follows the OMT Object Model.

Implementing more than one Object Model Contemporaneously

The tool should provide the possibility for the user to implement more than one Object Model contemporaneously. Multiple graphical main windows should appear on the screen in which the user can draw his models and multiple text editor windows should be opened simultaneously displaying the code correspondent to the diagrams as well as the contents of separate fragments.

A Syntax Driven Tool + a Free Textual Editor

Moreover, we want our tool syntax driven in both the graphical and the textual editors. We think that syntax driven tools provide users with the best support. In fact, they can assist inexpert users in producing syntactically and semantically correct documents right away and relieve them of the mundane task of repeatedly typing concrete syntax which is generated automatically. For instance, here in Dortmund the students use OPUS as a graphical tool that is syntax driven, and they said that they have good experiences with it. In addition, however, our tool should also support the facilities to freely complete the generated Beta skeleton with a conventional textual editor. This might be useful for more expert users for which these facilities might provide the faster way for programming.

Clear Drawing

After having seen other tools, we consider that the clearness is an important requirement of the graphical design drawn with the graphical editor. In each moment during the design phase, the user should be able to have a global and clear view of the Object Model he is drawing as much as possible. The OMT tool we have seen, for example, was not useful for designing systems in large because of the final representation of the Object diagram that resulted too big to be visualized in a window. It was also not easy to understand because it was too complicated. Moreover, these tools are resulted not easy to use. For these reasons, we have thought about the simpleness of our tool.

A Tool Easy to Be Used

Our tool should be easy to use in two principal ways.

- It should be as much natural and intuitive as possible, i.e. the user should see our tool as a facility for building a system instead of drawing it by hand.
- Moreover, the capability offered by the tool during the design phase should be clear for the user. This implies, for example, the use of icons to show the OMT graphical symbols the user is allowed to draw in a certain moment, and the

meaning of the main functions he can use.

Possibility to Insert Documentation

Another requirement that we deem important, after our experience in using graphical tools, is the possibility for the user to complete the design with an opportune documentation. The tool should give the possibility to add comments during the insertion of operations and attributes in the graphical design. The comments should also be automatically inserted in the Beta skeleton when this is generated.

Communication of Application Action to the User

Obviously, the tool should support those features that all the tools should support as the communications of application actions to the user [OSF92]. These include the possibility of the the tool to:

- *Give the user feedback*, i.e. let the user know that the tool has received his inputs, moreover, give users feedback whenever they have selected a component or menu item by highlighting the component or the menu item in some way.
- *Anticipate errors.* The tool should be able to recover from the unintended or erroneous operations. Whenever an inappropriate mouse button is pressed or an inapplicable menu option is selected, an error box with the corresponding message should appear. The user should not be able to continue without acknowledging the error (by pressing the OK button). Moreover, it should be able to provide more information when asked. We have thought that context-sensitive help improves understanding, reduces errors and eases recovery efforts.
- *Use explicit destruction.* Explicit destruction means that, when an action has an irreversible negative consequences, it should require the user to take an explicit action to perform it. We think that warnings protect the user from inadvertent destructive operations, yet allow the user to remain in control of the application, and encourage him to experiment without fear of lost.

6.2 Design

An Extended OMT Graphical Notation

In this paragraph we are going to explain the design phase followed during the realization of our tool. We have decided to design our tool complying with the OMT methodology which has been extended to include the concept of *Subsystem* [Emm95]. This concept has been introduced to increase the comprehension of the diagram by adding one abstraction level. In this way each subsystem has been refined by another OMT diagram. The graphical notation chosen to represent a subsystem is a shadowed box, as shown in fig. 6.1.

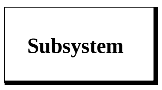


Figure 6.1: Graphical Representation of a Subsystem

In a diagram, each class or subsystem that has a relationship to a subsystem is given as a *port* in the diagram that refines the subsystem. We represent these ports by means of colored boxes as in fig. 6.2. We have decided that ports representing subsystems may contain definitions of operations that are defined in one of the classes that composes the subsystem.

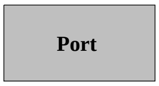


Figure 6.2: Graphical Representation of a Port

In the following sections we show the general design of our tool and we refine two of the most interesting of its subsystems.

6.2.1 Architecture of the Tool

In developing this phase, we have followed the requirements presented in the previous section. This section gives a presentation of the developed design and an explanation in which the points satisfying the requirements are underlined. In the following the first level of the tool design is shown.

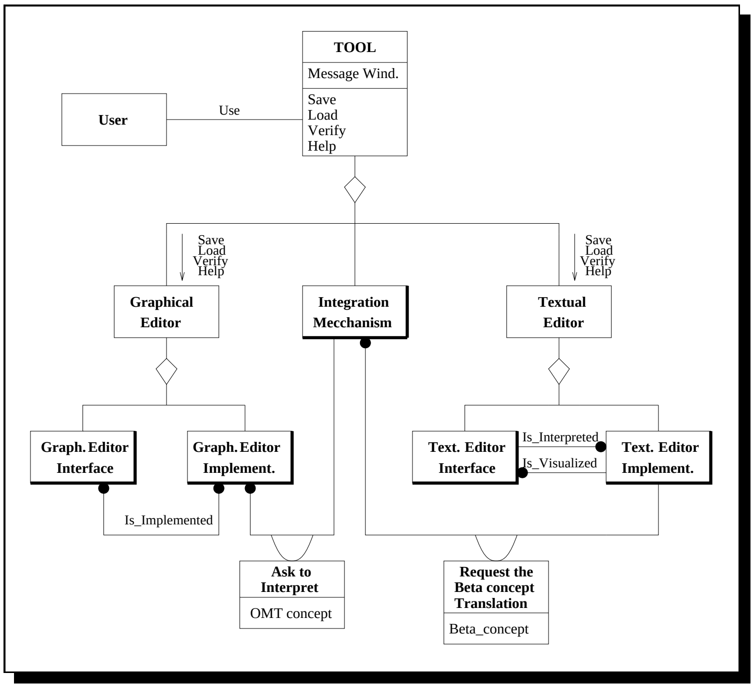


Figure 6.3: First Level of the Tool Design

In this diagram the class **Tool** represents our tool. Since we have required our tool to realize the mapping from an OMT diagram into Beta code, **Tool** is composed of:

Tool

- A class **Graphical Editor**: to allow the user to draw an OMT Object Model;
- A class **Textual Editor**: to visualize the Beta skeleton corresponding to the Object Model drawn with the graphical editor, and to allow the user to complete this skeleton;
- A class **Integration Mechanism**: to integrate the two editors and to implement the theoretical mapping from OMT to Beta. This allows the automatic generation of the Beta skeleton.

As we can see from the diagram in figure 6.3, we have foreseen a **Tool**'s attribute to realize the Communication of Application Actions to the user and also some basic operations that each syntax-driven tool should have [OSF92], such as *Verify*, *Save*, *Load* and *Help* (which is one of our requirements). In the design, the **Graphical** and **Textual Editor** are divided into an **Editor Interface** and an **Editor Implementation** design by means of subsystems.

Graphical Editor Interface

The **Graphical Editor Interface** provides the user with the means to draw an OMT diagram. These means are, for example, the operations *Move*, *Delete* and *Create*, that are defined for each OMT concept and are implemented in the **Graphical Editor Implementation**.

Graphical Editor Implementation

The **Graphical Editor Implementation** recognizes the kind of graphical OMT concepts and asks the **Integration Mechanism** to realize the mapping from the received OMT concepts into the appropriate Beta concepts.

Integration Mechanism and Textual Editor Implementation

Once the right translation has been decided, the **Integration Mechanism** asks the **Textual Editor Implementation** to translate the Beta concepts into the corresponding textual representation.

Textual Editor Interface

Finally the textual representation is visualized by the **Textual Editor Interface**. This sequence of interactions allows the automatic generation of the Beta skeleton. Moreover the user may require to complete the skeleton by the **Textual Editor Interface**. For this reason, the **Textual Editor Interface** must provide operations, such as *Expand_Placeholder*, *InsertText*, and *ChooseMode*, which allow the insertion of code in a syntax-driven or free-textual input mode.

The next sections analyzes the **Graphical Editor Implementation** and the **Textual Editor Implementation** subsystems in more detail, while we omit a detailed explanation of the **Graphical Editor Interface** and the **Textual Editor Interface** subsystems in which the declaration of the structures of the two editors is designed. Moreover, we have also decided to omit the refinement of the **Integration Mechanism** subsystem since its structure strictly depends of the implementation of the graphical and textual editor.

6.2.2 The Graphical Editor Implementation Subsystem

The **Graphical Editor Interface** provides the means to draw OMT diagrams, which are recognized from the **Graphical Editor Implementation**.

Element Implementation

As shown in figure 6.4, the implementation of each OMT concept is realized by **Element Implementation**, which is the superclass of an inheritance hierarchy defining all these concepts. In **Element Implementation** the operations to handle the insertion and the deletion of the elements are defined. These operations are then inherited and properly specialized in the subclasses.

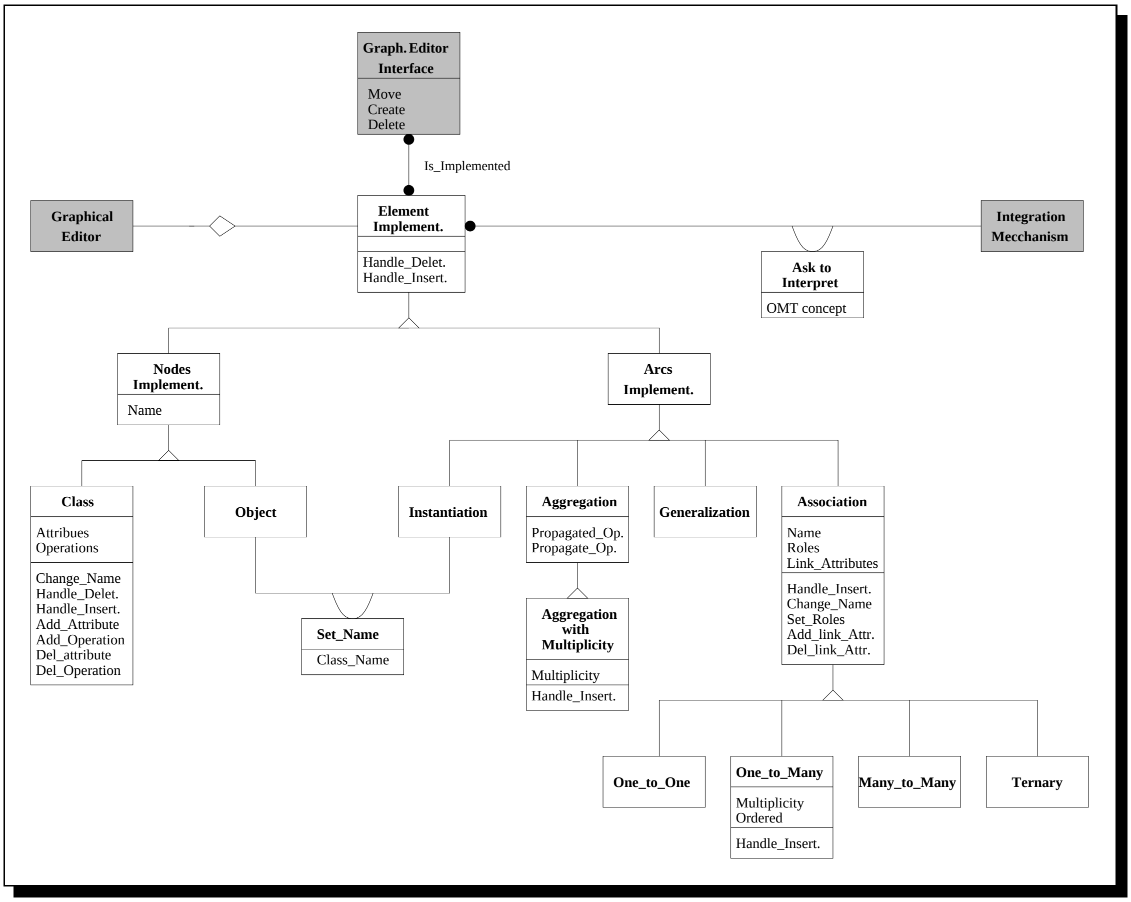


Figure 6.4: Diagram Refining the *Graphical Editor Implementation* Subsystem

Nodes Implementation and Arcs Implementation

Since the elements of an Object Model are nodes and arcs, the direct specialization of **Element Implementation** is given by the **Nodes Implementation** and **Arcs Implementation** classes. These two subclasses are in turn specialized by all the elements defined in the OMT notation.

Attributes and Operations on Classes

In each subclass, in fact, the attributes and the operations to handle these attributes are defined. The attributes, as they should, represent features that characterize the OMT concept the class represents. We only want to underline the designing choices made for the class representing One_to_Many associations (**One_to_Many**) and variable aggregations (**Aggregation_with_Multiplicity**). These classes have the multiplicity, as an attribute, but the operation to explicitly set it is not defined. In fact we think that the insertion of this feature should be realized during the insertion of the whole relationship which it belongs to. Indeed, with the chosen hierarchy schema, the multiplicity must be an integral part of the relationship and not an additional property. For this reason, in these classes, the operation *Handle_Insertion* is specialized to also allow the insertion of this feature. Moreover, as we can see from the figure, the operations *Handle_Insertion* and *Handle_Deletion* are also specialized in other classes. This happens, for example, in the class *Class*, because it should not exist a class without name, and because there could be situations in which a class can not be simply deleted - for example if it is in the middle of a generalization tree. The operation *Handle_Insertion* is also obviously specialized in the class **Association** to allow the insertion only between classes (and not for example in the middle of an empty screen) and because this is the only relationship that needs an inserted name.

Relationships in the Diagram

A particular treatment in setting the name is also necessary for *Objects*. The association *Set_Name* between the classes **Object** and **Instantiation** means that the name of an object can be set only when an instantiation relationship between an object and a class is created. The Link Attribute *Class_Name* allows to set the object name with the same name of the class which it belongs to. Instead, the association *Ask to Interpret*, between **Element Implementation** and **Integration Mechanism**, allows the interpretation of each OMT concept implemented in the **Graphical Editor Implementation**. This association, because of the inheritance structure, is in fact inherited by all the subclasses that specialize the Link Attribute. The request of interpretation represents the last step the subsystem has to perform. Note that in the port **Graphical Editor Interface**, representing the namesake subsystem, the operations *Move*, *Create*, and *Delete* are declared. These operations must be defined in the interface of the textual editor to allow the designing of OMT diagrams.

6.2.3 The Textual Editor Implementation Subsystem

As we have explained at the beginning, the **Textual Editor Implementation** translates the Beta concepts received from the **Integration Mechanism** and asks the **Textual Editor Interface** to visualize them. The designed diagram refining the correspondent subsystem is shown below.

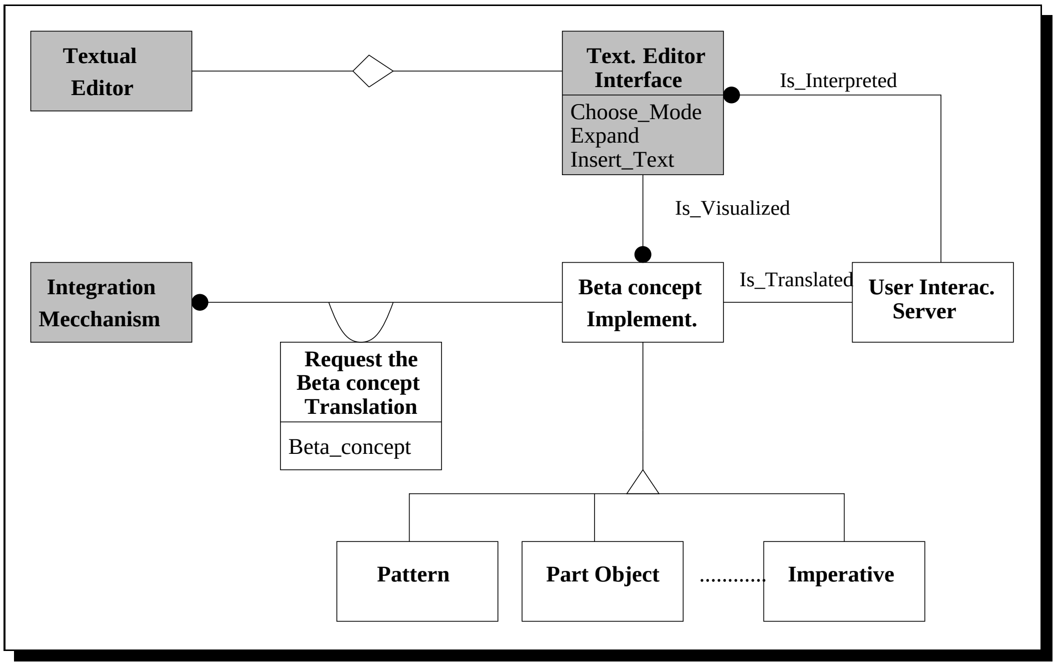


Figure 6.5: Diagram Refining the *Textual Editor Implementation* Subsystem

The most important class is **Beta Concept Implementation**. This class is the superclass of an inheritance hierarchy in which all Beta concepts are defined. **Beta Concepts Implementation** implements all those Beta concepts required by the **Integration Mechanism** which are then visualized by the **Textual Editor Interface**. Moreover, when the user interacts with the textual interface, this interaction is interpreted by the **User Interface Server**, which recognizes the kind of requested service and asks the proper translation to the **Beta Concept Implementation**. Also in this case, **Beta Concept Implementation** translates the request and implements the Beta concepts which will be visualized by the **Textual Editor Interface**. Note that all the implemented Beta concepts are visualized by means of the same association (*Is_Visualized*). In fact, whether the request of implementation comes from the **Integration Mechanism**, or from the **Textual Editor Interface**, the Beta concept to be visualized must be textually represented in the same way. As shown before, the port representing the **Textual Editor Interface** subsystem contains the declaration of operations needed to textually complete the Beta skeleton. These are synthesized in *Choose_Mode*, *Expand* and *Insert_Text* and allow the user to write in a syntax driven or free textual mode.

6.3 Summary

In this chapter we have seen which are the most important requirements of our tool. We said that we want a syntax-driven tool which lets the user draw an OMT diagram and which is able to translate this diagram in the corresponding Beta code.

Moreover, we gave the design of the complete tool underlining when some requirements are satisfied. During this phase we have individualized a graphical, a textual editor and an integration mechanism as the main components of our tool. Of these components we have only analyzed the design of the graphical and textual interface that in the first level are drawn as subsystems in more detail.

The whole design has been realized complying an OMT notation extended with the concept of subsystem. This notation has allowed us to express all the needed concepts in an elegant manner and on the basis of a well known Object Oriented methodology. This choice has been done in order to give an example of a design using OMT methodology and to show the advantages of the OOD. Our experience has reinforced the consideration made in section 4.2 about the subsystems. We have, in fact, ascertained that subsystems are useful in designing big systems and that the lack of a graphical notation for this feature in the OMT methodology is a disadvantage when you want to design in large.

The next chapters will present the choices made in order to find the means to realize our design. Then, the final implementation of the tool will be described as a following phase of the design showed in this paragraph.

Chapter 7

Selecting a Graphical Editor

The first step in the construction of our tool is to build up a graphical editor which enables the user to draw an OMT Object Model as well as to become easily integrated with a textual editor, in order to transform the OMT diagram into the correspondent Beta skeleton. Recently there has been much interest in the graphical editing model of interaction, with which the user can manipulate data structures by editing their visual representation in a syntactically and semantically consistent fashion. Although graph visualization is accepted as indispensable for many domains, the use of this technique is not very common in today's computer applications. Frequently a user has to deal with uncomfortable textual interfaces or poor ad-hoc drawings of graphs. The reason for this deficit is the great effort to implement a satisfying graph layout and to make the tool able to be integrated in a software development environment. There are several powerful tools for generating graphical user interfaces but, unfortunately, most of the existing systems are designed as isolated viewing components and do not offer good interaction capabilities for use as a full user interface for application programs.

In this chapter we analyze three different ways to generate a good graphical user interface: *daVinci*, the *Tk Toolkit* and *GraphProject*. These constitute three different approaches which will be compared in the next sections. This comparison will show that GraphProject is the most suitable for implementing our graphical editor.

7.1 daVinci

daVinci is a generic visualization system for generating high quality drawings of directional graphs. It has been developed by the Institute for Formal Methods in Software-Engineering at the University of Bremen, Germany, since the end of 1992 by Michael Frölich and Matthias Werner. *daVinci* is written in the pure functional language ASpecT, which has developed at the University of Bremen as well. The OPEN LOOK compliant user interface is implemented in C using the window toolkit Xview [FW94].

Term Representation of Graphs

daVinci allows the user to generate a graphical view of directional cyclic or acyclic graphs, in which readability qualities, as minimal number of crossing or minimal number of bends, are supported. The specification of a graph is given as a term representation which is composed of ASCII characters. Hence it is possible to create these graphs very easily, even with an usual text editor. The syntax of the specification is given in a Backus Naur Form and is determined by the used programming language ASpecT for simple parsing, i.e. it can be loaded directly with a given read function. The system is able to visualize all classes of graphs that can be expressed in the term representation: cyclic graphs, graphs with multiple edges, graphs with self-referring edges, graphs with one level and empty graphs are being supported. The term representation can be either loaded from a file or sent by a connected application program to the daVinci application interface.

Connection with Application

One fundamental concept of this system is that the graph visualization system is based on the idea that possibly there is a connection and communication with an application program. This is for generating and controlling the graph structure. So, it is not necessary for an application to store a graph in a file, which has to be loaded by the user. For this reason, the connected program is exclusively responsible for controlling the graph structure, and the only task of the visualization system is to display this graph on the screen: *daVinci*'s task is merely the presentation of the graph, instead of the modification like a graph editor. For this reason, *daVinci* provides an application interface for the connection with a program that is generating and controlling a graph. By using the commands of the communication protocol, a connected application is able to send graphs to the visualization system. The communication between both systems is event driven, i.e. the application sends a graph and is informed about events like selection of node by the user. After receiving an event notification, the application must interpret this information in its own context. Depending on the event, the application can modify its own data structures and sends an updated graph back to the visualization system to display the changes. So the application interface is bidirectional.

Interactively Modifiable

Moreover, another goal of *daVinci* is the ability to influence the visualization of a graph interactively. Experiences have shown us that the user often likes to influence or modify an automatically generated layout in detail, since standard representations are rarely perfect. For this reason, it is possible to influence the generated graph layout directly at "runtime" and these interactions have an immediate feedback to the visualization: the automatically calculated layout of nodes and edges can be refined manually (fine tuning), part of the graph can be hidden for a time (abstraction), the scale of the graph can be modified (scaling) and so on.

7.2 Tcl and the Tk Toolkit

Tcl and *Tk* are two software packages which together provide a programming system for the development and usage of graphical user interface applications. *Tcl* was born at the University of California at Berkeley since 1988 as a general-purpose scripting language which could be reused for many different purposes in many different applications [Ous94]. After some years a new idea was born: the component-based approach. Furthermore to allow users to develop a new component which was subsequently combined with existing components, *Tk* was born. *Tk* allows the components to be either individual user-interface controls or entire applications; in either case components can be developed independently and *Tcl* can be used to assemble the components and communicate between them.

X Window

Tk is a toolkit that allows the user to create graphical user interfaces for the X Window System by writing *Tcl* scripts in which most of the Xlib functions are used. It provides some commands for creating user interface elements called *widgets*, arranging them into interesting layouts on the screen using *geometry managers*, and connecting them with each other, with enclosing application, and with other applications. Both *Tcl* and *Tk* are implemented as a C library that can be included in C applications, and they comprise a collection of functions that can be invoked from an application to implement in C new widgets and geometry managers.

Drawings Graphics

Tk provides canvas widgets to allow the user to display and manipulate a variety of graphical objects such as rectangles, lines, bitmaps, and text strings. Texts are used to display and edit large multiline pieces of text. Both widgets are peculiar in that they allow you to tag objects and manipulate all the objects with a given tag. The tagging and the binding mechanisms make it reasonably easy to "activate" text and graphics so that they respond to the mouse. At the same time, however, all the most frequently used procedures, for example to allow nodes to be dragged interactively with the mouse, need to be defined explicitly.

New Applications

Tcl makes it easy for application to have powerful scripting languages. To create a new application, all the user needs to do, is to implement a few new *Tcl* commands that provide the basic features of the application. Then you can link your new commands with the *Tcl* library to produce a full-function scripting language that includes both the old and the new commands. Moreover, *Tcl* can also be used as a communication mechanism to allow different applications to work together. For example any windowing application based on *Tk* can send a *Tcl* script to any other *Tk* applications in order to be processed there.

7.3 GraphProject

GraphProject (GP) was developed by Engineering in Pomezia (Italy) since 1994 [CI94] and it is a tool for generating syntax-driven graphical editors. It allows the user to specify a graphical language, through the description of its composing terms and their composition rules. GP generates from this specification and from the implementation of the methods associated with each term, a graphical editor for the correspondent graphical language in a completely automatic way. *GraphProject* offers a specific language, i.e. *GP_Language*, to write the specification of the graph layout in which are defined: the different kinds of nodes and arcs, the names of the methods to manipulate them and the constrains between them in order to enable the user to define layout restrictions.

Hierarchy of Graphs

A GP diagram is structured as a set of graphs, possibly of different types, hierarchically organized. Each graph is constituted by *nodes*, *arcs* and *hypernodes*, where an hypernode is a particular kind of node that can be exploded in another graph as shown in fig 7.1. The definition of an hypernode describes the type of the graph associated with it together with its composition rules. The graph type associated with an hypernode can recursively contain the hypernode itself, among its composing node types. This means that there is no limit in the number of graphs composing a diagram.

Attribute and Methods

One of the most important features of GP_Language is that it allows the associations of attributes and methods to each object of the graphical language. Each term of the graphical language is seen as an object which encapsulates its own state (by means of the attributes) and is capable to react to messages sent to it. This feature makes it a tool that can be really seen as a generator of interpreters for graphical languages. To some extent, this permits a user to execute a GP diagram by invoking the methods associated with each object that compose it.

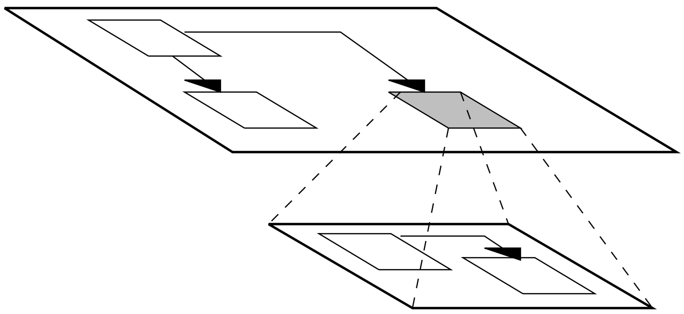


Figure 7.1: A GP Diagram

Object Oriented Editor

Moreover, the structure of the generator editor resulting from the specification is built in a completely Object Oriented way. The code of the editor is written in the O.O. language C^{++} in which each term is seen as a specialization of the predefined classes *HYPERNODE*, *NODE* and *ARC*.

O_2 Schema

GP runs on the O_2 database which is an Object Oriented database. This kinds of databases have been built to include all the features of both common database management systems and Object Oriented systems [BDK92]. Basically, their five main characteristics are: persistence, secondary storage management, concurrency, recovery and ad hoc query facilities. In the GP system, for each specification, a corresponding O_2 schema is automatically generated. This schema allows either a user or a tool, to retrieve, for each specified graphical language, all the diagrams drawn and stored with the associated editor. Therefore, it is possible under the O_2 system to browse these diagrams and access properties (but not methods) associated with their objects.

7.4 Comparisons and Conclusions

In the previous sections we have presented not only three different tools to build graphical interfaces, but more precisely we have presented three different approaches.

daVinci's approach is particularly useful when the user wants to write programs in order to draw or print graphs. In fact, a user is able to save a graph visualization in a Postscript file which can be printed on a Postscript printer or it can be included into a document with the appropriate publishing software. However, as we said before, daVinci's task is merely the presentation of a graph. To build up a real graphical editor, the developer has to write most of the code itself to control and modify the structure of the graph, and has to define the methods associated with each component of the graph. By taking advantage of the application interface, then an arbitrary program, in this case the editor, is able to use *daVinci* only as an abstract user interface for graphs.

The *Tcl/Tk's* approach provides probably the best framework of behaviour specifications to guide application developers, widgets developers, user interface system developers, window manager developers in the design and implementation of new products. Even if compared with toolkits, where you program in C - such as the Motif toolkit - *Tcl/Tk*, it is much easier to use and there is much less code to write. However, also this one presents some drawbacks: first the good graphical layout is payed in the effort to define the widgets and the commands to interconnect and manipulate them. Second, we have the same problem found in daVinci, where we need to write also the code to realize the whole editor.

Differently from these two, *GraphProject* is not only suitable to specify user interfaces, but it is really a graphical editor generator. This means that the developer only has to take into account the specification of the component terms of the graphs and their methods. All the code to manipulate the editor is automatically generated by the GP compiler. However GP layout is not as good as *daVinci*'s layout. Even if the efficiency of the layout algorithm is not sufficient for very large graphs, nevertheless *daVinci*'s graph layout is of high quality in most cases and it's better than *GraphProject*. For example, in *daVinci* there are two pixmaps, one for drawing the nodes and one for drawing the edges, while in GP there is only one for nodes.

Another important consideration for GP is the possibility to be integrated with other tools. Since our goal is to build a tool able to transform a graphical representation into a textual representation, what we are looking for, is a graphical editor which will be integrated with a textual editor. However, from this point of view all the approaches have the same behaviour: the integration has to be realized in the implementation of the methods associated with each object. The only difference between the approaches is that when the GP editor is generated, the compiler transforms each term in a C^{++} object providing also the capabilities to handle the state of the object and its methods. On the other hand, the use of the graphical editor generated by GP needs to dispose of an O_2 database system, and this may be seen as a disadvantage. In fact, this choice restricts the portability of the whole tool, but, if you (as in our case) dispose of this database, the renounce of the portability is sufficiently compensated by a great increase of efficiency.

For these reasons, we think GP is the most appropriate to build our integrated Environment. Using GP, the graphical editor can be easily generated in a completely automatic way from the specification, and can also be easily modified. This is due to the fact that the structure of the generated editor completely results build in an Object Oriented way. This also allows an implementation that can really realize the architecture of the editor developed during the design phase (see section 6.2).

Chapter 8

Selecting a Textual Editor

The goal of this chapter is to show the process followed in finding a syntax-driven textual editor generator able to be integrated with the graphical editor as well as to satisfy the basic requirements of tool specifications. So, we see which are the most powerful existing tool generators, such as *Cornell Synthesizer Generator*, the *Centaur System*, the *IPSEN* Project and the *GENESIS* generator, and we compare them, with respect to the tool specification languages they use. Then we explain the most important reasons why we have chosen the *GENESIS* tool.

Looking for these tool generators, we have also seen the *Mjoner Beta metaprogramming system* which is a programming environment that supports design, implementation and maintenance of large production programs, and in particular support the Object Oriented programming style. Unfortunately we have seen that some of its features are not suitable to build our editor and they made us deciding to reject it. For example, we have seen that this system maintains persistent data storing in a set of operating system files instead of a database system like most of the common tool generators do. In this way, information is often stored redundantly, and this leads to problems with consistency and difficulties with extracting and inserting new data. Instead database systems provide a single uniform view of the data, expressed in a structure-independent terms. Moreover, we want to underline that differently from other tools such as the *Cornell Synthesizer Generator* which take a specification in the form of an attribute grammar, the Mjoner Beta editor generator is based on the context free grammar formalism. Then in the Mjoner Beta the static semantic checkers are programmed by means of the metaprogramming system, while editors based on attribute grammars provide a declarative way of expressing semantics.

We are not going to treat the Mjoner Beta metaprogramming system in more details since the existing documentation is not enough for this purpose. On the contrary, we present the approach of each of the other generators on the basis of a given set of requirements on tool specification languages.

8.1 Requirements of Our Textual Editor

In this section we identify the parameters of tool generation process that usually vary between different tools in order to select the most important requirements of the textual editor we were going to define.

Abstract Syntax

We share the view of many researchers that tools need to be directed towards the languages they are intended for, in order to provide users with the best support, in particular if the end user does not master the language. Then, since we have required our tool to be syntax-directed, the first parameter we consider is the syntax of the supported language. The syntax of a formal language is usually defined by a grammar given in terms of an EBNF (Extended Backus Naur Form). Therefore, each increment of the language is defined by rules. At the same way, also tool specifications for syntax-directed tools must define the rules for each increment, i.e. they must define the syntactical structure of the document that usually is denoted as *abstract syntax*. In order to define it, a tool specification language must be capable to define both the increments and the mechanisms that must be applied to construct an increment. In this way, the syntactic correctness is always preserved because the tool only inserts syntactically correct templates, and only those commands are offered whose execution cannot violate the syntax definition.

Concrete Syntax

In order to translate the internal representation into an output representation the user is familiar with (in our case a Beta program), the textual representation of documents of a particular type may have to be adapted to particular processes. Usually this process is defined by the *concrete syntax* of a language, which defines the external representation of increments. The concrete syntax may be defined at once or split into the specification of the *input syntax* and the definition of the output syntax. The input syntax must be defined in order to construct a parser that checks freely input materials, while the output syntax, i.e. the *unparsing scheme* must be defined to transform the same input materials into an external textual representation.

Static Semantic and Inter-Document Consistency

In addition to grammar rules that define the language's abstract syntax, an editor specification contains declarations that define how to make static inferences about the objects being edited and how to define inter-document consistency constraints which increments of documents must obey. Tools must obey the *static semantics* of documents and also consistency constraints to related documents and provide the user with feedback about the errors when these constraints are violated. Therefore the specification must define the static semantics conditions such as scoping and typing for increments and also *inter-document consistency constraints* an increment has to related increments of other documents. Moreover the specification must also define whether or not violations are tolerated.

Operations

Besides generic operations of tools, such as opening a document or copying an increment to a paste-buffer, our editor specification should offer an increment-based manner to define increment specific operations for restructuring objects. Moreover, this kind of operations should depend on the current increment and on the language the tool is intended for. For example, the tool can use a selection to deduce possible and reasonable operations and present them in an appropriate way to the user.

Free Textual Input

Sometimes the structure-oriented mode of editing a document is not appropriate. For instance experienced users who master the language are capable of typing increments very fast. Such users may be faster in typing an increment than repeatedly selecting an increment. To address this kind of problems, users expect from the tool not only support for structure-oriented editing, but also facilities to freely edit increments with conventional text editors.

Concurrency

Usually more that one developer is involved in a software process, hence a tool must provide *multi-user support* which mean that document change in general and in particular the sketched change propagation must be subject to concurrency control. So the specification must explicitly define which operations must be performed in isolation to other concurrent operation execution in order to avoid anomalies of document updates.

8.2 Cornell Synthesizer Generator

The *Cornell Synthesizer Generator* is a system which is capable of generating hybrid syntax-directed tools. It has been developed at Cornell University, Ithaca, NY since 1978. The system is based on the concept of ordered attribute grammars [Kas80], which are a subclass of attribute grammars for which is efficiently decidable whether a given grammar is well-defined [Knu68]. The Synthesizer Generator defines both the grammars for syntax and static semantics and the functionality of tools using the *Synthesizer Specification Language (SSL)*.

Abstract Syntax

The abstract syntax consists of a collection of *productions* where the left-hand side of a production is a *phylum* and the right-hand side is a set of *operators*. Each phylum corresponds to a non-terminal symbol of a grammar such as an *expression*, *statement* or *declaration*, and represents the set of derivation trees that can be derived from the symbol. Instead each production corresponds to an operator and its purpose is to identify the production instances in a derivation tree. The abstract syntax may also be specified in a textual representation. In fact, the developers have some language constructs for defining unparsing schemes and can decide which nodes of the abstract-syntax tree are selectable or editable as a text. Besides for each phylum declared in the *SSL* specification, the transformation rules are offered to the user interface either in pop-up menus, or can be invoked from a command-line interface.

Attribute Grammars

Like all systems based on the concept of attribute grammars, also *SSL* offers the means to define static semantics of languages in terms of *attributes* and *equations*. In *SSL* the equations are associated with operators, while the attributes are associated with phyla and may be computed on the base of the attributes of the child phyla (*synthesized*) or of ancestors phylum (*inherited*). One of the main advantage deriving from the use of attribute grammars is that the designer does not need to bother about sequence execution of equations. In fact, this sequence is automatically derived by the *Synthesizer Generator* from the dependencies between equations. At the same time, the use of attribute grammar is also a weakness, by the fact that these grammars are strongly based on the syntax structure of a particular language. Therefore, e.g. the consistency constraints between phyla of different documents which are not syntactically related, cannot be expressed.

Concurrency Constraints

Another drawback of the *Cornell Synthesizer Generator* is the fact that the system is not able to define concurrency constraints on tool execution. In fact, the authors of the generators have not considered the possibility of cooperative development of software systems. The generated tool loads the document from the file-system during start-up and stores it before leaving the editor. There is no inter-document consistency constraints checking and the possibility of concurrent editing and viewing the same document by different developers is not allowed.

Reuse

Moreover, also reuse of *SSL* specifications is not supported by the language, since SSL has no language constructs for declaring component specifications.

8.3 Centaur

The *Centaur System* is a generic interactive environment which has been developed in the MENTOR-Project [DGK+84] that was carried out at INRIA between 1974 and 1986. This project is meant to support prototyping of languages, so it provides a framework which can be used to define a language in terms of syntax, static and dynamic semantics. It offers different languages for defining these aspects:

- METAL [KLM83] to define the abstract and concrete syntax;
- PPML (*Pretty-Printing Meta Language*) to define the unparsing scheme;
- TYPOL which is a rule-based language, used to define the static and dynamic semantics.

As we have seen for SSL, METAL defines the abstract syntax in terms of *phyla* and *operators*. Phyla define types of non-terminal nodes of an abstract syntax-tree, while operators define child nodes of the types of their arguments.

Concrete Syntax

The *Centaur System* splits the definition of the concrete syntax into the specification of the input syntax, defined in METAL, and the definition of the output syntax of an abstract syntax tree, specified in a dedicate language (PPML). A METAL specification is a collection of grammar rules, with annotations that specify what abstract syntax tree should synthesized. It is very similar to the production of a context-free grammar with the only difference that it adds the invocation of a tree building function which provides the mapping to the abstract syntax. Pretty-printing of abstract trees is then defined in the PPML specification, which is a collection of unparsing rules associated with abstract syntax patterns.

Besides, it is also possible to specify which part of the language can be edited in free textual input mode by the specification of *entry points*, which are also given in a rule based format.

Free Textual Editing

Generally, this kind of definition of a language syntax is not very concise. In fact, because of the distinction between abstract syntax, concrete syntax and entry points, the designer has to bother about the repeatedly definition of syntactic issues. In this way, the abstract syntax is also defined in the concrete syntax and in the entry points definition, changes to the syntax have their corresponding in updates of the METAL definition.

Considerations

Static and dynamic semantics are defined in TYPOL. The language enables definition of *Natural Semantics* which is based on *Structural Operational Semantics* [Plo81]. A TYPOL specification consists of a set of *axioms* and *inference rules* which define a formal system, where it is possible, to prove that a proposition holds. TYPOL specifications can be organized in separate modules. Moreover TYPOL provides powerful means for defining scoping rules and the type system of a language. Since the TYPOL compiler is realized by mapping a TYPOL program in a Prolog program, the designer can abstract from the execution order of the rules and state them in a declarative way.

Static and Dynamic Semantics

As we have seen for SSL, TYPOL does not support inter-document consistency constraints because the abstract syntax is strongly based on the particular language. Moreover, concurrency constraints cannot be expressed and both METAL, TYPOL and PPML specifications do not support reuse.

Inter-Document consistency and Concurrency

8.4 IPSEN

While the last two approaches are meant mainly to specify tools for a single language, *IPSEN* (Integrated Project Support Environment) project was intended for developing environments with highly integrated tools that support incremental and intertwined development of different documents. The used approach is based on attribute graphs to model and implement object structures such as different kinds of documents and their relationships. Therefore a graph grammars based language called *PROGRESS* (PROgrammed Graph REwriting SyStem) and a suitable method to apply this language have been developed since 1981 first at the University of Osnabrück and later at Aachen Technical University.

A *PROGRESS* specification defines abstract syntax graphs of documents. It consists of the definition of *node types*, *edge types*, *attributes*, a *start* node type and *graph rewriting rules*. Compared to attribute grammars, node types represent terminal and non-terminal symbols. The attributes serve the same purpose as attributes in the grammars. Graph rewriting rules represent productions and the corresponding semantic rules. Node types are defined in terms of *node classes* and node types, i.e. node type is

Abstract Syntax

defined by giving the name and defining the class which it belongs to. An edge type is defined by giving the name and the classes of the nodes that type starts from or leads to. Moreover, *PROGRESS* provides means to define inheritance between node classes. If a node class inherits from a super-node class, then all edges that start from the super node class may start from the subclass too. Differently from the other systems, *PROGRESS* does not provide any means for specifying concrete syntax or unparsing schemes.

Static Semantics

Usually, the aim of the static semantics is to restrict the set of the possible derivable sentences to those that are meaningful in the context of a particular project. For this purpose *PROGRESS* provides:

- *Derived attributes* as values attached to nodes; they are derived from values of possibly different nodes;
- *Path expressions* which define a set of paths through the graph;
- *Restrictions* as subsets of the identity relation which fulfills a particular condition.

Since restrictions and path expressions are not confined within the abstract syntax graph of a document, inter-document consistency constraints may be expressed as static semantics constraints.

Concurrent Accesses

PROGRESS does not support concurrent accesses of multiple users to the abstract syntax graph, but it only allows to bind applications of multiple graphs rewriting rules by *transactions*. The meaning of a transaction in *PROGRESS* is that either all rules are applied or none is.

Reuse

Moreover, PROGRESS does not support reuse of specifications since it is not possible to explicitly define relationships between different components of a specification.

8.5 GENESIS

The last project we consider is *GENESIS*. It is a project which suggests a number of domain-specific graphical and textual languages that a tool builder can use to develop the tool-specific component. The languages have been developed and evaluated within the ESPRIT-III Project (GOODSTEP) [GOO94] at Dortmund University, and are called GOODSTEP Tool Specification Languages (*GTSL*). These languages provide a library of pre-defined specifications for common properties of tools and provide the primitives for reusing and customizing these specifications. The specification languages follow the *Object Oriented paradigm* in the sense that the specifications are structured into component specifications, which are *classes* defining particular properties. These properties can then be inherited by other classes.

Abstract Syntax

For each class identified, *GTSL* defines its external and internal behavior separately, by the *class interface view* and *class specification view*. The internal structure of nonterminal increments is defined in the *abstract syntax section* in the interface of nonterminal increment classes. The abstract syntax is defined by enumerating names and types of son increments, in order to be able to refer the son in other sections.

Concrete Syntax

The mapping for input and output between abstract syntax graphs and the external representation is given by unparsing schemes, which are defined for non-terminal increment classes only. Differently from other systems which use different specifications, e.g. Centaur, in *GTSL* the unparsing section defines both the concrete syntax for which text edited with a text editor is checked, and the textual layout of the increment at the user interface.

Static Semantics

Static semantics as well as inter-document consistency are defined in a rule-based manner. The specification of each increment class may contain a *semantic rule section*, which consists of a list of semantic rules. Each rule defines a particular aspect of the static semantics and inter-document consistency constraints of an increment which has to be created or modified, and *collapse rule sections* where a collapse rule defines actions to be taken, if an increment is deleted. Moreover, differently from the other tools, *GTSL* includes a number of flexible mechanisms that can be used to define error messages. The strategy for accepting or rejecting erroneous input can be defined in interactions.

Concurrent Tool Execution

The last and the most important thing we want to underline is the fact that GTSL is the only language that deals with the specification of concurrent tool execution. The unit of concurrency control are the user-interactions, which may be delayed or even aborted in case of concurrency control conflicts to other concurrent interactions. Hence, interactions are performed in isolation. If one is completed, the effect of an interaction is durable, i.e. all changes that were made during the interaction persist even if the tool is stopped accidentally by a hard- or software failure.

8.6 Comparisons and Conclusions

Until now we have introduced several languages for tool specification on different levels of abstraction. Now we compare these systems in order to explain the rational for choosing *GTSL*.

We underline that none of the languages we have investigated except *GTSL* consider the specification of concurrent tools execution. This fact restricts their usability to projects with only a single developer, or that are small enough, that inter-document consistency constraints can be managed without tool support. This reason might be sufficient to motivate the choice of *GENESIS*. However, we think that this generator enables quite well the specification of the editor also on the other points of view, so we want to analyze also these points and show the reasons of our choice.

We have seen that the *Cornell Synthesizer Generator* is reasonably well suited for creating specialized syntax-directed textual editors which are tailored for editing a particular language, which do not support any inter-document consistency constraints and which may be only used by a single user at time.

Centaurs's Language TYPOL provides the same means perhaps even in a better way than *SSL*, specially for defining language semantics, even if its rules are not as concise as they could be. However, Centaur presents some drawbacks: first only static semantics correctness can be defined but not the actions to be taken in case of violation; second the specification of the used language has no inter-document consistency constraints and the designer has to bother about the consistency between the various specifications.

Differently from these two, *PROGRESS* reasonably well enables the definition of abstract syntax graphs and their integration. The language constructs that can be used for defining static semantics and inter-document consistency constraints are as powerful as those offered by *SSL*. The graphical components of *PROGRESS* specifications make the specification more comprehensive compared to TYPOL or *SSL* counterparts. However, there are also some weaknesses for *PROGRESS* since it does not support specifications of all tool concerns. In particular, it is not possible to define concrete syntax and unparsing schemes, nor it does not enable the definition of user-interactions. These last reasons make the tool unsuitable for our project.

On the contrary, another desirable feature of *GENESIS*, is that in *GTSL* classes are arranged into hierarchies, and this can be a benefit from the inheritance of several properties. For example in *GTSL*, static semantics rules are more concise than in the other languages, since they are inherited to subclasses. At the same way, the property of an increment of being freely editable is defined in an interaction which can be in turn defined in a common superclass and then inherited to all subclasses. In all the other languages dynamic semantics is formally defined, for instance in TYPOL it is based on Horn clauses, while here *GTSL* is weaker since its dynamic semantics has not been defined yet. However, this disadvantage is not so important for us, since our textual editor has not to support this task that is left to the Beta compiler. Instead, we think that the other features of the *GTSL* tool generator are able to provide powerful synthesis, analysis and manipulation resources while preventing counterproductive and meaningless transactions as well. *GTSL* provides also an external interface that makes it particularly suitable to be integrated with *Graph Project* and this makes the implementation of the Integration Mechanism easier. Moreover, as we have said for GP, the fact that the specification languages follow the Object Oriented paradigm, allows us to implement our textual editor complying the design shown in the section 6.2.

Chapter 9

Tool Implementation

In the previous chapters we have seen the editor generators which we have at our disposal to build our tool and how we think the mapping between OMT and Beta must be realized. This chapter shows the development process used during the realization of our tool. This process follows the design described in chapter 6. This has resulted in a realization which can be divided in the following three phases:

- Graphical editor realization (by means of GraphProject);
- Textual editor realization (by means of GTSL);
- Implementation of an integration mechanism between the two editors.

In this chapter we describe these phases in details.

9.1 Graphical Editor

Our graphical editor has been realized using GraphProject. As explained in section 7.3, the only things we had to implement were the terms of our graphical language and the methods associated with each term. So we did not have to bother about the configuration of the main windows with its MenuBar and about the other Dialog windows or MessageDialog windows yet, since these kinds of technical matters are fixed and automatically generated by the GP compiler.

The generation of the graphical editor was divided in two steps:

- The graphical language specification, in which we defined the editor interface;
- The implementation of the methods associated with each term of the graphical language, in which is defined the behaviour of the editor.

The rest of this section, first shows the higher level of our editor interface, in order to explain the environment in which the graphical language and the implementation of the associated methods are located, and then it explains the two steps mentioned before.

9.1.1 First Level of the User Interface

Before describing the first level of the user interface we want to underline that even if it useful to have an editor which is automatically generated by a compiler, at the same time this may also be seen as a restriction. In fact, for us the fixed configuration of the main window has been a restriction because we have not been able to add some features such as a help system support that we have considered a requirement of our tool in section 6.1.

Future Extensions

When the user calls the editor the first thing he - or she - sees is a main window like in fig. 9.1:

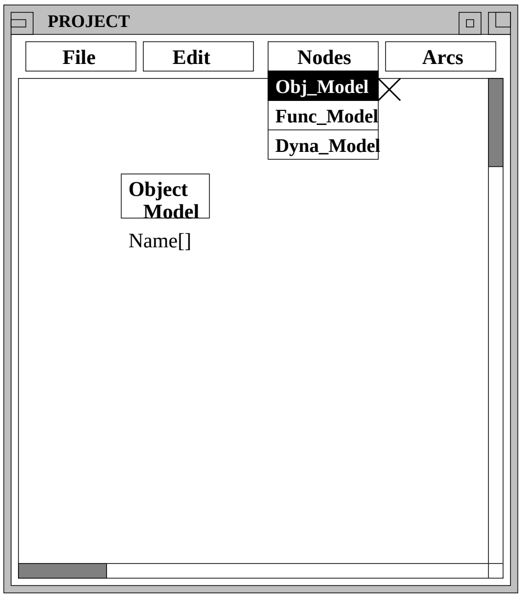


Figure 9.1: The Main Window

This figure shows that the user may choose to implement one of the three OMT models [RBP$^+$91], even if at the moment he can only choose to create one or more Object models clicking the relative options. If one of the options for the Functional and Dynamic models is selected, an appropriate message is displayed to inform the user that these options have been not implemented yet. In fact, we decided to structure this editor in such a way that someone can extent our work to support the graphical representation also of the other two models. In providing our tool with all the models, the respective graphical representations can take place within the framework of our editor without changes to the structure of the whole tool. Moreover, the realization of all the OMT representations might generate a more correct and complete Beta code.

The Menus

Each main window has a MenuBar with the following CascadeButtons: **File**, **Edit**, **Nodes** and **Arcs**. The Pulldown menu associated to the **File** CascadeButton has the following seven entries:

Info: provides informations for each entry of the menu;

New: allows the user to create a new diagram;

Load: to load any diagram previously saved;

Verify: to inform the user if the current graph is correct;

VerifyAll: to inform the user if the current diagram is correct;

Save: to save the current diagram;

Exit: to exit the editor.

The **Edit** Pulldown menu has two entries:

Move: which allows the movement of the selected object (node or arc) on the screen;

Delete: to delete the selected object.

The other two menus, for **Nodes** and **Arcs**, have an entry for each kind of node and arc defined, but we will see the terms of our language in the next section.

Selection

To select one of the entries of the menus above, the user has to click with the left mouse button on the appropriate cascade button and then, without releasing the button, on the desired entry position. If the user wants to launch one of the methods of the menu **Edit**, he has to select before the method and afterwards the object the method has to be applied to. Instead, if he wants to insert a new node or arc, he has to select the appropriate kind of node or arc first, and then to click on the screen to put it in the desired position.

GP Restrictions

For a more intuitive use of our graphical editor, we would have liked to use icons to represent each term of the language and also the functions to handle these terms. As shown in [CI94] this is not possible since all these interactions are realized by means of PullDown menus in which you can insert only labels. Moreover also the MessageDialog windows and the windows to take inputs from the users are predefined and generated by the compiler. This has been another limitation because we have not been able to define their size and the position where they should appear on the screen.

9.1.2 Specification of the Graphical Language

The interface of the graphical editor is defined by means of the terms' specification composing our graphical language (OMT). The OMT component terms are arcs and nodes [RBP+91]. In GP the specification of arcs and nodes is achieved by defining a class for each kind of arc and node. In each class we have then defined the graphical layout and declared the associated attributes and methods which define the state and the behaviour of the term defined by the class respectively. In this way each graphical object is also an object in the Object Oriented perspective.

Graphical Layout

To fully describe a graphical language we specify the graphical layout of the terms composing the language. The graphical layout of any node type is generated by means of bitmaps and by the definition of a label. Instead, for the definition of arcs layout, GP offers just a small set of possibilities which have been properly managed in order to realize the different OMT relationships, even with some occurring problems.

Defined Nodes and Arcs

Usually, in the OMT notation we have several different kinds of arcs (relationships), and two kinds of nodes (class and object). In our graphical editor the defined kinds of arcs are as many as the different relationships supported by the OMT notation:

Assoc_1_1: for One_to_One association;

Assoc_1_many: for One_to_Many association;

Assoc_many_many: for Many_to_Many association;

Ternary_ass: for ternary association;

Generalization: for the namesake relationship;

Aggregation: for variable aggregation;

Agg_mult: for aggregation with fixed multiplicity;

Instantiation: for the namesake relationship.

As we said before, in OMT there are only two kinds of nodes: *class nodes* and *instance nodes*. Since GP layout provides only few kinds of arcs and any kind of nodes, we have decided to implement some of the treated relationships by means of the composition of nodes and arcs (in this case we call these arcs *component arcs*). So we have defined six kinds of nodes:

Class: for drawing a class;

Instance: for drawing an instance;

Generalization: used as component node to draw the namesake relationship;

Aggregation: as *Generalization* is used as a component node to draw the namesake OMT relationship;

Ternary: used as component node to draw a ternary association;

LinkAttribute: used as node that contains the Link Attributes in associations with this facility.

The first two correspond to the OMT nodes. *Generalization*, *Aggregation* and *Ternary* are used to realize the namesake relationships and the node *LinkAttribute* has been used to realized associations with Link Attributes. In figure 9.2 some examples of the use of this kind of composed arcs are shown.

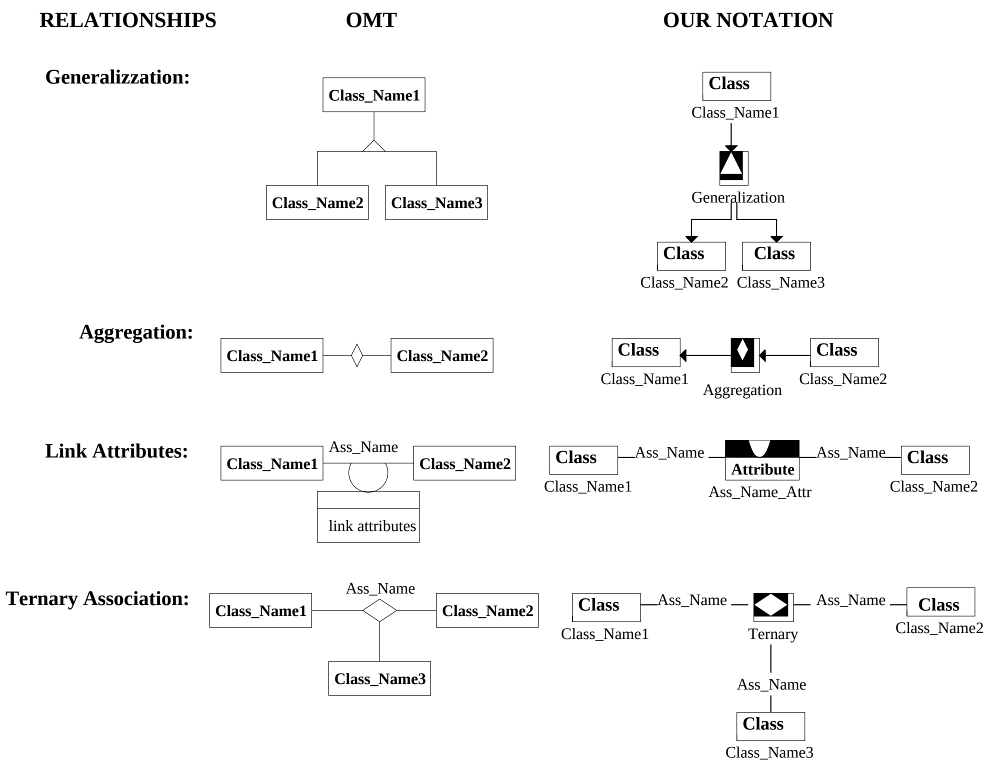


Figure 9.2: Example of Composed Relationships

As we can see, the label defined on a node is used to implement the class name, while for the other nodes they are used to identify the kinds of relationships. Moreover, the name of a class can be modified by the user while the other labels are fixed. Since GP does not allow to show more than one label for each node, it has not been possible to visualize attributes and operations inside a class. They have been associated to the class as attributes of the node and visualized in another window when the user requests it. The notation of the other relationships is realized by dividing the available symbols with the following result:

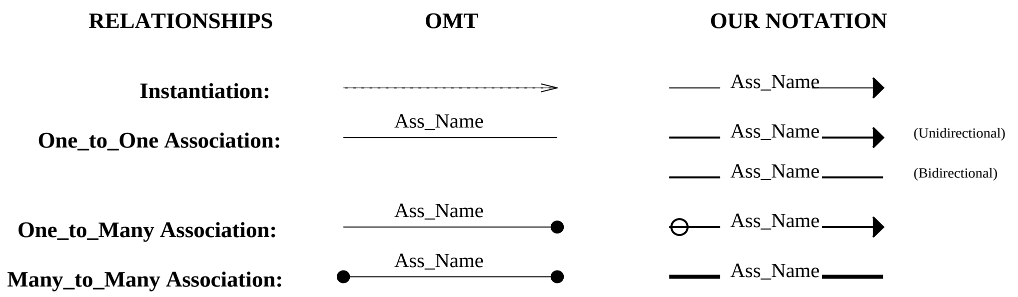


Figure 9.3: Other Relationships

As we can see, for some kinds of associations, in our notation we have used an arrow instead of a line. This implementation choice has been taken not only because of the GP graphical layout, but also because we needed a way to indicate the direction of the associations. In fact, OMT uses a natural language and the direction is implicitly given by the name of the association or by means of the role names. Since a tool can not understand the meaning of a label, an explicit direction is needed. Usually in an OMT diagram some other concepts are associated with arcs, such as the multiplicity and the concept of ordered in One_to_Many associations or the concept of role names. Since by means of GraphProject it is not possible to visualize these informations in the graph, as for the attributes in a class, we have realized these concepts as attributes associated with the arcs, and we have allowed the insertion and the display by means of windows. We will see this realization in the next section. The code for the interface specification is shown in the Appendix A.

9.1.3 Method Implementation

As we have seen before, each term of the graphical language is seen as an object which encapsulates its own state and which is able to react to messages sent to it. In this step we analyze the implementation of the methods declared in the interface specification. These methods define how the editor reacts to messages sent by the user. In designing a system, the user must be able to add, in his Object diagram, classes with their own attributes and operations, associations with their multiplicity, and so on.

Now we will see how we have defined the behaviour of the editor for each of the OMT concepts.

Classes

To create a *Class node* the user has to select the entry *Class* from the **Nodes** menu. Then he has to insert the name of the class. In our editor it is not possible to have a class without a name, because when the class will be translated into Beta code, it is necessary to have an identifier for the pattern implementing the class. In case the user omits to insert a name, the editor does not allow to create the class. On classes we have defined the methods to set and change the class name, as well as to handle attributes and operations. When the user, in refining his design, wants to perform one of these actions, he has to click on the class and the editor offers a pop-up menu, as shown in fig. 9.4.

In the pop-up menu all the operations that the user may select are listed. For example, if the user wants to add an operation, he has to select the entry *AddOperation* and the editor visualizes three windows for the insertion of the name and the result type, the parameters of the operation and a short comment related to the operation (as shown in the fig. 9.4). The insertion of the attributes is realized in a similar way. We allow the insertion of the attributes names, the types, and a related comments. Action to Add an Operation

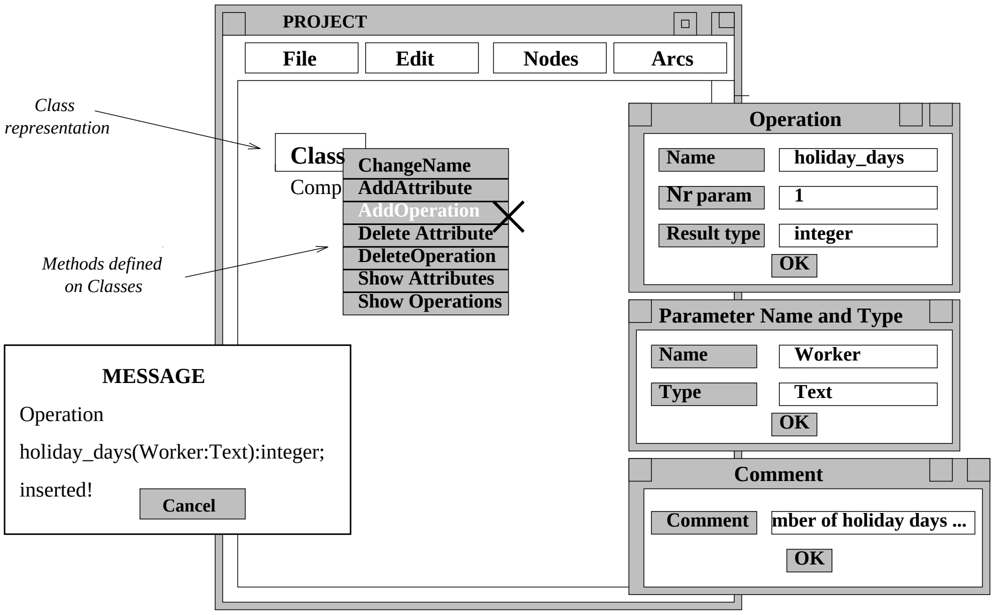


Figure 9.4: Realization of the Method AddOperation

The comments inserted during the operations *AddAttribute* and *AddOperation* are directly mapped in the Beta skeleton after the declaration of the corresponding inserted The Comment attribute or operation. As mentioned before, when the user wants to visualize the inserted attributes and operations he has to explicitly request it from the editor by selecting one of the methods *ShowOperations* or *ShowAttributes*. For instance, fig. 9.5 shows the window that appears on the screen when a depiction of the attributes of a certain class is required.

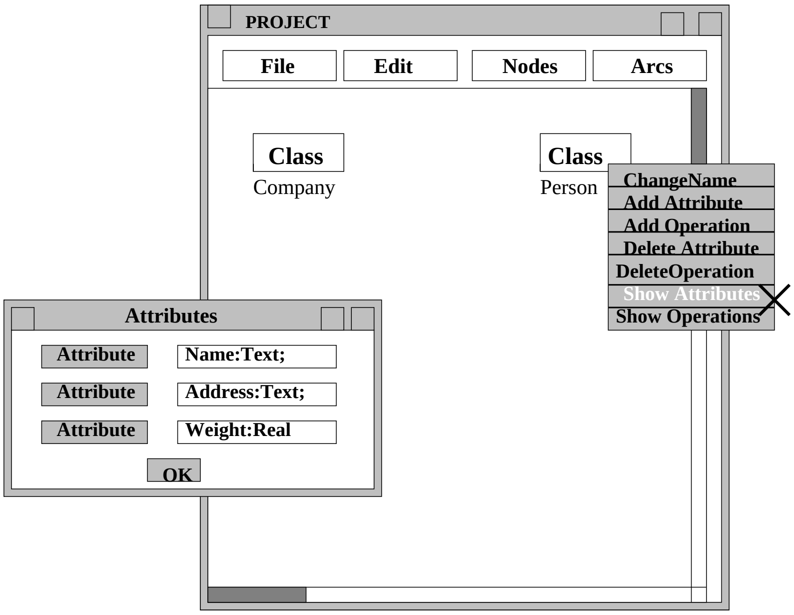


Figure 9.5: Realization of the Method ShowAttributes

If there are no defined attributes then a message is shown to inform the user about it.

The deletion of attributes and operations can be executed by selecting the appropriate entry in the menu **Edit** and then inserting the name of the attribute or operation the user wants to delete. The deletion must be confirmed in an input window.

Generalization

The user may also decide to organize classes using generalizations. To draw this kind of relationship the user has to insert the node *Generalization* first, and subsequently the necessary arcs of type *Generalization*. On these kinds of arcs and nodes we have not defined any specific methods, except the ones to handle the insertion and the deletion of the relationship.

In order to preserve a semantically correct graph, we have enabled the deletion only of those arcs that exit from the node *Generalization*. In addition, the deletion of the node *Generalization* itself causes the deletion of the whole relationship.

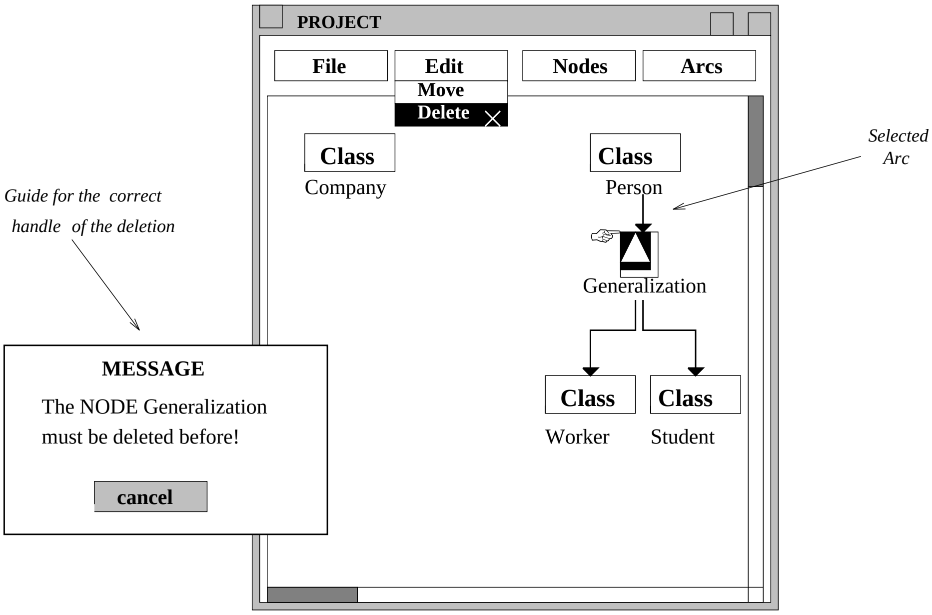


Figure 9.6: Deletion of a Generalization

So if the user tries to delete an arc entering the node *Generalization*, the editor ignores the command and displays a message to inform the user about the error. Fig. 9.6 shows this kind of situation.

Associations

Associations are all implemented in a similar way. The insertion is realized by selecting the entry corresponding to the particular kind of association the user prefers. Then, by clicking on the classes which must be involved in the association a new call to the corresponding method is made. As we mentioned before for classes, also for associations it is not possible to perform the insertion without setting a name. However, in this case, the corresponding method asks the user to insert the name of the association for more than once. If the name is not inserted after a certain number of requests, then the association is not inserted. Moreover, in OMT the direction of an association is implicitly given by the meaning of the name. Since our graphical editor can not understand this meaning, the user has to specify the direction. Therefore, during the insertion the editor asks the user also either the association must be bidirectional or not.

A Particular Case: the One_to_Many Association

The only kind of association on which we have defined more particular methods is the One_to_Many association, because this is one of the most complicated to handle. In fact when the user wants to insert this kind of association, it is not possible to set the multiplicity or make the association ordered directly by drawing these concepts. Therefore, the editor has to ask the user to define them by means of other windows. The editor asks the user for these informations by means of one of the defined methods that will be explained later.

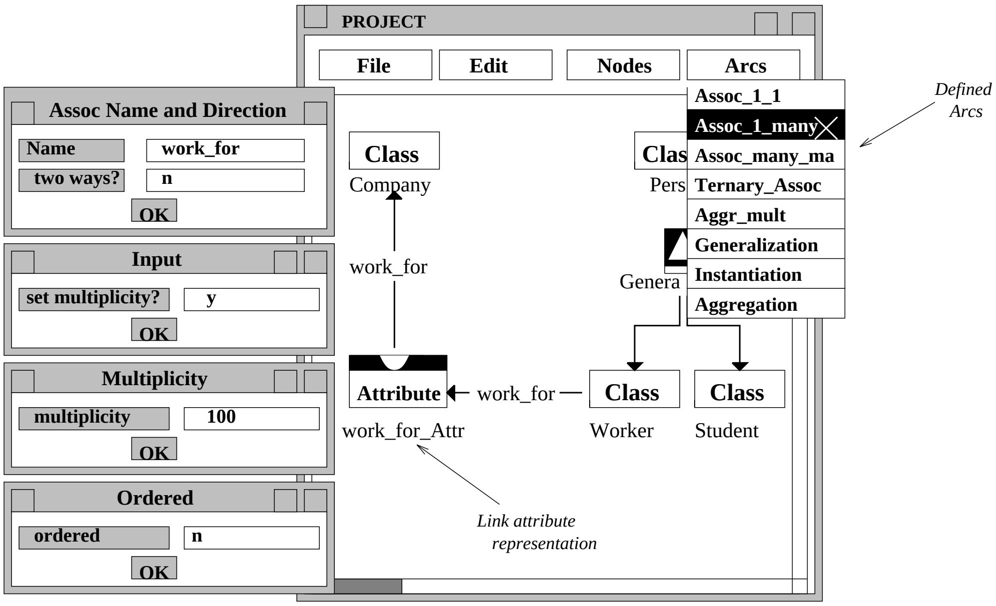


Figure 9.7: Insertion of a One_to_Many Association

Fig. 9.7 shows what happens when the user inserts a new One_to_Many association in a diagram and the corresponding method is invoked.

Link Attributes

It is also possible to define Link Attributes on associations. As we have seen, in order to allow the use of this kind of OMT concepts, we have defined a particular kind of node: *LinkAttribute*. This node contains the attributes of the association which are handled by means of the same methods shown for attributes on classes. To insert this association the user has to insert the node *LinkAttribute* before and afterwards, he has to connect this node with the classes involved in the association by means of two arcs of the appropriate kind of association.

Defined Methods

On associations we have defined the following selectable methods:

- *ChangeName*: to change the name of the association;
- *InsertRoles*: to allow the user to insert the role names;
- *ShowRoles*: to visualize the inserted (if any) role names;
- *ShowInfo*: (defined only for One_to_Many associations) to visualize the information associated with the association: multiplicity and if the association is ordered or not.

In case of associations with Link Attributes, all these methods assure that for each change made on one component arc the same change is also made on the other. In this way we can assure that the attributes of the arcs are always consistent and that it is possible to see the two component arcs as defining the same association.

Role Names

Another important characteristic of the OMT notation is the notion of role names. In OMT each role is associated with one end of the association and it identifies this end in an unique way. To allow the insertion and to display the role names we have defined the two methods *InsertRoles* and *ShowRoles*. Fig. 9.8 shows how the user can obtain these information from the editor.

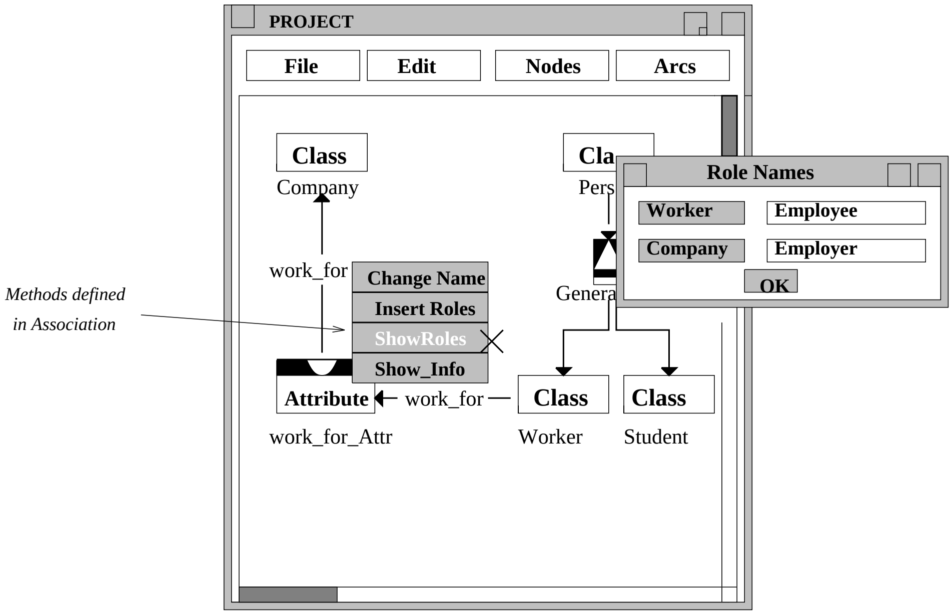


Figure 9.8: Realization of the Method ShowRoles

When the user selects the method *ShowRoles* a window appears on the screen to display the role names.

Aggregation

The last consideration is about aggregation. Similarly to generalization, to insert an aggregation the user has to insert the node *Aggregation* before and then several arcs of the same kind. Moreover, a user may also decide to insert an aggregation with multiplicity and therefore to set the multiplicity. The method corresponding to this insertion has been implemented in the same way as for the multiplicity in One_to_Many association.

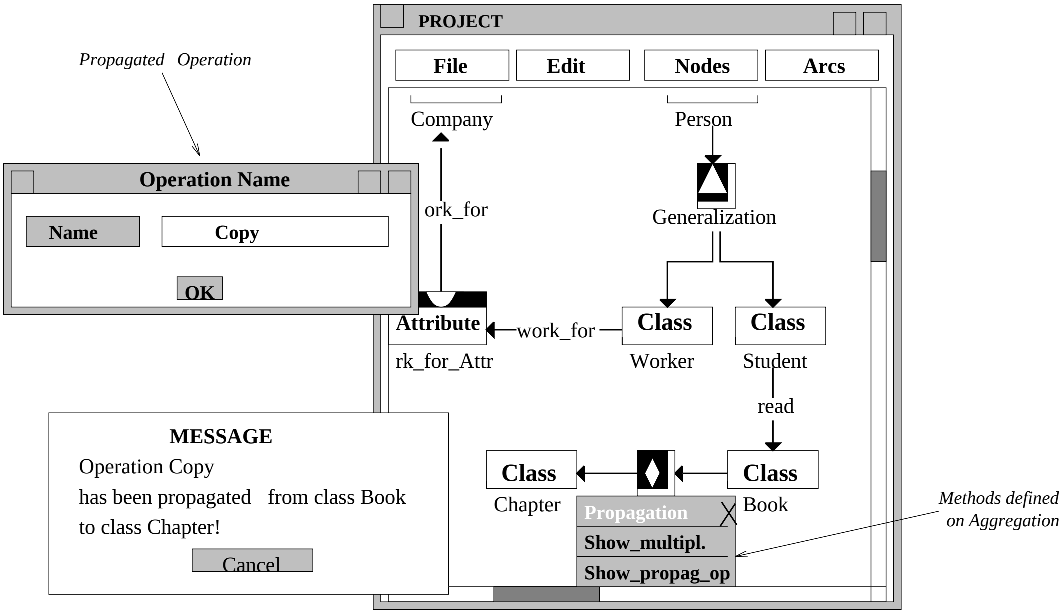


Figure 9.9: Propagation of An Operation

Propagation of Operations

Often aggregation offers the possibility to propagate an operation from an aggregate to the component classes. In the figure above is shown how the user may propagate the operation *Copy* from the class *Book* (the aggregate) to the class *Chapter* (the component).

9.1.4 Summary

In this section we want to analyze the main characteristics of our graphical editor. In chapter 6 we gave some requirements we wanted for our tool and that we have tried to follow during the implementation. Now we show the results of this first phase, by comparing our editor with the given guidelines.

First of all, since GP is a syntax-driven editor generator, our graphical editor is syntax-driven. Moreover, the tool supports communication of application action to the user. For example, when a not applicable menu option is selected, the tool shows an error box with an appropriate message and a request to acknowledge the error is performed. Any operation that produces an irreversible result must be explicitly confirmed and the selection of operations used to insert data in the Object Model (as the insertion of an attribute) is followed by a message to inform the user about the executed operation.

Unfortunately, since the configuration of the main window is fixed , we have not been able to add a CascadeButton for the **Help** functionality and therefore, to allow our tool to support this feature. Moreover, for the same reason, we have not been able to use panel with icons, because all the entries are implemented by menus in which only labels can be inserted.

During the construction of our editor we had some problems in exactly representing the OMT notation because of the lack of the GP layout which does not offer so many possibilities. We analyze this problems to underline some implementation choices and their consequences.

A first problem was in drawing some kinds of arcs, such as associations with link attributes, aggregation, generalization and ternary association. Since the GP layout does not allow the user to draw these kinds of arcs as they appear in the OMT notation in real, and does not offer many different kinds of arcs, we have implemented these relationships by means of the composition of nodes and arcs. The consequences of this decision are first, that in this way the user interface has become slower to use, and secondly that we have had to define additional methods to handle these arcs as if they were fiscally one link. Also the notation of the various kinds of associations has been changed in order to indicate the direction of the drawn association. This was necessary because obviously a tool can not understand the direction from the meaning of labels.

We have also changed the OMT notation by the introduction of two different levels of abstraction in order to visualize:

- the attributes and the operations in classes and link attributes;
- some additional notations that OMT defines on arcs, such as multiplicity or the concept of ordered associations.

Despite of these problems, in our opinion the graphical editor resulted rather efficient since we have been able to represent all the OMT concepts we wanted to map and to give a final graphical representation of an Object Model that is really near to the OMT notation.

Adding an abstraction level to the OMT notation has contributed to make clear the representation given by the tool for the Object diagrams. We think that for extended systems, it is not useful to show all the attributes and operations for each class, especially if the number of these features is high. Putting attributes and operations in a second abstraction level, we allow a better view of the drawing design and the user can see his design in a more abstract level. This kind of representation, that in our case has been enforced by the GP layout, is instead a choice for other OOAD methodologies, such as BON and Booch.

Anyhow, in this editor we allow the user to insert some documentation (comments) that other OMT tools, we have seen, do not allow. In this way, the developer can write down the decisions taken during the designing phase and then later find again these decisions written in the textual representation in order to complete the implementation phase in a more consistent way.

The last but most important feature of our editor is the possibility to be extended to cover also the implementation of the other OMT models. This has been possible because, as said in section 7.3, the resulting editor is completely implemented in an Object Oriented way. So it will be possible to add modules that will implement the handle of the other two models.

9.2 Textual Editor

In this section we are going to describe the main features of our textual editor. As we said in chapter 8, we have realized our textual editor by means of GTSL [GOO94].

According to the *Object Oriented paradigm* the specification of the editor is structured in component specifications, where each component is a class defining particular properties of an *increment* of a given grammar, i.e. a syntactical unit of the language to be implemented. The declaration of a class is in turn split into two parts: the interface and the specification. The interface defines the resources exported from the class that may be used in other classes while the class specification defines those aspects of the class that need not to be known from outside, such as the implementation of the methods.

Development Process

Our textual editor development process has followed different phases which are automatically guided and supported by the GENESIS environment. The first step in building a tool is to individualize its structure. In GTSL a partial structure is given by the syntax of the target language, which is defined by means of its grammar written in terms of some normalized EBNF (Extended Backus Naur Form). Then the classes which compose the editor specification need to be identified to define the increments of the language. Their inheritance relationships are then derived from the grammar and displayed in a class hierarchy overview. At the same time also the Entity/Relationships view is automatically generated from the grammar specification to determine the syntactic relationships among the classes. Moreover, the increment modifications and the user interactions must be defined to allow the user to invoke particular operations on increments.

Hence the different steps followed during the construction of our textual editor have been:

- Developing a new version of the Beta grammar;
- Derivation of the inheritance view;
- Derivation of an Entity/Relationship view;
- Implementation.

From the first three steps the classes interfaces and a part of the classes specifications have been automatically generated by means of the GTSL tools support. Therefore the last step in the realization of the textual editor was the declaration and implementation of the methods and user interactions. Because of the big number of generated classes, it is not possible to describe the specification of our editor by the description of each class, as we have done for the graphical editor. Then we will presents our textual editor by the description of the four steps defined above.

9.2.1 A New Version of the Beta Grammar

To obtain a comprehensive structure of the textual editor specification, this should be structured according to the structure of the grammar of the language the editor is intended for (in our case the target language is Beta). For this reason we have studied the grammar formalism used in the Mjolner Beta System [LMN93]. This formalism is often more complicated than it should be necessary. Moreover, using this grammar, incorrect Beta programs can be generated. Therefore the given grammar has been transformed into a more correct grammar written in terms of a normalized EBNF. The normalization of the EBNF has been enforced by the fact that in this way it is possible to map the grammar into a GTSL specification. Moreover we think that the new formalism also simplifies the comprehension of the grammar.

Grammar Transformation Process

The grammar transformation process consists of three steps. The first step has been the correction of the grammar taken from [LMN93]. Then the correct grammar has been transformed in a normalized EBNF which has been used to generate the textual editor specification. In the last two steps, we have been supported by the GENESIS environment which includes a syntax-directed editor to edit the grammar and then another tool to automatically generate a significant amount of the textual editor specification.

Figure 9.10: Grammar Transformation Process

Fig. 9.10 shows how, from a given correct grammar, a skeleton of the whole specification is produced. Besides, during this two steps from the grammar specification, a first classes interface definitions, the inheritance view and also the entity/relationship view were generated. Therefore both the views were then refined during the next two phases in order to produce a complete specification of the classes interfaces.

The rest of this section will explain the first step of the grammar transformation process, i.e. the corrections we have applied to the given Beta grammar. The rules of the used formalism and the new complete grammar (with some useful productions added to integrate the textual and the graphical editor) are given in the Appendix B.

Main Program

In the Mjolner Beta System the building of each fragment begins from the following productions:

$$\begin{array}{ll} < BetaForm >::= & \mid < DescriptorForm > \\ & \mid < AttributesForm > \end{array}$$
$< DescriptorForm >::= < ObjectDecsriptor >$
$< AttributesForm >::= < Attributes >$
$< ObjectDescriptor >::= < SuperPatternOpt >< MainPart >$

One might suppose to use $< BetaForm >$ as the root of the grammar, but in this way programs like the one below can be also generated:

$$\begin{array}{lll} < SuperPattern > & (\# & < Attributes > \\ & \textbf{enter} & < Evaluation > \\ & \textbf{do} & < Imperatives > \\ & \textbf{exit} & < Evaluation > \\ & \#); & \end{array}$$

which is not correct if it is considered as a main program. In fact using the given productions we can also produce the nonterminals *SuperPattern*, *EnterPart* and *ExitPart* (which are declared to be optional). However, usually a main program cannot have an *EnterPart* or *ExitPart*, and it is never a subpattern of any other pattern.
For these reasons we have changed these productions in the following way:

$< BetaProgram >::= (\# < Attributes >$ **do** $< Imperatives > \#)$

Labelled Imperatives

For the *LabelledImperatives* the following productions are originally defined:

$$\begin{array}{ll} < Imp >::= & \mid < LabelledImp > \\ & \mid < LabelledCompoundImp > \\ & \mid < LeaveImp > \\ & \mid < RestartImp > \\ & \ldots \end{array}$$

These productions allow the use of the *Leave-Imperative* or the *Restart-Imperative* as if they would be common imperatives without any connection with the other labelled imperatives. Hence it is possible to produce programs which are semantically incorrect. For example it is allowed to produce:

$(\#$ $\quad < \textit{Attributes} >$
do $\quad (L : \ldots \{\textit{list of imperatives}\} \ldots : L);$
$\qquad$ **leave** L
$\#);$

where the LeaveImperative **leave** L has no meaning outside the LabelledCompoundImperative $(L :\ldots \{\textit{list of imperatives}\} \ldots : L)$;

The productions were changed in the following way:

$$
\begin{array}{ll}
< Imperative >::= & | < LabelledImperative > \\
 & | < LabelledCompoundImperative > \\
 & \ldots
\end{array}
$$
$$< LabelledImperative > ::=< NameDcl >:< LabelImperatives >$$
$$
\begin{array}{ll}
< LabelledCompoundImperative >::= & (< NameDcl >:< LabelImperatives >: \\
 & < NameDcl >)
\end{array}
$$
$$< LabelImperatives >::+ < LabelImperative >$$
$$
\begin{array}{ll}
< LabelImperative >:: & | < LabelledImperative > \\
 & | < LabelledCompoundImperative > \\
 & | < LabelledFor > \\
 & | < LabelledIf > \\
 & | < LeaveImperative > \\
 & | < RestartImperative >
\end{array}
$$

We have restricted the use of the *Leave-Imperative* or the *RestartImperative* only in a *LabelledCompoundImperative* or in a *LabelledImperative*. This has been done underlining the difference between labelled imperatives and what is not of this type.

Alternatives in the If-Imperative

In the original grammar the *If-Imperative* structure has the following form:

(**if** $< Evaluation >$
$// < Evaluation_1 >$ **then** $< Imperative_1 >$
$// < Evaluation_2 >$ **then** $< Imperative_2 >$
$\ldots$
$// < Evaluation_N >$ **then** $< Imperative_N >$
else $< Imperative >$
if)

The grammar described in [LMN93] generates this kind of structure by means of the rules:

$$
\begin{array}{ll}
< IfImperative >::= & (\textbf{if} < Evaluation >< Alternatives > \\
 & < ElsePartOpt > \textbf{if})
\end{array}
$$
$$< Alternatives >::+< Alternative >$$

$< Alternative >$::= $< Selections >$ **then** $< Imperative >$
$< Selections >$::+ $< Selection >$
$< Selection >$::=—$< CaseSelection >$
$< CaseSelection >$::=// $< Evaluation >$
$< ElsePartOpt >$::?$< ElsePart >$
$< ElsePart >$::=**else** $< Imperatives >$

These rules are more complicated than necessary, because if it is possible, it is better to generate the same structure by means of simplest productions. The reason for this choice is that the number of classes in our editor grows with the number of productions. For this reason we have changed the productions above in the following way:

$< IfImperative >$::= (**if** $< Evaluation >< Alternatives >$
$< ElsePartOpt >$ **if**)
$< Alternatives >$::+$< Alternative >$
$< Alternative >$::= $< Selection >$ **then** $< Imperative >$
$< Selection >$::=// $< Evaluation >$

This reduces the number of the productions to a half.

Index

In Beta we need to use an index in two different situations: when we are declaring a repetition or when we are using a for-imperative. A repetition is declared in the following way:

A : [*eval*] Ref;

where *A* is the name of a repetition of a static or dynamic reference, *Ref* is a declaration of a static or dynamic pattern and *eval* is an evaluation resulting in an integer number. In the original grammar for this construct we have found the rule:

$< RepetitionDecl >$::= $Name$: [$< index >$] $< ReferenceSpecification >$

Instead, the for-imperative has the following form:

(**for** index:Range **repeat** Imperative-list **for**)

where *index* is the name of an integer-object and *Range* is an integer evaluation.
In the given grammar the corresponding rule is:

$< ForImperative >$::= (**for** $< index >$ **repeat** $< Imperatives >$ **for**)

Now, looking at the productions describing index, we find the followings:

$$
\begin{aligned}
< index > ::= \quad & | < SimpleIndex > \\
& | < NamedIndex > \\
< SimpleIndex > ::= & < Evaluation > \\
< NamedIndex > ::= & < NameDcl > : < Evaluation >
\end{aligned}
$$

In this way, we can also use a *SimpleIndex* in a for-imperative or use a *NamedIndex* in a repetition, but this is not correct at all, while it is only allowed to use *SimpleIndex* in a repetition and a *NamedIndex* in a for-imperative. For this reason we have decided to omit the rule for *index* and to use directly *SimpleIndex* and *NamedIndex* in the rules for repetition an for-imperative, as shown below:

$$< RepetitionDecl > ::= Name : [< SimpleIndex >] < ReferenceSpecification >$$
$$< ForImperative > ::= \quad (\textbf{for} < NamedIndex > \textbf{repeat} < Imperatives > \textbf{for})$$

Assignment

In the given grammar an evaluation specifying an assignment is described by the following productions:

$$
\begin{aligned}
< Evaluation > ::= \quad & | < Expression > \\
& | < Assignment > \\
< Assignment > ::= & < Evaluation > \quad -> \quad < Transaction > \\
< Transaction > ::= & \ldots
\end{aligned}
$$

These rules can easily produce a correct evaluation like the one below:

$1- > P1.move$

but often a dynamically generated object is needed to describe systems where new objects are generated during program execution. In this case you need to realize something like

$\&T[]- > A1[]$

where *A1* is the name of the dynamic reference of T, i.e. *A1* is declared as *A1* : ^T.
By means of the productions above, this cannot be achieved because on the left side of an assignment we can only use an expression or another assignment, while we need to use something like a transaction.
So we have changed these productions in the following way:

$$
\begin{array}{ll}
< Evaluation >::= & | < Expression > \\
 & | < Assignment > \\
< Assignment >::= < Transaction > \quad -> \quad < Transaction > & \\
< Transaction >::= & | < ObjectEvaluation > \\
 & | < ComputedEvaluation > \\
 & | < ObjectReference > \\
 & | < EvalList > \\
 & | < Evaluation > \\
 & | < StructureReference >
\end{array}
$$

We have substituted an *Evaluation* with a *Transaction* in the production for an *Assignment*, and we have added the *Evaluation* to the alternative rule for the *Transaction*.

The given grammar modified is the version used to proceed in the grammar transformation process. In this section we have only shown the corrections made to make the grammar able to generate more syntactically correct programs. Moreover, we have also applied some useful modification to this grammar (see Appendix B) in order to implement our mapping from OMT. For example we have divided the definition of a pattern in the definition of a simple pattern, a procedure pattern and a functional pattern. This allows us to leave the *do part* from the syntax of the basic pattern that is used to represent classes and the *exit part* from the syntax of the procedure pattern that is used to represent operation without result value. In this way the resulting grammar is not able anymore to generate all the possible Beta programs but it generates only programs written in the Object Oriented programming style.

9.2.2 Inheritance View

The specification of the tool resulting from the grammar transformation process was still incomplete, but this skeleton has been however a good starting point for our tool specification process since an initial GTSL class hierarchy can be derived.

In fact, the Inheritance View displays all our tool specific classes and their inheritance hierarchy. In addition, also all those pre-defined classes that participate in inheritance relationships with tool specific classes are included. In this way it has been easy to identify which class inherits properties from other classes and to refine the generated view by modifying the inheritance relationships or adding some useful abstract classes.

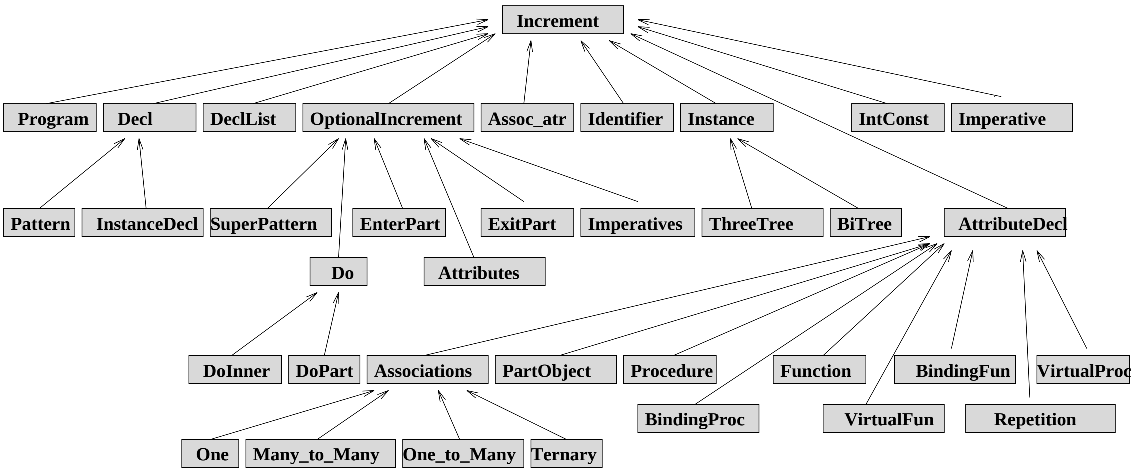


Figure 9.11: Inheritance View of Our Interface Editor Specification

Fig. 9.11 shows the generated Inheritance View for our textual editor. This view has been afterwards modified by adding some other classes with a graphical syntax editor provided by the GENESIS environment. For example, one of the later added classes is the *DoInner* class that has been created to allow the transformation of a pattern into a virtual pattern. This transformation is necessary when an operation, specializing one implemented by the pattern, is graphically added in a generalization and, as result, has to become abstract. In fact, this causes also the insertion of an INNER in the do part of the transformed virtual pattern. The variations in the inheritance view by means of the editor have also caused a modification of the tool configuration as well.

9.2.3 Entity/Relationship View

An Entity/Relationship (E/R) view document is generated together with the tool configuration document. This view displays the names of abstract syntax children and the links of semantic relationships between them. The E/R view is automatically generated and refines the specification by modifying the inheritance relationships and the semantic links. As for the inheritance view this can be modified by an editor but in this case is not allowed to add or delete classes. The notation used to visualize this kind of representation is an OMT-like notation used in the GENESIS environment. Classes are depicted as rectangles, semantic relationships are drawn as arrows, while the relationships between classes and their abstract syntax children are represented by means of the generalization relationship defined in the OMT notation.

In fig. 9.12 a part of the graph generated from our specification is shown. This figure describes the part relative to the declarations. Here *AttributeDecl* is an abstract class which is semantically related with the class *Identifier*. Each of its abstract syntax children is also related with the class Identifier (the relationship is inherited from the abstract class to its children) but each of them may also adds its own semantic relationships.

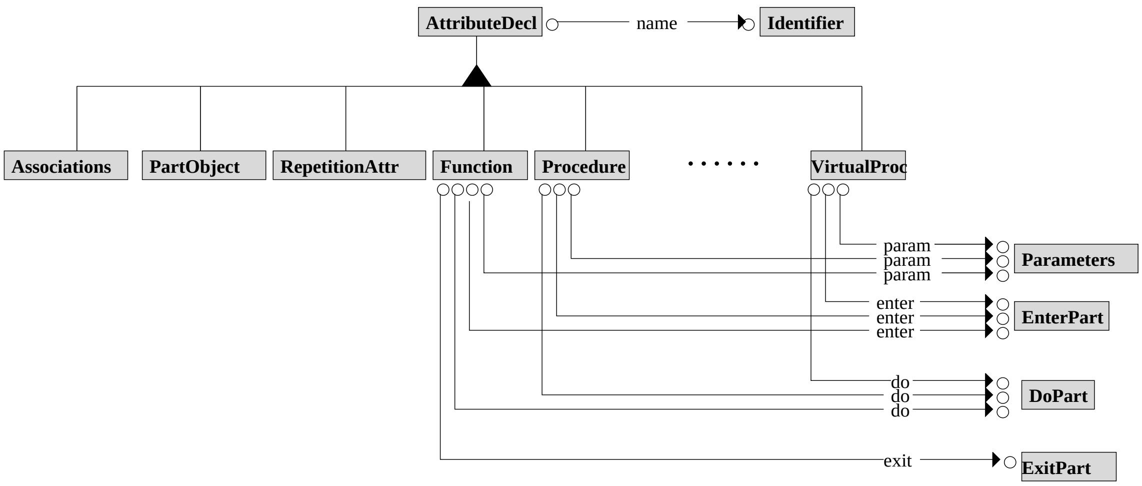


Figure 9.12: E/R View of The Interface Editor Specification

For instance, the class *Function* is also related with the classes *Parameters*, *EnterPart*, *DoPart* and *ExitPart*.

9.2.4 Implementation

The last step during the specification of a textual editor is the final implementation of all the classes generated during the previous steps. The resulting textual specification of the editor is composed by the interface of each generated class and by the specification of all the declared methods and user interactions. On the contrary from some other textual editors, in our editor the definition of the user interactions is independent from the implementation of the methods. Straightaway, the user interactions are less than the methods to distinguish the action performed by the user on the textual editor from those that came from the graphical editor. This allows to avoid the user to erase the parts of the code that is automatically generated. This code can only be erased by deleting the correspondent OMT concept drawn with the graphical editor. In this way the consistence between the graphical and textual notation is ensured, and moreover, the problem about the consistence of the associations individualized during the mapping (see section 5.1) is also solved.

When we were looking for a textual editor generator, we have identified some requirements that our tool should have. there were:

- Ability to assist the user in producing syntactically correct documents;
- Let the user decide to use the free textual editing mode or not;
- Checking static semantic constraints;

- Easy integration with the graphical editor.

We will now explain, by means of some examples, how we have realized the first three requirements. The integration of the textual and graphical editor will be discussed in the next section.

Syntax-Directed Editor

In section 6.1 we have required our tool to be directed towards the language it is intended for. This means that the tool should always be able to preserve the syntactically correctness of the program the user is developing.

As an example consider the window of the textual editor corresponding to the operation *holiday_day* we have inserted by the graphical editor at pag. 103. After the insertion of the operation in the graphical model, the window of the textual editor displays the following situation. Example

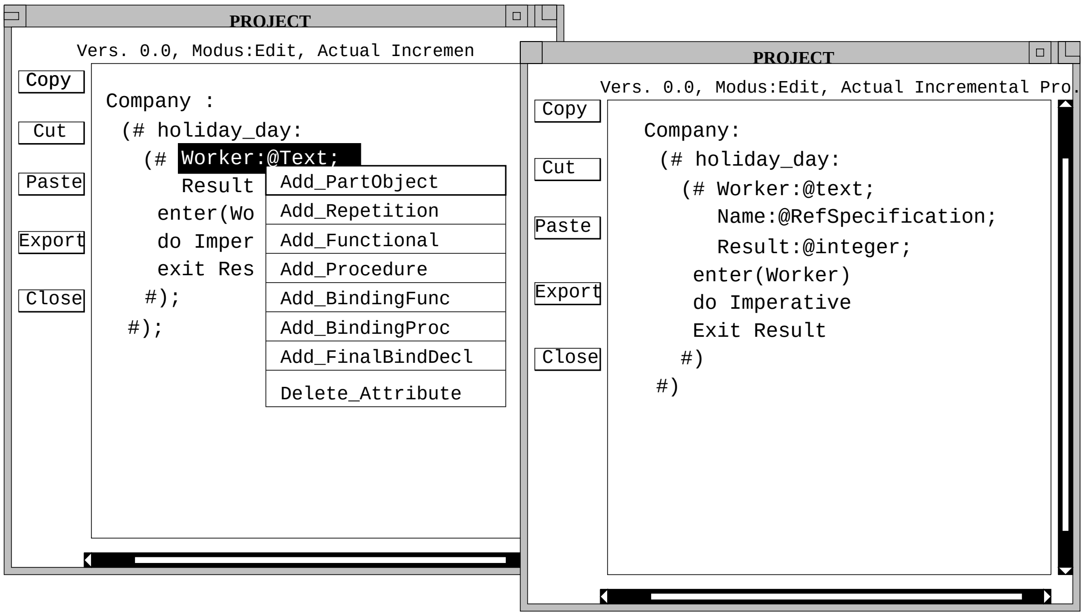


Figure 9.13: Replacement of a Template in a Syntax-Driven Mode

Fig. 9.13 shows what happens when the user selects for instance the parameter *Worker* in the inserted operation. Pushing one of the mouse buttons causes the tool to deduce a set of possible commands applicable to the selections that are offered in a pop-up menu. The user may select a command, for instance in this example *Add_PartObject*. The implementation of the corresponding method causes the tool to insert a new *PartObject* template after the current increment. The user will be able to replace this template by means of the editor.

Syntactic Correctness

As this example suggests, syntactical correctness is always preserved because the tool only inserts syntactically correct templates. In this way, only those commands are offered whose execution cannot violate the syntax definition.

Free Text Input

Sometimes the structure oriented mode of editing a document is not so appropriate. For instance, experienced users who master the language are capable of typing very fast. Such users may be faster in typing an increment then repeatedly selecting this increment. To address this kind of problem, users expect from the tool not only support for structured-oriented editing, but also facilities to freely edit increments with conventional textual editors.

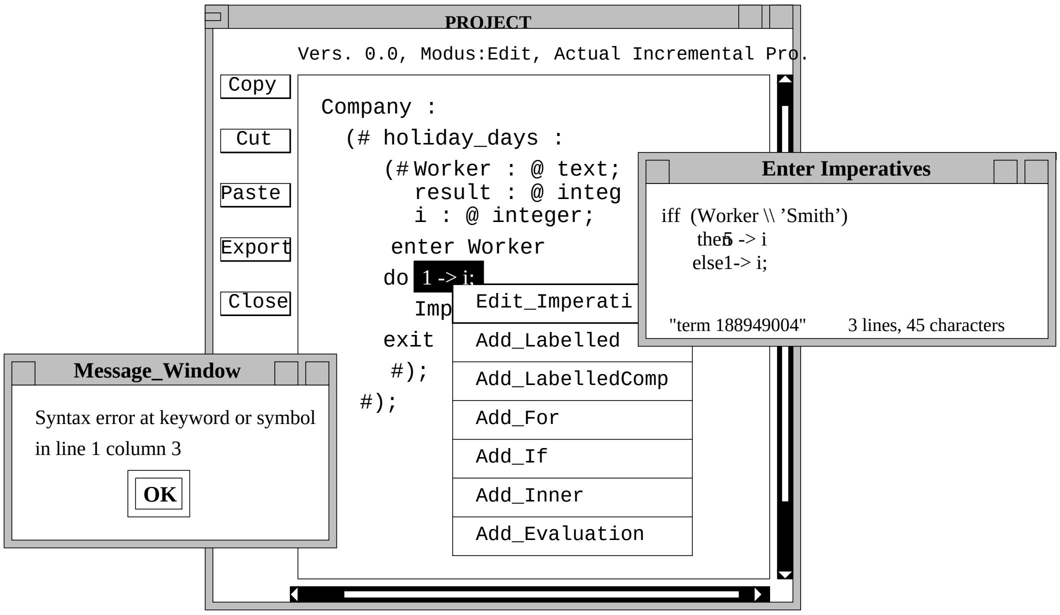


Figure 9.14: Free Textual Input Mode

Example

As an example we consider a statement which must be transformed into an if-statement without affecting the body. When the user chooses the interaction *Edit-Imperative*, the editor shows a textual representation of the selected statement and presents this representation in a Unix text editor. Then, to ensure the static correctness the tool parses the new text as soon as the user has finished editing. If the text contains syntax errors, then the tool displays a message to inform the user about them. Fig. 9.14 shows what happens if the user makes a mistake and writes "iff" instead of "if".

Static Semantic Correctness

Beside syntactic correctness of documents, our textual editor can also support the user in achieving correctness of the static semantics. Static semantics of languages, for instance, requires that each type used as a parameter type or as a result type is declared. Since it is possible in Beta to use features which are defined below, handling these kind of errors can not be defined. However, we have thought that it would be appropriate to temporarily allow such errors, but to draw the user attention on them. We achieve this in our editor by underlining errors with messages.

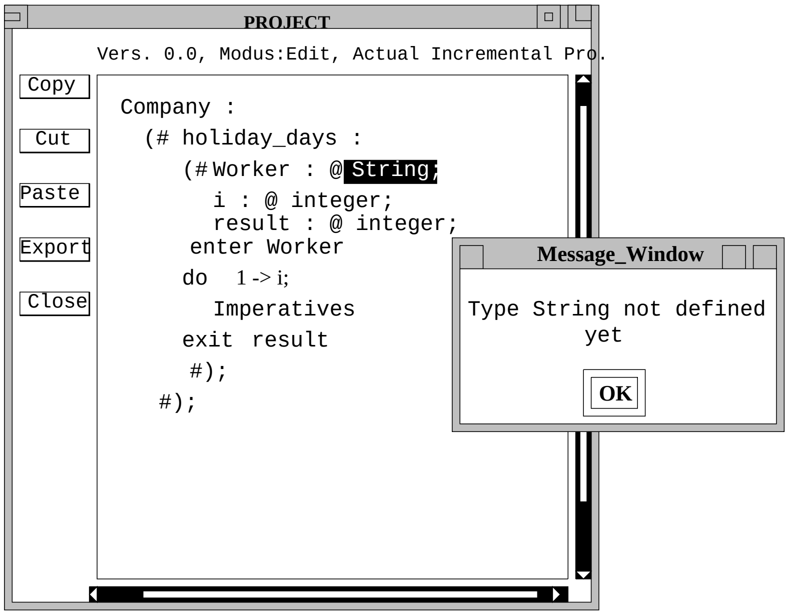


Figure 9.15: Handling of Semantic Errors

For instance, in the figure above, we have defined the type of the parameter *Worker* as a *String*, which is not defined in Beta as a basic pattern. In this situation we are not able to say if this is a semantic error since the pattern string might be defined later during the development of the system. In this case the corresponding method causes the editor to display a message window to inform the user that the type string has not been defined yet.

9.2.5 Summary

This chapter has shown our experience in building the textual editor by means of GENESIS environment. In our opinion the generated editor is quite good, even if the whole development process has been long and the production of the specification has required much work. In fact, since the specification of the editor is realized by declaring the interface and the specification of all classes, we had to edit a great number of files. The presence of many files allows the development of such an editor concurrently by many developers, but it becomes disadvantageous when the number of the developers is not great at all (as in our case). The same feature, however, makes the specification reusable and allows to separately compile the different classes.

Also another problem has been caused by the GTSL compiler. The configuration compiler translates the GTSL specification into C^{++} code and then all the compiled C^{++} classes are linked together with the GTSL library. The result is that it is not possible to declare in the specification identifiers that are also C^{++} keywords. This has caused some problems because, since we did not know this features of the GTSL compiler, we used as identifiers some Beta keywords that unfortunately were also C^{++} keywords. This has led to a long and tedious revision of most of the specification.

Now we analyze the results of the editor specification. As we have done for the graphical editor, also here we compare the main features of the generated editor with the requirements given in chapter 6. We have already analyzed those requirements which are only proper of the textual editor, therefore now we will consider those features that belong to the complete tool.

First of all a requirement of the tool was to handle both the graphical and the textual editors in order to always have the two documents consistent. Since any change to the structure of the system has to be realized from the graphical representation, the user is allowed to perform by text only those actions that are concerned with the implementation of the system and not with its specification. Moreover, since the deletion of most of the code has to be realized by means of the graphical editor, the recover from unintended operations is ensured as well. In fact it is possible to delete the code inserted by the user and not the part automatically generated by the editor.

Moreover, we have required our tool to be syntax directed. For this purpose we have given a new grammar and structured our editor according to this grammar. In order to prove the correctness of the generated grammar we have also built a parser in "yacc" and we have tested the grammar parsing some correct and incorrect programs. In this way we cannot be sure of the correctness of our grammar, but however there is no other way to prove it.

9.3 Tool Integration

The integration between the graphical and the textual editor is realized by means of a Communication Protocol Subsystem which provides our tool with a communication protocol for sending messages from the graphical editor and receiving service requests by the textual editor. We use this kind of integration mechanism, instead of operating system primitives, because in this way the basic communication mechanism has been hidden from the rest of the tool architecture. This allows to arrange for portability and provides a dedicate, safe and application-specific protocol for the communication between the two editors. Our integration mechanism has been implemented using the Communication Protocol Subsystem offered by the GENESIS environment. In this section, the architecture of this subsystem will be described in order to identify the structure of our integration mechanism. Then we will give a description of the main features of the defined messages and some examples showing how the implementation of the mapping is realized by means of the communication mechanism.

9.3.1 The Communication Protocol Subsystem

Architecture of the Communication Protocol Subsystem

We now want to describe the architecture of the Communication Protocol Subsystem and to explain how it works. In the architecture of our textual editor there are some external components, such as the Communication Protocol Subsystem, which need to be integrated with the other (internal) components. The integration has been realized by means of a single class, called Control class. This class is the supervisor of all the communications in the Communication Protocol Subsystem. The *Channel* class provides instead the channel for all communications between the two editors. The textual editor has an object *Channel* stored in an instance variable of the textual editor Control class. After the creation of this Control class, also an instance of the communication channel is generated to allow the textual editor to receive service requests from the graphical editor. Service requests as well as events are represented as messages. When the graphical editor invokes a service request, this request arrives at the Control class. The Control class in turn calls an operation to the Communication Protocol Subsystem in order to obtain a message object representing the particular service. The service then executes the request invoking some methods on classes defined in the textual editor.

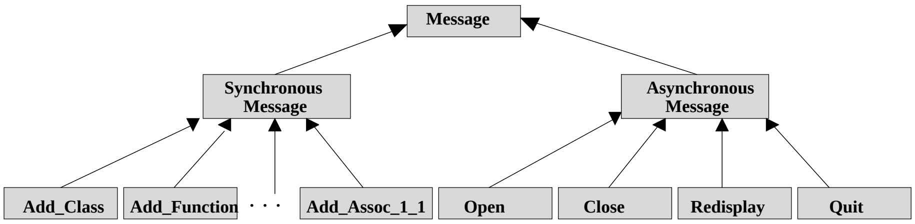


Figure 9.16: Message Subdivision

Synchronous and Asynchronous Messages

As shown in fig. 9.16, messages are divided into synchronous and asynchronous messages. Those messages were already defined as well as the Control and the Channel class. Asynchronous messages are used to implement operations like *Open* or *Close*, where the graphical editor does not need to receive any acknowledgement about the completion of the service. Synchronous messages instead, are those messages we have specialized in order to realize our mapping. This later kind of messages must be synchronous since the graphical editor needs to be informed about the success of the requested service before continuing operating.

9.3.2 Messages Definition

The Communication Protocol implementation offered by the GENESIS environment is reusable among arbitrary tools. This is why we have been able to use this system also for the implementation of our integration mechanism. Therefore, since a general message is defined in order to have the message types parametrized, the definition of our messages is a specialization of the predefined message. All the integration mechanism is implemented using the C^{++} language and the messages are defined as subclasses of the class defining the general message.

Mapping Implementation

By means of the message mechanism we have implemented the real mapping from OMT to Beta. For each kind of *Message* defined, representing a service request, we have also defined the corresponding *Service* which executes the request by invoking some methods on the textual editor. These methods handle the variations of the textual representation on the basis of the received request. Obviously, even if the real mapping is realized by the *Services*, this starts in the graphical editor, which sends the appropriate messages and prepares their proper parameters.

Message Structure

Each *Message* class contains the declaration of its attributes (representing the parameters of the correspondent message) and offers the methods to access them. One of the attributes is always the name of the textual Document corresponding to an Object Model drawn in the graphical editor. This allows to contemporaneously build two different Object Models and to have at the same time the respective textual representations as well as the contents of different Beta fragments. The other attributes instead are dependent on the the particular kind of service request and they are used to implement one of the graphical concepts. The implemented *Service* then contains only the declaration (and the implementation) of one method, *Execute*, which takes as parameters the corresponding message and performs the actions needed to implement the request.

Defined the Architecture of Messages

In order to implement our mapping, we have defined the messages dividing them on the basis of the OMT concept such as class, generalization, association and aggregation. Following we explain the service requests these messages represent in correspondence of each OMT concept.

- *Classes.* We have defined the messages to create and delete a class, to add attributes and operations and to delete both;
- *Generalization.* We have defined messages to set the name of the superpattern and to transform an operation into a virtual function or procedure;
- *Associations.* We have defined all the messages to create all kinds of associations. To handle the link attributes or to delete the associations the graphical editor may use the same messages already defined on classes;
- *Aggregation.* We have defined the messages to propagate an operation, while to

create the instances of the component parts we have used the messages defined for the classes.

9.3.3 Examples of Message Use

In this section we propose two examples to show how we have implemented two of the OMT concepts which were not directly mapped into Beta. We show how we have translated the specialization and the propagation of operations.

Specialization of Operations

When a user adds an operation in a subclass of a generalization, the tool has to understand if this operation is a specialization of another operation. So it checks, by means of the graphical editor, if the inserted operation has already been declared in one of the superclasses. In this case, the graphical editor performs two different actions: first it inserts the operation as a binding function or procedure; afterwards it transforms the namesake operation in the superclass in a virtual function or procedure. Both the actions are realized by means of two messages. The first action is performed by sending one of the following messages:

AddBindingFunctionMessage(DocName, ClassName, FunctionName)

or

AddBindingProcedureMessage(DocName, ClassName, ProcedureName)

where *DocName* is the name of the document where the textual editor has to realize the changes; *ClassName* is the name of the class in which this function or procedure has to be added (the subclass) and *FunctionName* and *ProcedureName* are the names of the function or procedure that the user is going to specialize.
This message is received by the corresponding Service: in this case *AddBindingFunctionService* or *AddBindingProcedureService*. These Services in turn invoke some methods defined in the textual editor to add the binding function or procedure and to set its name.

When the textual editor has finished to add the new operation in the subclass, the graphical editor sends another message to transform the superclass operation in a virtual operation. The invoked message is:

MakeVirtualMessage(DocName, ClassName, OperationName)

Also in this case this message is executed by the correspondent Service, *MakeVirtualService*, which again realizes the transformation invoking the correspondent method in the textual editor.

Propagation of Operations

Another interesting situation our tool has to manage is when a user wants to propagate an operation from an aggregate class to one of its component classes. As we have shown in the first paragraph, in this case the user selects one of the aggregation arcs entering the component class and invokes the corresponding method. Then the graphical editor asks the user to insert the name of the propagated operation. Therefore the textual editor checks that the operation has been defined on the aggregate classes and then sends two messages to the textual editor to realize the propagation.
The first message is:

AddFunctionMessage (DocName, ClassName, OperationName, Parameters, EnterPart)

or

AddProcedureMessage (DocName, ClassName, OperationName,Parameters, EnterPart)

depending from the kind of pattern it has to add in the textual representation (see chapter 5). In these messages, *ClassName* is the name of the component class, *OperationName* is the name of the propagate operation, *Parameters* and *EnterPart* are two parameters which contain other informations needed to add the operation. This message is received from the Service, *AddOperationService*, which calls the corresponding methods to add the operation in the textual version. The second message instead informs the aggregate to propagate that operation:

PropagateOperationMessage (DocName, ClassName,OperationName, Imperative)

where Imperative is an identifier which represents the imperative calling the propagated operation on the component class. This message is received by *PropagateOperationService* which inserts the imperative in the textual representation of the operation in the aggregate class (see also section 5.1).

9.3.4 Summary

This chapter has shown the last phase of our tool development process - the implementation. We have seen how to integrate the graphical and the textual editor and how to realize the mapping described in chapter 5.

The main important decision was to realize the integration by means of a Communication Protocol Subsystem written in C^{++} instead of operating system primitives for inter process communications, such as sockets or pipes. We have made this choice because in this way we were able to realize the integration at a much higher level of abstraction and the whole communication mechanism is hidden from the rest of the tool architecture.

Moreover, this kind of integration is particularly enabled since there is a homomorphism between the GTSL specification of our tool and its interface to the external components, such as the Communication Protocol Subsystem. This morphism has been defined in the GENESIS environment, and it has been used by the methods implementation capabilities of the GP_Language. In this way we think we have used the means we had at our disposal as good as possible.

Chapter 10

Conclusions and Further Work

10.1 Conclusions

This thesis offers an example of integration between the analysis, design and implementation phases of the software development process. We have seen how OMT is the most suitable OOAD methodology for constructing a formal description of complex systems that has to be refined into Beta executable programs. A proposal of a mapping from the most fundamental OMT concepts into correspondent Beta concepts has been given. Moreover, the integration between these design and implementation languages has been concretized by means of a tool composed of a graphical and textual editor. With this tool the user is able to draw an OMT Object Model with a graphical editor, and to complete with the textual editor an automatically generated Beta skeleton that complies the mapping. Both the graphical and the textual editor are syntax driven and in addition the textual editor allows the user to program in a free textual input mode.

By giving the solution proposed in the mapping from OMT to Beta, this thesis gives an example of how the semantic gap between the design and the implementation phases can be closed. The realization of the mapping has been one of the most interesting phases of our work. It led us to individualize those characteristics that make Beta a multi-perspective language and to take these characteristics off from our mapping in order to allow only implementations in a pure Object Oriented programming style. For example, it led us to take off the action part from those patterns that implement classes leaving this part only in those patterns that implement operations. In this way, as it should be, it is not possible to execute classes and we allow the access to attributes only by means of operations defined on them.

In developing this mapping we have also found some problems due to the implementation of the Beta compiler. They disabled the mapping of some OMT concepts of which multiple inheritance is the most important. For this features we gave a simulation of all the aspects that make it useful. However, we have not been able to give a unique translation covering all of them. Moreover, the consistency of associations and link attributes in One_to_One associations is not ensured by the mapping. The first problem is solved by means of the tool. In fact, using the textual editor, we do not allow the deletion of the part of the code that is automatically generated from the design. This ensures at all times the consistency between the Object Model drawn with the graphical editor and its textual representation. Moreover, during the implementation of the data structures used to realize the mapping for Many_to_Many and Ternary associations, we have discovered that it is not possible to specialize recursive functional or procedure pattern. This has avoided the use of one of the characteristics of inheritance, and therefore a more efficient implementation of the data structures.

In order to obtain an integration between analysis, design and implementation that was as complete as possible some decisions have been made. We have chosen OMT as the OOAD methodology to realize the integration between analysis and design from a theoretical point of view, and GraphProject and GENESIS, as generators for our editors that were efficient and able to build a well integrated tool. Our experiences have proved that, despite some problems, these decisions have been appropriate for our goal. OMT has proved to be the most suitable methodology to be translated in Beta giving no problems during the mapping. This method is one of the most complete and unambiguous, and for this reason, OMT has been used also as methodology during the design phase of our tool. In this phase, however, we have ascertained the need of a graphical notation for subsystems that Rumbaugh proposes only. A notation for these subsystems and a way to design a refinement of them has been introduced.

Instead, the choice of GP as a generator for our graphical editor, has caused more problems. The biggest problems were caused by some limitations in its graphical layout. This has enforced, for instance, the introduction of a second level of abstraction to handle and visualize some OMT concepts such as attributes and operations on classes or multiplicity on some relationships. Moreover it has also enforced the implementation of some OMT relationships by means of a composition of nodes and arcs. This has resulted in a graphical editor slower to use and in an implementation difficult to realize. Other problems we have found during the implementation are in visualizing comments and handling OMT attributes and operations as graphic objects. In fact GP does not allow to visualize long strings and to store a complex structure as an attribute of a graphical object. We have replaced complex structures with strings that, however, have been difficult to treat. In addition, it was not possible to provide the graphical editor with icons that we required to increase the clearness of the interface of the tool.

Our experiences in using GENESIS have been positive. It provided us with powerful and expressive means to implement the designed textual editor and the integration mechanism. The only negative aspect is that the specification of the editor has been long and tedious. This is due to the fact that GENESIS has been built to be used by a group of several developers and this has not been our case. Moreover, we have found a bug in the implementation of the compiler which is not able to distinguish the declared identifiers from C^{++} keywords. This has caused a complete and tedious revision of our code.

Considering the complete tool, choosing GP and GENESIS has been however a good solution. Together they offer a good example of how it is possible to realize a good integration between a textual and a graphical editor. This is possible since the GENESIS environment provides an external interface by means of the Communication Protocol Subsystem with which GP is able to connect, by means of the methods implementation capabilities.

The goal of our thesis to close a semantic gap between the abstraction mechanisms used in design and implementation has been realized. Adding the implementation of the mapping to the choice of OMT as an OOAD methodology that covers also the analysis phase, we gave an example of a good integration between the three most important software development process phases. Moreover, by means of the integrated Software Engineering Environment developed, we were able to concretize and verify that the integration between OMT and Beta is not only a theoretical study.

10.2 Further Work

We now present some proposals to complete and improve our work.

During our work we have treated only one of the three OMT models - the Object Model, and actually our tool allows to translate in Beta only a design of this model. However, providing the graphical editor with a proper interface, we have predisposed our tool to be extended in the future to support also the development of the Functional and Dynamic models. Hence this extension is realizable by implementing the other two models in the graphical editor and adding messages in the integration mechanism. Then the whole tool might be reused and extended to cover the whole methodology.

Since the Beta language is still under development, the integration between OMT and Beta might be getting easier if some extensions in Beta will be made. In fact adding some libraries to support sets and repetitions of instances (and not only of references) it might be possible to implement variable aggregation and give a better solution for the implementation of fixed aggregation. Moreover, Beta does not support multiple inheritance at the moment. We think that to simulate this feature it should be useful to declare renomination of operations with specialization. However, we are not sure that, also in this way, it is possible to give a unique impementation able to simulate all different aspects of this mechanism.

The problems found, due to the poor graphical layout of GP, should be remedied with an extension of it. This could allow to faithfully represent the OMT notation and to make the graphical editor more easily to use.

One feature GP supports is the possibility to draw hypernodes. An hypernode allows the generalization of a graph hierarchy starting from a "dummy" hypernode, called root, and which extends graph by graph by means of different windows corresponding to hypernodes in the previous graph. Since we have not been able to faithfully represent the OMT notation, this feature suggested us to add a graphical notation for subsystems that are defined in OMT, but not graphically supported, and which were useful in the design of the tool. However, the introduction of subsystems has not been possible since GP does not support inheritance between nested hypernodes.

Moreover, actually our tool allows to draw different object models corresponding to differents Beta fragments which are seen as independent documents. From the graphical editor point of view, the various Object Models are all parts of the same graphical document and, hence, they may be handled only by a single user at time. To provide our tool with multiuser support the possibility to associate different graphical documents to different hypernodes should be added. Different users should be able to contemporaneusly develop, save and load, different graphs belonging to the same diagram. Only in this way the consistency of the diagram, and hence of the correspondent textual documents, can be ensured.

The integration mechanism between the two tools has been realized by means of a communication protocol provided by the textual editor. This protocol, however, only allows boolean values to go back from the textual editor to the graphical one. This feature makes reverse engineering impossible to be realized, since it does not allow to propagate the changes made by the textual editor to the graphical editor. To implement this kind of facility, the integration mechanism should be realized by means of a common repository. In this way, in fact, we were not forced to send values from one to the other editor. We did not follow this kind of solution because, however, GP is not able to dynamically change the structure of the drawn graphs but only allows changes made by user interaction. Therefore an expansion of GP would allow the realization of reverse engineering simply by reimplementing the integration mechanism in a common respository. If also GENESIS would be extended to be able to send back more significant messages, only a few changes should be necessary.

All these extensions to our work are possible since the whole tool is realized in an Object Oriented style, i.e. in a modulare way, and, hence, supporting reusability of all its code. The modularity could also allow another interesting and possible modification with the substitution of the graphical or the textual language. Changing Beta with another O.O. language would simply mean to modify the textual editor, while most of the tool might be reused. The substitution of the graphical language leads to an analogous solution.

Bibliography

[Bai89] S.C. Bailin. *An Object-Oriented Requirements Specification Method.* HP Laboratories Bristol, HPL-91-52, June 1991.

[BDK92] Francois Bancilhon, Claude Delobel, Paris Kanellakis. *Building an Object-Oriented Database System, the story of O_2.* Morgan Kaufmann Publishers,1992.

[Boo91] Grady Booch. *Object Oriented Design With Applications.* The Benjamin/Cummings Publishing Co., Inc., Redwood City, CA 94065, 1991.

[Boo] G. Booch. *Object-Oriented development.* IEEE Trans. on Software Eng., se-12(2), 211-21.

[Car91] Carrol, M. . *Using Multiple Inheritance to Implement Abstract Data Types.* The $C^{+}+$ report, 3(4), Apr. 1991.

[CF92] Dennis de Champeaux & Penelope Faure. *A comparative study of Object-Oriented analysis methods.* March/April 1992.

[CG90] Carre, B., Geib, J-M. *The Point Of View Notion For Multiple Inheritance.* In OOPSLA'90' Object-Oriented Programming Systems, Languages and Applications, Vol. 25, No. 10, Oct. 1990.

[CI94] P. Corte and A. Inferrera. *Graph Project, Preliminaries Notes.* Engineering - Ingegneria Informatica S.p.a. GP - Version 1.0 Feb. 1994.

[Coa91a] P.Coad. *OOA & OOD: Continuum of Representation.* Journal of Object-Oriented Programming, pp. 55. Feb. 1991.

[Coa91b] P.Coad. *OOA/OOD and OOP.* Journal of Object-Oriented Programming, pp. 74, Mar./Apr. 1991.

[Coa91c] P.Coad. *Adding to OOA Results.* Journal of Object-Oriented Programming, pp. 64, May. 1991.

[Coa91d] P.Coad. *OOD Criteria, Part I..* Journal of Object-Oriented Programming, pp. 67, Jun. 1991.

[Col84] Colter, M.A. *A Comparative Examination of System Analysis Techniques.* MIS Q. (Mar. 1984) 51-66.

[Coo88] S. Cook *Impressions of ECOOP'88.* Journal of Object Oriented Programming, 1(4), 1988.

[Cox86] Brand J. Cox. *Object-Oriented Programming.* Reading Mass. : Addison-Wesley, 1986.

[CY91] Coad, P., and Yourdon, E. *Object Oriented Design.* Prentice Hall, Englewood Cliffs, N.J., 1991.

[DGK+84] V.Donzeau-Gouge, G.Kahn, B.Lang, M.Melese. *Document structure and modularity in Mentor.* ACM SIGSOFT Software Engineering Notes, 1984.

[Edw89] Edwards, J. Basic Ptech skills, course notes, *Associative Cesign technology.* Westborough, MA, 1989.

[Emm95] W.Emmerich *Tool Construction for Process-Centered Software Development Environments based on Object Database Systems* University of Paderborn, Germany, 1995. Forthcoming.

[FLP] Robert B. France, Maria M. Larrondo-Petrie. *Notation for Software Design.*

[FW94] Michael Fróhlich, Mattias Werner. *daVinci V1.4.1 User Manual.* University of Bremen, Dec., 1994.

[Gib90] Gibson, E. *Objects-born abd bred.* Byte, October, 245-254, 1990.

[GOO94] GOODSTEP Team. *The GOODSTEP Project: General Object Oriented Database for Software Engineering Processes.* In K. Ohmaki, editor, *Proc. of the Pacific Software Engineering Conference, Tokyo, Japan, pages 410-420.* IEEE Computer Society 12 Press, 1994.

[GR83] Adele Goldberg, David Robson. *Smalltalk-80: The Language and its Implementation.* Reading, Mass. : Addison-Wesley, 1983.

[Gui91] Guimaraes, N. . *Building generic User Interface Tools: an Experience With Multiple Inheritance.* In OOPSLA'91' Object Oriented Programming Systems, Languages and Applications, Vol. 26, No. 11, Nov. 1991.

[Hsi92] Donovan Hsieh. *Survey of Object-Oriented Analysis/Design Methodologies and Future CASE Frameworks.* CSL Technical Report. Computer Science Laboratory SRI-CSL-92-04, March 1992.

[IDE93] STP/OMT. *Software through picture.* IDE company, (Interactive development Environment), Sep. 1993.

[IJK90] Tadao Ichikawa, Erland Jungert, Robert R. Korfhage. *Visual Languages and Application..* Plenum Press, NY., 10013, 1990

[Kas80] U.Kastens. *Ordered Attribute Grammars.* Acta Informatica,1980.

[Kee89] Sonya Keene. *Object Oriented Programming in Common Lisp: A Programmer's Guide to CLOS..* Reading, Mass. : Addison-Wesley, 1989.

[KL88] Won Kim, Frederick H. Lochovsky. *Object-Oriented Concepts, Databases, and Application.* New York: ACM Press, 1988

[KLM83] G.Kahn, B.Lang, M.Melese. *Metal: a Formalism to Specify Formalisms.* Science of Computer Programming, 1983.

[KLM+93] J. Lindskov Knudsen, M. Lofgren, O. Lehrmann Madsen, B. Magnusson. *Object-Oriented environments: The Mjolner Approach.* Prentice All Object-Oriented Series, 1993.

[Knu68] D.E.Knuth. *Semantics of Context-Free Languages.* Mathematical System Theory, 1968.

[KWE91] Kurtz, B., S.N. Woodfield, and D.V. Embley. *Object Oriented system Analysis and Specification.* Hewlet-Packard & CS Dept., Brigham Young University, 1991(?).

[Lew88] C. Lewerentz. *Extended Programming in the Large in a Software Development Environment.* ACM SIGSOFT *Software Engineering Notes*, 1988.

[LMN93] Ole Lehrmann Madsen, Birger Moller-Pedersen, Kristen Nygaard. *Object-Oriented Programming in Beta Programming Language,* 1993.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall 1988.

[MP92] David E. Monarchi and Gretchen I. Puhr. *A Research Typology for Object-Oriented Analysis and Design.* Communications of the ACM, Vol.35, No. 9, pp. 35-47, Sep. 1992.

[OM] Odell, J. and J. Martin. *Object Oriented Analysis and Design.* Prentice Hall Englewood Cliffs, NJ (forthcoming)

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley publishing company, 1994.

[OSF92] OSF/Motif Style Guide. Revision 1.2, 1992. (For OSF/Motif Release 1.2). Open Software Foundation 11 Cambridge Center Cambridge, MA 02142.

[PJC+90] Page Jones, M., Constantine, L.L. and Weiss, S. *Modeling the Object Oriented System: The Uniform Object Notation.* Comp. language, Oct. 1990, pp. 70-89.

[Plo81] G.Plotkin. *A structural approach to operational semantics.* Aarhus Report DAIMI, Aarhus University, Denmark, 1981.

[Pre87] Pressman, R.S. *Software Engineering: A Practitioner's Approach.* Second ed., McGraw-Hill, 1987.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. *Object-Oriented Modeling and Design,* 1991.

[RL89] R, K.Raj, H.M. Levy. *A Compositional Model For Software Reuse.* In Proceeding of the 1989 ECOOP'89', British Computer Society Workshop Serie Cambridge University Press, 1989.

[RS95] Alex Repenning, Tamara Sumner. *Agentsheets: A Medium for Creating Domain-Oriented Visual Languages.* PhD dissertation, Dept of Computer Science, 1995.

[Rum87] James E. Rumbaugh. *Relations as semantic constructs in an object-oriented language.* OOPSLA'87' as ACM SIGPLAN 22, 12 (Dec. 1987), 466-481.

[SM89] S. Shlaer and S.J. Mellor. *An Object-Oriented Approach to Domain Analysis.* ACM SIGSOFT Software Engineering Notes, Vol. 14 No. 5, pp. 66-77, July 1989.

[SR90] *AT&T C++ Language System: Selected Readings.* CenterLine Software, Inc. Cambridge, Massachusetts, 1990.

[Sto86] B. Stroustrup. *The C++ Proramming Language.* Addison-Wesley, 1986.

[Swi93] Robert Switzer *Eiffel: an Introduction.* Prectice Hall, Object Oriented Series, 1993

[WBW+90] Wirfs-Brock, R.J., Wilkerson, B. and Wiener, L. *Designing Object Oriented Software.* Prentice Hall, Englewood Cliffs, N.J., 1990

[Weg92] Wegner P. *Dimensions of Object Oriented Modeling.* Computer , Oct. 1992, pp. 12-20.

[WN95] Kim Walden & Jean-Marc Nerson. *Seamless Object-Oriented Software Architecture, Analysis and design of reliable system.* Prentice Hall , 1995.

[WPM90] A.I. Wasserman, P.A. Pircher, and R.J. Muller. *The Object-Oriented Structured Design for Software Design Representation.* IEEE Computer, pp. 50-62, Mar. 1990.

[ZSG79] Marvin V. Zalkowitz, Alan c. Shaw, John D. Gannon. *Principles of Software Engineering and Design.* Prentice Hall, Englewood Cliffs NJ, 1979.

Appendix A

Graphical Editor Specification

This appendix presents the specification of the graphical editor interface, written in the GP specification language - GP_Language. The specification describes each term composing the graphical language by means of class declarations. We give a short presentation of each of these classes and the associated code. The order in which we present all the classes is given by the GP compiler which only allows to specify before the arcs, then the nodes and, at the end, the hypernodes.

Each class contains the definition of the shape of the graphical terms as described in section 9.1. Moreover, in all the defined classes, some common attributes, such as `Document` and `DocVersion`, and methods, such as `HandleInsertion` and `HandleDeletion` are declared.

`Document`: is the name of the Object Model currently drawn. This attribute has been inserted in order to implement the messages the graphical editor sends to the textual editor. In each moment, `Document` corresponds to the name of the textual document;

`DocVersion`: is a string representing the version of the Document.

Since these attributes are always the same in all the classes, they will be omitted from the following presentation.

The methods `HandleInsertion` and `HandleDeletion` are launched when the graphical term correspondent to the class in which they are defined is drawn or deleted. They handle the deletion and the insertion of the terms on the basis of their semantics. For this reason, they have different implementation in different classes and they will be explained in detail in each class.

A.1 Defined Arcs

The defined arcs are those we have already presented in section 9.1.

A.1.1 One_to_One Association

This class describes the namesake OMT relationship. On One_to_One associations the following attributes are defined:

`Origin` and `Destination`: names of the classes involved in the association.

`Bidirectional`: boolean value which indicates if the association is one way or two ways. It is used when the graphical editor sends the messages. By means of this attribute the editor can decide if it must send a message only to one class or both the classes involved in the association.

`Role1` and `Role2`: these attributes store the role names of the classes involved in the association.

We have also defined the followings methods:

`HandleInsertion`: this method asks the user to set the name of the association and to specify if he wants the association bidirectional. If the user does not insert the name, the request is repeated for other six times and, if the name has not been inserted yet, then the association is not created. If the association is related with a node `LinkAttribute`, the method checks if the other arc of the association has been already inserted and, in this case, sets the same name. Also the name of the class containing the Link Attributes is properly set.

`ChangeName`: when the user selects this method a window appears on the screen, displaying the old association name, and allowing the insertion of the new name. If the association is related with a `LinkAttribute`, then the new name is also set on the other arc of the association. Also the name of the class containing the Link Attribute is properly updated.

`InsertRoles`: the user calls this method to set the role names. The method displays a window, in which the names of the involved classes are shown, and asks for the role names. Only when the user has inserted both the names, these names are assigned to the corresponding attributes. If the association is related with a `LinkAttribute`, the role names are assigned also to the attributes of the other arc.

`ShowRoles`: this method is invoked by the user to visualize the role names associated with each class involved in the association. It displays a window in which the role names and the associated classes are shown.

ChangeLabel: this method can be invoked by other classes but not by the user. It is used by the method `ChangeName` to change the name of the other component arc of the association in case of Link Attributes.

HandleDeletion: this method firstly asks the user to confirm the deletion. If the arc is not connected to a `LinkAttribute` it deletes the arc. Otherwise, it does not allow the deletion and informs the user that before he has to delete the node `LinkAttribute`.

This is the declaration of the class implementing this kind of association:

```
define Assoc_1_1 : ARC
 with_shape  line = MEDIUM arrow = SINGLE label = "Assoc_1_1"
 with_attributes public char* Document
                 public char* DocVersion
                 public char* Origin
                 public char* Destination
                 public boolean Bidirectional
                 public char* Role1
                 public char* Role2
 with_methods trig_on_insert boolean HandleInsertion()
              visible boolean ChangeName()
              visible boolean InsertRoles()
              visible boolean ShowRoles()
              public boolean ChangeLabel()
              trig_on_delete boolean HandleDeletion()
```

A.1.2 One_to_Many Association

On One_to_Many associations there are the following attributes:

Origin and **Destination**: correspond to the namesake attributes described for One_to_One association;

Fixed and **Ord**: are two strings used to store the multiplicity of the association and if the association is ordered or not.

Bidirectional, **Role1** and **Role2**: correspond to the namesake attributes described for One_to_One association;

The followings methods also defined:

HandleInsertion: this method asks the user to set the name of the association and to specify if he wants the association bidirectional. If the user does not insert the name, the request is repeated for other six times and, if the name has not been inserted yet, then the association is not created. Otherwise, if the user inserts the name, then the method asks if he wants to set the multiplicity (fixed) and if he wants the association ordered. In this case the correspondent attributes are set with the appropriate values. If the association is related with a node `LinkAttribute`, the method checks if the other arc of the association has been already inserted and, in this case, set the same name. Also the name of the class containing the Link Attributes is set.

`ChangeName`: when the user selects this method a window appears on the screen, displaying the old association name, and allowing the insertion of the new name. If the association is related with a node `LinkAttribute`, then the new name is also changed on the other arc of the association and on the class containing the Link Attributes.

`InsertRoles`: the user calls this method to set the role names. The method displays a window in which the names of the involved classes are shown and the insertion of the role names is required. Only when the user has inserted both the names, these names are assigned to the corresponding attributes. If the association is related with a `LinkAttribute`, the role names are assigned also to the attributes of the other arc.

`ShowRoles`: this methods is invoked by the user to visualize the role names associated with each class involved in the association.

`ShowInfo`: this method shows the informations about the multiplicity (if it is not fixed an empty string) and if the association is ordered (yes/no).

`ChangeLabel`: as for One_to_One association, this method can be invoked by other classes but not directly by the user. It is used by the method `ChangeName` to change the name of the other component arc of the association in case of Link Attributes.

`HandleDeletion`: this method firstly asks the user to confirm the deletion. If the arc is not connected to a node `LinkAttribute` it executes the deletion, otherwise, it does not allow the deletion and informs the user that before he has to delete the node `LinkAttribute`.

This is the code associated to the class describing One_to_Many associations.

```
define Assoc_1_many : ARC
 with_shape line = MEDIUM arrow = SINGLE symbol = BALL
 with_attributes public char* Document
                 public char* DocVersion
                 public char* Origin
                 public char* Destination
                 public char* Fixed
                 public char* Ord
                 public boolean Bidirectional
                 public char* Role1
                 public char* Role2
```

```
with_methods trig_on_insert boolean HandleInsertion()
                visible boolean ChangeName()
                visible boolean InsertRoles()
                visible boolean ShowRoles()
                visible boolean Show_Info()
                public boolean ChangeLabel()
                trig_on_delete boolean HandleDeletion()
```

A.1.3 Many_to_Many Association

The class implementing Many_to_Many associations contains the declaration of the following attributes:

`Origin` and `Destination`: correspond to the namesake attributes described for the previous associations.

`Role1` and `Role2`: which, as for the other associations, store the role names of the classes involved in the association.

We have also defined the followings methods:

`HandleInsertion`: this method asks the user to set the name of the association. If the user does not insert the name, the request is repeated for other six times and if the name has not been inserted yet, then the association is not created. If the association is related with a node `LinkAttribute`, the method checks if the other arc of the association has been already inserted and, in this case, sets the same name. Also the name of the class containing the Link Attributes is set.

`ChangeName`: when the user selects this method a window appears on the screen displaying the old association name and allowing the insertion of the new name. If the association is related with a node `LinkAttribute`, then the other arc of the association and the class containing the Link Attributes are also updated.

`InsertRoles`: the user calls this method to set the role names. The method displays a window in which the names of the involved classes are shown and asks for the role names. Only when the user has inserted both the names, these names are assigned to the corresponding attributes. If the association is related with a node `LinkAttribute`, the role names are assigned also to the attributes of the other arc.

`ShowRoles`: this method is invoked by the user to visualize the role names associated with each class involved in the association. It displays a window in which the role names are shown.

`ChangeLabel`: this method can be invoked by other classes but not by the user. It is used by the method `ChangeName` to change the name of the other component arc of the association in case of Link Attribute.

`HandleDeletion`: this method first asks the user to confirm the deletion. If the arc is not connected to a node `LinkAttribute`, it executes the deletion. Otherwise, it does not allow the deletion and informs the user that before he has to delete the node `LinkAttribute`.

```
define Assoc_many_many: ARC
 with_shape line = LARGE
 with_attributes public char* Document
                 public char* DocVersion
                 public char* Origin
                 public char* Destination
                 public char* Role1
                 public char* Role2
 with_methods trig_on_insert boolean HandleInsertion()
              visible boolean ChangeName()
              visible boolean InsertRoles()
              visible boolean ShowRoles()
              public boolean ChangeLabel()
              trig_on_delete boolean HandleDeletion()
```

A.1.4 Ternary Association

This class defines the arcs used to draw the ternary association relationships. On ternary association we have defined the attributes: `Role1`, `Role2` and `Role3` which store the role names of the classes involved in the association.

We have also defined the methods:

`HandleInsertion`: that allows the insertion of the association name. Since a ternary association is composed of one node and three arcs, this method first checks if the arc the user wants to insert is the first arc composing the association, and in this case asks the user to insert the name. As for the other associations, if after six times the name has not been inserted yet, then the arc is not generated. If the arc is not the first arc of the association, the method sets the name with the same name of the other arcs.

`ChangeName`: when the user selects this method a window appears on the screen to allow the insertion of the new name. Then this change is also propagated to the other two arcs involved in the association.

`InsertRoles`: the user calls this method to set the role names. The method displays a window in which the names of the involved classes are shown and asks for the insertion of the role names. Only when the user has inserted all the three names, these names are assigned to the corresponding attributes.

`ShowRoles`: this methods is invoked by the user to visualize the role names associated with each class involved in the association. It displays a window in which the role names are shown.

HandleDeletion: this method asks the user to confirm the deletion and, in case, deletes the node.

Following the associated code is shown.

```
define ternar_Assoc : ARC
 with_shape  line = MEDIUM arrow = SINGLE
 with_attributes public char* Document
                 public char* DocVersion
                 public char* Role1
                 public char* Role2
                 public char* Role3
 with_methods trig_on_insert boolean HandleInsertion()
                 visible boolean ChangeName()
                 visible boolean InsertRoles()
                 visible boolean ShowRoles()
                 trig_on_delete boolean HandleDeletion()
```

A.1.5 Aggregation

This class defines the arcs of type aggregation used to draw the aggregation relationship. On this class, of type arc, the following attributes are defined:

NrPropagated: this attribute stores the number of propagated operations on this aggregation. Since propagating an operation on an arc entering in a node `Aggregation` has not meaning, this attribute will be meaningful only on those arcs exiting that node.

PropagatedList: this attribute stores in only one string the names of all the operations propagated on this arc.

We have also defined the methods:

HandleInsertion: this method simply allows the insertion of the arc and sets the associated label at the empty string.

Propagation: this method allows the user to propagate an operation only on those arcs exiting the node `Aggregation`. When the user invokes this method a window appears on the screen to insert the name of the operation to propagate. Then the method checks if this operation has been declared in an aggregate class and in this case appends the name of the operation in the attribute `PropagatedList` and sends a message to the textual editor to textually realize the propagation. If the operation has been not declared in the aggregate, an error message is displayed on the screen to inform the user about it.

`ShowPropagatedOp`: this method firstly checks if there are propagated operation. If there are not, then a message window informs the user about it. Otherwise, a new window is displayed to show the names of all the propagated operations.

`HandleDeletion`: we have decided to allow the deletion only of those arcs exiting the node `Aggregation_node`. Therefore, this method first checks if it is possible to delete the arc and, only in this case, allows the deletion. If the arc enters the node `Aggregation_node` then a message window displays the user that he has to delete before the node.

This is the correspondent code.

```
define Aggregation : ARC
 with_shape  line = MEDIUM arrow = SINGLE
 with_attributes public int NrPropagated
                 public char* PropagatedList
 with_methods trig_on_insert boolean HandleInsertion()
              visible boolean Propagation()
              visible boolean ShowPropagatedOp()
              trig_on_delete boolean HandleDeletion()
```

A.1.6 Aggregation With Multiplicity

This kind of aggregation may only appear in an aggregation relationship as one of the arcs exiting the node `Aggregation`. The attributes defined on aggregation with multiplicity are:

`Fixed`: this attribute is a string used to store the multiplicity of the aggregation.

`NrPropagated`: this attribute stores the number of propagated operations on this aggregation. Since this arc is always exiting a node `Aggregation`, we can be sure that this attribute is always meaningful.

`PropagatedList`: this attribute stores in only one string the names of all the operations propagated on this arc.

The defined methods are:

`HandleInsertion`: this method only allows the insertion of the arc and sets the label associated with the arc to an empty string. Moreover, it asks the user to insert the multiplicity. It is not allowed to create this kind of aggregation without setting the multiplicity. Therefore, if after six times the user has not fixed the multiplicity, then the aggregation is not created.

`Propagation`: this method allows the user to propagate an operation only on those arcs exiting the node `Aggregation`. When the user invokes this method a window appears on the screen to insert the name of the operation to propagate. Then the method checks if this operation has been declared in an aggregate class and in this case appends the name of the operation in the attribute `PropagatedList` and sends a message to the textual editor to textually realize the propagation. If the operation has been not declared in the aggregate, an error message is displayed on the screen to inform the user about it.

`ShowMultiplicity`: this method displays the multiplicity the user has set by means of a window displayed on the screen.

`ShowPropagatedOp`: this method firstly checks if there are propagated operations. If there are not, then a message window informs the user about it. Otherwise, a window is displayed to show the names of all the propagated operations.

`HandleDeletion`: this arc can be always deleted, so this method simply allows the deletion of the arc.

```
define Aggr_mult: ARC
 with_shape  line = MEDIUM arrow = SINGLE symbol = BALL
 with_attributes public char* Fixed
                 public int NrPropagated
                 public char* PropagatedList
 with_methods trig_on_insert boolean HandleInsertion()
             visible boolean Propagation()
             visible boolean ShowMultiplicity()
             visible boolean ShowPropagatedOp()
             trig_on_delete boolean HandleDeletion()
```

A.1.7 Generalization

This kind of arc is used to implement the namesake OMT relationship. On this arc we have not define any attribute while we have defined the following methods:

`HanldeInsertion`: this method simply allows the insertion of the arc setting the name with the empty string.

`HandleDeletion`: similarly to the `Aggregation` arc, also for this kind of arcs we have decided to allow the deletion only of those arcs exiting the node `Generalization_node`. This method asks the user to confirm the deletion and checks if it is possible to delete the arc. In this case it allows the deletion, otherwise it displays an error message.

```
define Generalization : ARC
 with_shape  line = MEDIUM arrow = SINGLE
 with_methods trig_on_insert boolean HandleInsertion()
             trig_on_delete boolean HandleDeletion()
```

A.1.8 Instantiation

This arc represents the namesake OMT relationship. On this class we have only defined the following two methods:

`HandleInsertion`: this method sets the name of the arc with the empty string and the name of the Instance node with the name of the class it belongs to, between brackets.

`HandleDeletion`: this method asks the user to confirm the deletion and, in case, deletes the arc.

```
define Instantiation : ARC
 with_shape  line = SMALL arrow = SINGLE
 with_methods trig_on_insert boolean HandleInsertion()
              trig_on_delete boolean HandleDeletion()
```

A.2 Defined Nodes

The defined nodes are those we have presented in section 9.1. We describe before the nodes `Class`, `LinkAttribute` and `Instance` that correspond to nodes in the OMT Object Model. Then, we describe those nodes we have introduced only to realize the aggregation, generalization and ternary association relationships as composition of a node and arcs.

A.2.1 Class

In order to handle attributes and operations on drawn classes, we have defined the following attributes:

`AttributeNr` and `OperationNr`: are two integer attributes used to store the number of the inserted attributes and operations respectively.

`AttributeList` and `OperationList`: are two strings used to store the inserted attributes and operations.

On this class, we have also defined the followings methods:

`HandleInsertion`: this method allows the insertion of a class. It asks the user to insert the name of the class. If he does not insert the name, the class is not generated. Instead, if the user inserts the name, this method checks if there is already another class with the same name. In this case it does not allow the insertion and an error message is shown. Otherwise the class with the inserted name is generated.

`ChangeName`: this method displays a window with the old name of the class and asks the user to insert a new name.

`AddAttribute`: when the user launches this method a window appears on the screen to allow the insertion of an attribute. Since an attribute may be a basic attribute or a repetition, if the user wants to insert a repetition, he has to specify this kind of attribute and to insert the name, the type and the range. Otherwise only the name and the type. If the user does not insert the name or the type, the method repeats the request for six times and, at the end, does not allow the insertion. If the insertion is correct, the method checks to see if another attribute with the same name has been already defined. In this case, it does not insert the attribute. Otherwise it appends the attribute to the string `AttributeList` and increments the attribute `AttributeNr`. After the insertion of the attribute, the method also allows the user to insert an optional comment.

`AddOperation`: this method allows the insertion of a new operation. It asks the user to insert the name the result type and the number of parameters. Then if the class is a generalization of another class, the method checks if the same operation has been already defined in the superclass and launch the method `VirtualOperation`. If this operation is not a specialization of another operation then a new window to insert the parameters and a possible exit value is displayed. After editing the parameters the method asks the user to insert an optional comment. `OperationList` and `OperationNr` are updated.

`DeleteAttribute`: this method checks if there are inserted attributes. If there are not, then it displays a message to inform the user about it. Otherwise, it asks to insert the name of the attribute the user wants to delete. Then it checks if this attribute has been inserted and in this case realize the deletion erasing the attribute from the string `AttributeList` and decrementing the attribute `AttributeNr`. If the attribute is not inserted, it displays an error message.

`DeleteOperation`: this methods checks if there are inserted operations and, if there are not, displays a message to inform the user. Otherwise, it asks the user to insert the name of the operation and checks in the string `OperationList` to see if the operation has been inserted. In this case it deletes the operation from the string `OperationList` and decrements the attribute `OperationNr`. Otherwise it displays and error message.

`ShowAttributes`: this method displays the inserted attributes. If there are not, then a message window appears on the screen to inform the user. Otherwise, this method scans all the string `AttributeList` to select the inserted attributes and displays them in a window.

`ShowOperations`: this method works similarly to the method `ShowAttributes` but on the string `OperationList` instead of `AttributeList`.

`VirtualOperation`: this method is launched by the method `AddOperation` when a class is a generalization of another class. It checks if the operation, the user wants to add, has been declared in a superclass. In this case, it returns the boolean true. False otherwise.

`HandleDeletion`: this method asks the user to confirm the deletion and then allows the deletion of the class. As consequence of this deletion, all the deletable arcs exiting or entering the class are also deleted.

```
define Class : NODE
 with_shape bitmap = classP
 with_attributes public char* Document
                 public char* DocVersion
                 public int AttributeNr
                 public char* AttributeList
                 public int OperationNr
                 public char* OperationList
                 public int Deletable
 with_methods visible boolean ChangeName()
                trig_on_insert boolean HandleInsertion()
                visible boolean AddAttribute()
                visible boolean AddOperation()
                visible boolean DeleteAttribute()
                visible boolean DeleteOperation()
                visible boolean ShowAttributes()
                visible boolean ShowOperations()
                public boolean VirtualOperation()
                trig_on_delete boolean HandleDeletion()
```

A.2.2 LinkAttribute

It defines the class containing the Link Attributes of an association. On the class `LinkAttribute` the following attributes are declared:

`AttributeNr` and `AttributeList`: are analogous to the namesake attributes defined in the class `Class`.

`Deletable`: is a integer value used to mark the node when this is included in an association. It is used to handle the deletion of all the associations.

Moreover, there are the following methods:

`HandleInsertion`: this method allows the insertion of a `LinkAttribute` and momentarily sets the label associated to the class with an empty string.

`AddAttribute`, `DeleteAttribute` and `ShowAttributes`: are defined as the namesake methods in the class `Class`.

`HandleDeletion`: this method asks the user to confirm the deletion and then deletes the associated graphic term. As a consequence of this deletion, also the arcs composing the association this node is linked to are deleted.

```
define LinkAttribute :  NODE
 with_shape label = "Link_Attributes" bitmap = attribute
 with_attributes public int NumArc
                 public int AttributeNr
                 public int Deletable
 with_methods trig_on_insert boolean HandleInsertion()
             visible boolean AddAttribute()
             visible boolean DeleteAttribute()
             visible boolean ShowAttributes()
             trig_on_delete boolean HandleDeletion()
```

A.2.3 Instance

For the class representing OMT instances, we have defined two methods:

`SetName`: this method is launched when a new arc `Instantiation` is created and sets the name of the instance. The name is the same of the class the `Instatiation` arc exits from, between brackets.

`HandleDeletion`: this method asks the user to confirm the deletion and, in case, deletes the instance. Deleting this node, the correspondent arc `Instantiation` is also deleted.

```
define Instance : NODE
 with_shape bitmap = instP
 with_methods public boolean SetName()
             trig_on_delete boolean HandleDeletion()
```

A.2.4 Aggregation_node

On this class we have defined only one attribute - `Deletable`. This is an integer value used to mark the node when it is inserted in an aggregation relationship. It is used to handle the deletion of the arcs `Aggregation` and `Aggr_mult`.

We have also defined the method `HandleDeletion` which asks the user to confirm the deletion and, in case, deletes the node and all the arcs entering and exiting the node.

```
define Aggregation_node : NODE
 with_shape label = "Aggregation" bitmap = romboP
 with_attributes public int Deletable
 with_methods trig_on_delete boolean HandleDeletion()
```

A.2.5 Ternary_node

On the class `Ternary_node` three attributes and two methods are defined. The attributes `Class1`, `Class2` and `Class3` are three strings used to store the names of the classes involved in the ternary association.

Moreover, these are the defined methods:

`HandleInsertion`: this method is launched when the node is inserted and it simply sets the name of the node with an empty string.

`HandleDeletion`: this method asks the user to confirm the deletion and, in case, deletes the node and all the arcs entering the node.

```
define Ternary_node : NODE
 with_shape label = "Ternary" bitmap = terP
 with_attributes public char* Class1
                 public char* Class2
                 public char* Class3
 with_methods trig_on_insert boolean HandleInsertion()
              trig_on_delete boolean HandleDeletion()
```

A.2.6 Generalization_node

On `Generalization_node` we have defined only one attribute, `Deletable`. It is an integer value used to mark the node when it is inserted in a generalization relationship. It is used to handle the deletion of the arcs `Generalization`. We have also defined the method `HandleDeletion` which asks the user to confirm the deletion before deleting the node and all the arcs entering and exiting the node.

```
define Generalization_node : NODE
 with_shape label = "Generalization" bitmap = triangleP
 with_attributes public int Deletable
 with_methods trig_on_delete boolean HandleDeletion()
```

A.3 Defined Hypernodes

We describe now the defined hypernodes. We have declared four different hypernodes:

- `OMT_Graphical_Editor`: which represents the graphical document produced by the editor;
- `Object_Model`: that corresponds to an OMT Object Model and it is the only one we have implemented;

- `Functional_Model` and `Dynamic_Model`: corresponding to the Functional and Dynamic OMT models.

A.3.1 OMT_Graphical_Editor

This "dummy" hypernode has been introduced to contain the three hypernodes correspondent to the three OMT models. It contains the declarations of the three hypernodes and of the arc `None` that has been inserted only to make the declaration consistent, but it does not represent any significant arc. Following the associated code is shown.

```
define OMT_Graphical_Editor: HYPER
 with_nodes
     Object_Model Functional_Model Dynamic_Model
 with_arcs
     None
 with_shape
     title = "PROJECT"
```

A.3.2 Object_Model

`Object_Model` is an hypernode that represents the OMT Object Model the user can draw. It contains the declarations of all the nodes and arcs described before.

On this hypernode we have defined only the method `Connect`, which asks the user to insert a name and realizes the connection with the textual editor creating a new document with the same name of the hypernode. Moreover the hypernode also includes the specification of the composition rules of such a graph. These rules allow to specify the "domain" and "co-domain" of the arcs, the number of the arcs entering and exiting a node and also if a cycle can be drawn. Following, the code of the class `Object_Model` is shown.

```
define Object_Model : HYPER
 with_nodes
     Class Instance Aggregation_node Generalization_node Ternary_node
     LinkAttribute
 with_arcs
     Assoc_1_1 Assoc_1_many Assoc_many_many ternar_Assoc Aggregation
     Aggr_mult Generalization Instantiation
 with_methods trig_on_insert boolean Connect()
 with_shape
         bitmap = partenza
         title = "Architecture Editor"
         window = (500,700)
         paper = (3000,3000)
 with_rules
     BIND(Instantiation,Class,Instance)
```

```
    BIND(Assoc_1_1,Class,Class)
    BIND(Assoc_1_1,LinkAttribute,Class)
    BIND(Assoc_1_1,Class,LinkAttribute)
    BIND(Assoc_1_many,Class,Class)
    BIND(Assoc_1_many,Class,LinkAttribute)
    BIND(Assoc_1_many,LinkAttribute,Class)
    BIND(Assoc_many_many,Class,Class)
    BIND(Assoc_many_many,Class,LinkAttribute)
    BIND(Assoc_many_many,LinkAttribute,Class)
    BIND(ternar_Assoc,Class,Ternary_node)
    BIND(ternar_Assoc,Ternary_node,Class)
    BIND(Aggregation,Class,Aggregation_node)
    BIND(Aggregation,Aggregation_node,Class)
    BIND(Aggr_mult,Aggregation_node,Class)
    BIND(Generalization,Class,Generalization_node)
    BIND(Generalization,Generalization_node,Class)
    NUM_ARC_OUT(Instance,Instantiation,1,1)
    NUM_ARC_IN(Class,Generalization,0,1)
    NUM_ARC_IN(LinkAttribute,Assoc_1_1,1,1)
    NUM_ARC_OUT(LinkAttribute,Assoc_1_1,1,1)
    NUM_ARC_IN(LinkAttribute,Assoc_1_many,1,1)
    NUM_ARC_OUT(LinkAttribute,Assoc_1_many,1,1)
    NUM_ARC_IN(LinkAttribute,Assoc_many_many,1,1)
    NUM_ARC_OUT(LinkAttribute,Assoc_many_many,1,1)
    NUM_ARC_IN_OUT(Ternary_node,ternar_Assoc,3,3)
    NUM_ARC_IN(Aggregation_node,Aggregation,1,1)
    NUM_ARC_OUT(Aggregation_node,1,INFINITY)
    NUM_ARC_IN(Generalization_node,Generalization,1,1)
    NUM_ARC_OUT(Generalization_node,Generalization,1,INFINITY)
    NO_DCYCLE(Aggregation)
    NO_DCYCLE(Generalization)
```

A.3.3 Functional_Model and Dynamic_Model

These hypernodes only contain the declarations of the nodes `Process` and `State` and of the arcs `Dataflow` and `Transition`. These elements have been declared only to allow the definition of the hypernodes but they do not have any implementation. We have defined only the method `HandleInsertion` which displays a message to inform the user that this model has not been implemented and does not allow the generation of the hypernode. The code corresponding to the two hypernode is shown below.

```
define Functional_Model : HYPER
 with_nodes
    Process
 with_arcs
    DataFlow
 with_methods trig_on_insert boolean HandleInsertion()
```

```
define Dynamic_Model : HYPER
 with_nodes
     State
 with_arcs
     Transition
 with_methods trig_on_insert boolean HandleInsertion()
```

Following there is the code associated with nodes and arcs defined on the hypernodes. Since we have not implemented the two models, on them we have not defined any method or attribute.

```
define Process : NODE
 with_shape label = "process"

define DataFlow  : ARC
 with_shape label = "data_flow"

define None  : ARC
 with_shape label = "None"
 with_methods trig_on_insert boolean Insert()

define State : NODE
 with_shape label = "state"

define Transition  : ARC
 with_shape label = "transition"
```

Appendix B

Beta Grammar

In the following we present the new complete grammar resulting from our revisions of the Beta grammar given in [LMN93]. This grammar has been corrected (see section 9.2.1) and some productions have been changed or added in order to allow the generation only of those programs written in an Object Oriented programming style. Moreover some productions have been introduced only to realize the mapping from an OMT Object Model. The grammar has been written in terms of an extended, normalized BNF, whose notation will be explained at the end of this appendix.

Our grammar begins with the rules to generate the main program, which may be seen as constituted by a list of declarations and a list of imperatives. Each declaration may be:

- a `ClassPattern`: corresponding to an OMT class;
- a `PartObject`: corresponding to an instance of a declared class;
- an `InstanceTree`: corresponding to an instance of a Many_to_Many or ternary association.

```
BetaProgram         :  "origin" "'~beta/basiclib/v1.3/betaenv"'
                       "(#"
                              DeclList
                       "do"
                             Imperatives
                       "#)"
DeclList            : {Decl}.
Decl                : ClassPattern | PartObject | InstanceTree.
```

We start now in describing a `ClassPattern` that is used to represent OMT classes. Class pattern
Each class is uniquely identified by the name of the class that corresponds to the name of the class pattern. Moreover, since a class can be a subclass of another, an

optional SuperPattern may be declared. The body of the class is constituted by a list of attribute declarations. We have omitted the action-part from the class body, since, from an Object Oriented point of view, each class only defines the state (attributes) and the behaviour (operations) of all the objects belonging to it. In this way, we have walk around the ambiguity of Beta due to the use of patterns as single abstraction mechanism.

```
ClassPattern     : ClassName ":" SuperPattern
                          "(#" Attributes
                          "#);".
SuperPattern     : | ClassName.
Attributes       : | {AttributeDecl}
```

Attributes

Each attribute may be one of the followings:

- A part-object;
- The declaration of a repetition;
- A functional or procedure pattern;
- A virtual functional or procedure pattern;
- A binding of a virtual pattern;
- The declaration of an association.

Here the correspondent productions are shown.

```
AttributeDecl    : | PartObject
                   | Repetition
                   | Associations
                   | FunctionalPattern
                   | ProcedurePattern
                   | VirtualFunc
                   | VirtualProc
                   | BindingFunc
                   | BindingProc.
PartObject       : Identifier ":" ClassName ";".
Repetition       : Identifier ":[" Index "]" ClassName ";".
Associations     : One | One_to_Many | Ordered | Many_to_Many | Ternary .
```

Functional and Procedure pattern

Usually in Beta a Functional pattern is a pattern that computes a value on the basis of a set of input parameters and gives this value in output by means of the exit-part. Instead, a Procedure pattern is a pattern used to temporary state informations when an object belonging to it is generated. Since this logical distinction is not clear from the syntactical definitions of the two patterns, we have decided to distinguish them only on the basis of the exit-part. So we call Functional Pattern that pattern attribute provided of an exit-part and Procedure Pattern the other. This also comply the distinction we have made for OMT operations. We have translated operations with output parameters with Functional patterns, otherwise with Procedure patterns.

```
FunctionalPattern    : Identifier ":"
                           "(#" Identifier
                                Attributes
                                EnterPart
                                Do
                                ExitPart
                           "#);".
ProcedurePattern     : Identifier ":"
                           "(#" Identifier
                                Attributes
                                EnterPart
                                Do
                           "#);".
```

In Beta a virtual patterns is a particular kind of pattern that, in the mapping, has been used to map those OMT operations of a superclass that are specialized in a subclass. In Beta the operation declared in the subclass is in Beta declared as binding. Since we have seen operations only as Functional and Procedure Patterns, we have restricted the features of virtual and binding only to these patterns. Virtual pattern

```
VirtualFunc    : Identifier ":<"
                     "(#" Identifier
                          Attributes
                          EnterPart
                          Do
                          ExitPart
                     "#);".
VirtualProc    : Identifier ":<"
                     "(#" Identifier
                          Attributes
                          EnterPart
                          Do
                     "#);".
BindingFunc    : Identifier "::<"
                     "(#" Identifier
                          Attributes
                          EnterPart
                          Do
                          ExitPart
```

```
                  "#);".
BindingProc     : Identifier "::<"
                  "(#" Identifier
                       Attributes
                       EnterPart
                       Do
                  "#);".
```

Action-Part

In Beta, the action-part of a pattern is composed of an enter, a do and an exit part. The enter-part is a list of parameters which may be entered prior to execution of the object. The do-part is a list of imperatives that describes the actions to be performed when the object is executed, and the exit-part is a list of output parameters which may be produced as a result. For these statements we have changed the syntax of Beta in order to make easier the realization of our mapping. Therefore the `EnterPart` is seen as an `Identifier`, and in the `ExitPart` only a predefined variable `result` is returned. Moreover we have also distinguished between `DoPart` and `DoInner`. As we said in section 9.2.1 we have used the `DoInner` only in the body of a virtual functional or virtual procedure pattern.

```
EnterPart      : | "enter"  Identifier.
Do             : | DoPart | DoInner .
DoPart         : "do" Imperatives Imperatives.
DoInner        : "do" Imperatives "INNER" Imperatives .
ExitPart       : "exit result" .
```

Associations

For each association defined in OMT we have introduced the corresponding production in the Beta grammar. One_to_One associations are realized by means of references to the related class. One_to_Many associations are instead declared as a `Set` or `List` of pointers to the class of the many end of the relationship. More complicated are the productions to declare Many_to_Many and ternary associations, that have been implemented by means of two different kinds of trees (see section 5.1).

```
One              : Identifier ":^" ClassName ";".
One_to_Many      : Identifier  ":@Set"
                     "(# element:<" ClassName
                     "#);".
Ordered         : Identifier  ":@List"
                     "(# element:<" ClassName
                     "#);".
Many_to_Many     : Identifier ":^Tree;".
ternary          : Identifier ":^TernaryTree;".
```

When a Many_to_Many or ternary association is declared also an instance of the association needs to be generated. This instance is defined with the same name of the association. Then the particular kind of tree is specialized by means of the related class names. Also the productions to add Link Attributes on these kinds of associations are given.

```
InstanceTree     : Identifier ":@" Instance .
Instance         : BiTree | ThreeTree .
BiTree           : "Tree(# type1:<" ClassName ";"
                          "type2:<" ClassName ";"
                           Atr
                       "#);".
ThreeTree        : "TernaryTree(# type1:<" ClassName ";"
                                 "type2:<" ClassName ";"
                                 "type3:<" ClassName ";"
                               "#);".
Atr              : | Assoc_atr.
Assoc_atr        : "Attribute:<" ClassName.
```

Imperatives

We want now to give the productions to generate sequences of imperatives. As we have seen, the do-part of an object is a sequence of imperatives that describe actions to be executed. In Beta we have two kinds of imperatives: the first kind are *evaluation imperative* (which will be described later) and the second kind, the *control structure imperative*, are few imperatives for controlling the flow of execution. We start in describing this later kind of imperatives. We omit the description of these productions because sufficiently explained in section 9.2.1. We want only to note that these productions do not correspond to any OMT concept, while these are useful only when the user wants to textually complete the do-parts of procedure or functional patterns.

```
Imperatives                : {Imperative}.
Imperative                 : | LabelledImperative
                             | LabelledCompoundImperative
                             | ForImperative
                             | IfImperative
                             | InnerImperative
                             | Evaluation.

LabelledImperative         : Identifier ":" LabelImperatives.

LabelledCompoundImperative : "(" Identifier ":" LabelImperative ":" Identifier ")"

LabelImperatives           : {LabelImperative}.
LabelImperative            : | LabelledImperative
                             | LabelledCompoundImperative
                             | LabelledFor
                             | LabelledIf
                             | LeaveImperative
                             | RestartImperative.
```

```
ForImperative              : "(for" index "repeat" Imperatives "for)".
LabelledFor                : "(for" index "repeat" LabelImperatives "for).

IfImperative               : "(if" Evaluation  Alternatives
                                   ElsePart "if)"
LabelledIf                 : "(if"  Evaluation LabelAlternatives
                                    ElsePart "if)".

Alternatives               : {Alternative}.
LabelAlternatives          : {LabelAlternative}.
Alternative                : Selection "then" Imperative.
LabelAlternative           : Selection "then" LabelImperative.
ElsePart                   : | "else" Imperatives.
LeaveImperative            : "leave"  Identifier.
RestartImperative          : "restart" Identifier.
InnerImperative            : "inner" Identifier.
```

The basic mechanism for specifying sequences of object execution steps is called an evaluation. An evaluation is an imperative that may cause changes in state and/or produce a value when it is executed. An evaluation may be an expression or an assignment. Objects declared as basic pattern and the assignment of these objects behaves like ordinary variables and assignment in traditional procedural programming languages. In the same way it is also possible to define value assignment for patterns. Assignments are used to give parameters in input to patterns implementing operations.

```
Evaluations    : | {Evaluation}.
Evaluation     : | Expression
                 | Assignment.

Assignment     : Transaction "->" Transaction.
Transaction    : | ObjectEvaluation
                 | ComputedObjectEvaluation
                 | ObjectReference
                 | EvalList
                 | Evaluation
                 | StructureReference.

EvalList       : "(" Evaluations ")".
```

The following productions describe all possible ways to dynamically generate instances of patterns. These allow the dynamic creation of objects and the call of operations from an object to another.

```
ObjectDescriptor         : SuperPattern "(#" Attributes
```

```
                                                EnterPart
                                                DoPart
                                                ExitPart
                                           "#);".
ObjectEvaluation          : | ObjectDescriptor
                            | Reference.
Reference                 : | AttributeDenotation
                            | DynamicItemGeneration.
ObjectReference           : Reference "[]".
StructureReference        : AttributeDenotation "##".
DynamicItemGeneration     : "&" ObjectSpecification.

AttributeDenotation       : | Identifier
                            | Remote
                            | ComputedRemote
                            | Indexed
                            | ThisObject.

Remote                    : AttributeDenotation "."  Identifier.
ComputedRemote            : "(" Evaluations ")" "." Identifier.
Indexed                   : AttributeDenotation "[" Evaluation "]".
ThisObject                : "this"  "(" Identifier ")".
```

Basic Pattern and Their Operations

A number of predefined basic patterns for commonly used data types are available. For the integer and real are available some functional patterns corresponding to the usual arithmetic functions. For the boolean pattern, the functional patterns **and**, **or** and **not** are defined.

```
Expression  : | BoolExp
              | NumExp.

BoolExp     : | BoolConst | EqExp
              | LtExp     | LeExp
              | GtExp     | GeExp
              | NeExp     | notExp
              | andExp    | orExp
              | identifier.

LtExp       : NumExp "<" NumExp.
LeExp       : NumExp "<=" NumExp.
GtExp       : NumExp ">" NumExp.
GeExp       : NumExp ">=" NumExp.
EqExp       : Expression "=" Expression.
NeExp       : Expression "<>" Expression.
notExp      : "not" BoolExp.
andExp      : "and" BoolExp.
```

```
orExp        : "or"  BoolExpr.

NumExpr      : | identifier
               | IntegerConst
               | AddExp
               | MulExp
               | DivExp
               | MinusExp.

AddExP       : NumExp "+" NumExp.
MulExp       : NumExp "*" NumExp.
DivExp       : NumExp "div" NumExp.
MinusExp     : NumExp "-" NumExp.
```

Predefined Classes

The only predefined classes are `IntegerConst`, `BoolConst` and `Identifier`. Moreover we have also the class `ClassName` which has been introduced only to realize the checking of static semantics in the generated textual editor.

```
index           : | SimpleIndex
                  | NamedIndex.
NamedIndex      : NameDcl ":"  Evaluation.
ClassName       : '[A-Za-z][A-Za-z0-9]*'.
IntegerConst    : '([0-9][0-9]*[0x[0-9]+)'.
BoolConst       : 'true | false'.
Identifier      : '[A-Za-z][A-Za-z0-9]*'.
```

The used formalism

In this section the rules of the formalism used to generate our grammar are shown. We first give this formalism in a BNF notation.

```
<grammar>         ::= <production-list>
<production-list> ::= <production> | <production> <production-list>
<production>      ::= <symbol> ``:'' <expr> ``.''
<expr>            ::= <alternative> | <optional> | <structure> | <reg-exp>
<alternative>     ::= <symbol> ''|'' <alternative> | <symbol> ''|'' <symbol>
<optional>        ::= ``|'' <structure> | ``|'' <reg-exp>
<structure>       ::= <component-list>
<component-list>  ::= <component> | <component> <component-list>
<component>       ::= <keyword> | <symbol> | <list>
<list>            ::= ``{`` <symbol> ``}'' <opt-delimiter>
<opt-delimiter>   ::= ``(`` <keyword-list> ``)''
<keyword-list>    ::= <keyword> | <keyword> <keyword-list>
```

```
<keyword>         : [‘‘].*[‘‘]
<reg-exp>         : [’].*[’]
<symbol>          : [a-zA-Z_-][a-zA-Z_-0-9]*
```

As this formalism describes, each *nonterminal* must be defined by exactly one of the following rules:

1. An *alternation rule* has the following form:

   $$\mathrm{A}_0 \quad : \quad \mathrm{A}_1 \quad | \ \mathrm{A}_2 \ | \quad \ldots \quad | \ \mathrm{A}_N$$

   where A_0, A_1,...,A_N are nonterminal symbols.

2. A *constructor rule* has the following form:

   $$\mathrm{A}_0 \quad : \quad \mathrm{w}_0 \ \mathrm{A}_1 \ \mathrm{w}_1 \ldots \mathrm{A}_N \ \mathrm{w}_N$$

   where A_0, A_1, ..., A_N are nonterminal symbols and w_0, w_1, ..., w_N are possibly empty strings of terminal symbols.

3. A *list rule* has one of the following forms:

   $$\mathrm{A} \quad : \quad \{ \ \mathrm{B} \ \} \ (\ \mathrm{w} \ )$$

   where B is a nonterminal and w is a possibly empty string of terminal symbols. The nonterminal A generates a list of B separated by w.

4. An *optional rule* has the following form:

   $$\mathrm{A} \quad : \quad | \ \mathrm{B}$$

   where B is a nonterminal. The nonterminal A may generate the empty string or B.

Terminal symbols are defined by productions with regular expressions on the right hand side.

Appendix C

Textual Editor Specification

This appendix presents the most significant part of the specification of the textual editor interface. Only the specification of the components of the language that are automatically generated from the graphical editor is described, while we omit those components that can be only textually edited such as the imperatives.

The structure of the interface specification is given by declaring the interface of each *increment class* defining all the concepts of a syntactic component of the language. The classes have been derived from the grammar given in Appendix B and then refined during the development of the various steps of the editor specification - the derivation of the inheritance and entity/relationship view (see sections 9.2.2 and 9.2.3). For each class an explanation of the relevant features and the associated code is given.

C.1 BetaProgram

The class `BetaProgram` is our root increment class. This class represents the syntactic unit of each document that is created corresponding to a generic fragment of a Beta program. Like all root increments of documents it has a common superclass, `DocumentVersion`, which allows to equip all defined editors with all required commands for version management. In the import interface section of the class, all resources on which the definition of the interface relies are defined. In particular, it enumerates all classes which are used within the interface for type declaration purposes. In this case, since in the grammar specification we have defined a `BetaProgram` as constituted of `DeclList` and `Imperatives`, these two classes must appear in the import interface. As we can see from the code below, in the method section of the interface there are a number of methods which are declared as *implicit*. Their specification do not need to be defined further, but however, it must be included for type-checking reasons. Some specific methods are also added. In particular, we have defined the method `get_declList` which returns the list of declarations inserted in a given fragment. This method is used by the integration mechanism to select the declarations of a particular defined class.

Following, the code of the class `Beta Program` is shown.

```
NONTERMINAL INCREMENT INTERFACE BetaProgram;

  INHERIT DocumentVersion;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT DeclList;
    IMPORT Imperatives;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    ABSTRACT SYNTAX
      TheDeclList : DeclList;
      TheImperatives : Imperatives;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      "origin",
      "'~beta/basiclib/v1.3/betaenv'", (NL),
      "(#",WS,
      TheDeclList,
      "do",(NL),WS,
      TheImperatives,(NL),
      "#)"
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):Program;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
      METHOD get_DeclList():DeclList; // This method returns the list of
                                      // declarations inserted in a Program.
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE BetaProgram.
```

C.2 DeclList

In GTSL each increment class has a reference to its abstract syntax father. In this case the class `DeclList` inherits from the class `Increment`, which is the most general abstract increment class, which allows its children to inherit all the properties that every increment should have. For example, methods to expand placeholder, check the static semantics correctness, etc. In the abstract syntax section, each variable defined represents a child with respect to the abstract syntax. List type, or set type constructors may be used for defining the type of the child. In this case the class `DeclList` has been declared as a type representing a list of instances of the class `Decl` in which the declarations are defined. Some methods to allow the generation and the deletion of instances of the classes `ClassPattern`, `PartObject` and `InstanceDecl` have also been added.

```
NONTERMINAL INCREMENT INTERFACE DeclList;

  INHERIT Increment;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT ClassPattern;
    IMPORT InstanceDecl;
    IMPORT Identifier;
    IMPORT Decl;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    ABSTRACT SYNTAX
      TheDecls: LIST OF Decl;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      TheDecls
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):DeclList;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);

      // These methods create a new ClassPattern,
      // PartObject or InstanceDecl template.
      // All these templates are added at the end
      // of the list DeclList.
      METHOD expand_with_class();
      METHOD expand_with_partObject();
      METHOD expand_with_instance();
```

```
      // They add a new ClassPattern, PartObject
      // or InstanceDecl template after the
      // current increment.
      METHOD add_class(cursor:Decl);
      METHOD add_partObject(cursor:Decl);
      METHOD add_instance(cursor:Decl);

      // This method deletes the current increment.
      METHOD delete_decl(cursor:Decl;enforced_deletion:BOOLEAN):BOOLEAN;
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE DeclList.
```

C.3 Decl

Decl is defined as abstract increment class. Abstract increment classes are used to collect and encapsulate all common instance variables and methods of increments deriving from the grammar of the document language. The class Decl is used to group together all possible kinds of declarations in a Beta program. Classes, part objects and instances are therefore considered as being a specialization of the class Decl. Each of these classes is constituted by a name, and this name and the methods to handle it are declared as specific attributes and methods of the class Decl.

```
ABSTRACT INCREMENT INTERFACE Decl;

  INHERIT Increment;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Identifier;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    ABSTRACT SYNTAX
        name:Identifier;
    END ABSTRACT SYNTAX;

    METHODS
      DEFERRED METHOD isolate();
      DEFERRED METHOD collapse();
      DEFERRED METHOD parse(Str:STRING):Decl;
      DEFERRED METHOD check();
      DEFERRED METHOD unparse():STRING;

      // The followings are the methods to set,
      // change and return the name of a
```

```
      // declaration.
      METHOD expand_name(str:STRING):BOOLEAN;
      METHOD change_name(str:STRING):BOOLEAN;
      METHOD get_name():Identifier;
    END METHODS;

  END EXPORT INTERFACE;

END ABSTRACT INCREMENT INTERFACE Decl.
```

C.4 Class Pattern

This increment class represents the corresponding Beta class pattern. A class pattern is constituted of a name (inherited from the class `Decl`), an optional `SuperPattern` and a list of attributes declarations. On this increment class we have defined the methods to return the list of the declared `Attributes`, `get_Declarations`, and to set the name of the superpattern, `set_superPattern`. These methods are invoked by some of the messages defined in the integration mechanism. The first is used to select the declared attributes, while the other to realize the generalization relationship.

```
NONTERMINAL INCREMENT INTERFACE ClassPattern;

  INHERIT Decl;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT SuperPattern;
    IMPORT Attributes;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    ABSTRACT SYNTAX
      TheSuperPattern : SuperPattern;
      TheAttributes : Attributes;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      name,
      ":",
      TheSuperPattern,(NL),
      "(#",WS,
      TheAttributes,(NL),
      "#)",
      ";",(NL)
    END UNPARSING SCHEME;

    METHODS
```

```
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):ClassPattern;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);

      // It returns the list of attributes inserted
      // in the current ClassPattern.
      METHOD get_Declarations():Attributes;

      // This method sets the variable TheSuperPattern
      // with the string received as parameter;
      METHOD set_superPattern(str:STRING):BOOLEAN;
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE ClassPattern.
```

C.5 Attributes

The class `Attributes` represents the list of all possible declarations that a class `ClassPattern` may contain. Therefore, in its import section all these kinds of declarations have been inserted and we have also defined all the methods to handle them. In fact, we have provided this class with the methods to add these declarations at the end of the list or after the current increment, and also the method to delete them.

```
NONTERMINAL INCREMENT INTERFACE Attributes;

  INHERIT OptionalIncrement;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT PartObject;
    IMPORT Repetition;
    IMPORT Associations;
    IMPORT FunctionalPattern;
    IMPORT ProcedurePattern;
    IMPORT BindingFun;
    IMPORT BindingProc;
    IMPORT Identifier;
    IMPORT AttributeDecl;
    IMPORT One;
    IMPORT One_to_Many;
    IMPORT Ordered;
    IMPORT Many_to_Many;
```

```
    IMPORT Ternary;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    ABSTRACT SYNTAX
      TheAttributes : LIST OF AttributeDecl;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      TheAttributes
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):Attributes;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);

      // These methods create a new PartObject, Repetition,
      // FunctionalPattern,  ProcedurePattern, BindingFunc,
      // BindingProc, One,  One_to_Many,Ordered, Many_to_Many
      // or Ternary template and add these templates at the
      // end of the list Attributes.
      METHOD expand_with_part_object();
      METHOD expand_with_repetition();
      METHOD expand_with_functional();
      METHOD expand_with_procedure();
      METHOD expand_with_bindingFunc();
      METHOD expand_with_bindingProc();
      METHOD expand_with_one();
      METHOD expand_with_one_many();
      METHOD expand_with_ordered();
      METHOD expand_with_many();
      METHOD expand_with_ternary();

      // All these methods add one of templates above
      // after the current increment.
      METHOD add_part_object(cursor:AttributeDecl);
      METHOD add_repetition(cursor:AttributeDecl);
      METHOD add_functional(cursor:AttributeDecl);
      METHOD add_procedure(cursor:AttributeDecl);
      METHOD add_bindingFunc(cursor:AttributeDecl);
      METHOD add_bindingProc(cursor:AttributeDecl);
      METHOD add_one(cursor:AttributeDecl);
      METHOD add_one_many(cursor:AttributeDecl);
      METHOD add_ordered(cursor:AttributeDecl);
      METHOD add_many(cursor:AttributeDecl);
```

```
      METHOD add_ternary(cursor:AttributeDecl);

      // The following method deletes the current increment.
      METHOD delete_declaration(cursor:AttributeDecl;
                        enforced_deletion:BOOLEAN):BOOLEAN;
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE Attributes.
```

C.6 AttributeDecl

The class `AttributeDecl` is the father of all the following increment classes:

- `PartObject` corresponding to a Beta part object;
- `Repetition` corresponding to the declaration of a repetition in Beta;
- `Associations` to declare any kind of association;
- `FunctionalPattern` and `ProcedurePattern` for functional and procedure patterns;
- `VirtualFunc` and `VirtualProc` to realize the virtual of a functional or procedure patterns;
- `BindingFunc` and `BindingProc` to realize the binding of a virtual patterns.

Since each of these classes is constituted by a name, this `name` and the methods to handle it are then declared as specific attributes and methods of the class `AttributeDecl`. Moreover, we have also declared the attribute `imp`, which represents the imperatives in functional and procedure patterns, or virtual functional and procedure patterns, or binding functional or procedure patterns.

```
ABSTRACT INCREMENT INTERFACE AttributeDecl;

  INHERIT Increment;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Identifier;
    IMPORT Imperatives;
    IMPORT Imperative;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
```

```
    ABSTRACT SYNTAX
      name:Identifier;
      imp:Imperatives;
    END ABSTRACT SYNTAX;

    METHODS
      DEFERRED METHOD isolate();
      DEFERRED METHOD collapse();
      DEFERRED METHOD parse(Str:STRING):AttributeDecl;
      DEFERRED METHOD check();
      DEFERRED METHOD unparse():STRING;

      // The followings are the methods to set,
      // change and return the name of an
      // attribute.
      METHOD expand_name(str:STRING):BOOLEAN;
      METHOD change_name(str:STRING):BOOLEAN;
      METHOD getName():Identifier;

      // The following method transforms the string received
      // as parameter in an Imperative template and adds this
      // template at the end of the imperatives list.
      METHOD edit_imperative(str:STRING):Imperative;
    END METHODS;

  END EXPORT INTERFACE;

END ABSTRACT INCREMENT INTERFACE AttributeDecl.
```

C.7 PartObject

The increment class `PartObject` has been inserted to allow the declaration of any instance of a Beta class. Each instance is composed by a `name` (inherited from the increment class `AttributeDecl`) and by a `type`, representing the name of a declared class. On this class we have defined the method `set_type` to set the value of the attribute `type`.

```
NONTERMINAL INCREMENT INTERFACE PartObject;

  INHERIT AttributeDecl;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Identifier;
    IMPORT Repetition;
  END IMPORT INTERFACE;
```

```
  EXPORT INTERFACE
    ABSTRACT SYNTAX
      type:Identifier;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      name,
      ":@",
      type,
      ";"
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):PartObject;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);

      // This method sets the type of the PartObject
      // with the string received as parameter.
      METHOD set_type(str:STRING):BOOLEAN;
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE PartObject.
```

C.8 Repetition

The increment class `Repetition` allows the declaration of an instance of a Beta repetition. This instance is composed by a `name` (inherited from the increment class `AttributeDecl`), by a value `IntConst` and a `type`, representing the range and the type of the declared repetition. On this class we have defined the methods, `set_range` and `set_type`, to set respectively the values of the range and the attribute `type`.

```
NONTERMINAL INCREMENT INTERFACE Repetition;

  INHERIT AttributeDecl;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT IntConst;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
```

```
    ABSTRACT SYNTAX
      TheIntConst : IntConst;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      name,
      ":[",
      TheIntConst,"]@",
      type,
      ";"
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):Repetition;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);

      // These methods set the range and the type of
      // the Repetition with the value and the string
      // received as parameter.
      METHOD set_range(int: INTEGER):BOOLEAN;
      METHOD set_type(str:STRING):BOOLEAN;
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE Repetition.
```

C.9 FunctionalPattern and ProcedurePattern

The increment classes `FunctionalPattern` and `ProcedurePattern` are two classes representing almost the same language construct. The distinction has been introduced to distinguish those OMT operations that return values from those that do not. The fundamental distinction is therefore in the declaration of the import section and of the unparsing section, since in `FunctionalPattern` we have declared the `ExitPart` and in `ProcedurePattern` we have not. On both the increment classes we have defined the methods to set and return the input parameters. Moreover, we have defined the method `make_virtual` to transform these two classes into the corresponding virtual classes - `VirtualFun` and `VirtualProc`. This method is invoked by one of the messages implementing the integration mechanism when an operation is added in the graphical editor to specialize one that is already defined in a superclass. Following the code of the two classes is shown.

```
NONTERMINAL INCREMENT INTERFACE FunctionalPattern;

  INHERIT AttributeDecl;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT EnterPart;
    IMPORT DoPart;
    IMPORT DoInner;
    IMPORT ExitPart;
    IMPORT ProcedurePattern;
    IMPORT Identifier;
    IMPORT VirtualFunc;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    ABSTRACT SYNTAX
      TheParameters: Identifier;
      TheEnterPart : EnterPart;
      TheDoPart : DoPart;
      TheExitPart : ExitPart;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      name,
      ":",(NL),
      "(#",
      TheParameters,(NL),
      TheEnterPart,
      TheDoPart,
      TheExitPart,
      "#);",(NL)
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();

      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):FunctionalPattern;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);

      // This method sets the parameters of the
      // FunctionalPattern (represented as a string)
      // with the string received as parameter.
      METHOD expand_parameters(str:STRING):BOOLEAN;

      // This method transforms the current FunctionalPattern
```

```
      // into a VirtualFunc pattern.
      METHOD make_virtual():VirtualFunc;

      // The following method returns the list
      // of the enter parameters of the
      // FunctionalPattern (represented as a
      // string)
      METHOD get_enter():EnterPart;
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE FunctionalPattern.


NONTERMINAL INCREMENT INTERFACE ProcedurePattern;

  INHERIT AttributeDecl;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT EnterPart;
    IMPORT DoPart;
    IMPORT Identifier;
    IMPORT VirtualProc;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    ABSTRACT SYNTAX
      TheParameters: Identifier;
      TheEnterPart : EnterPart;
      TheDoPart : DoPart;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      name,
      ":",(NL),
      "(#",
      TheParameters,(NL),
      TheEnterPart,
      TheDoPart,
      "#);",(NL)
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):ProcedurePattern;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
```

```
      IMPLICIT METHOD unparse_to_file(filename:STRING);

      // This method sets the parameters of the
      // ProcedurePattern (represented as a string)
      // with the string received as parameter.
      METHOD expand_parameters(str:STRING):BOOLEAN;

      // The following method transforms the current
      // ProcedurePattern into a VirtualProc pattern.
      METHOD make_virtual():VirtualProc;

      // This method returns the list of enter
      // parameters of the ProcedurePattern
      // (represented as a string)
      METHOD get_enter():EnterPart;
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE ProcedurePattern.
```

C.10 EnterPart

The increment class `EnterPart` has been defined as a child of the class `OptionalIncrement`, which is declared as a superclass for all increment classes that trace back to optional productions of the grammar definition. Hence, it equips any subclass with a command to remove the increments. The class `EnterPart` realizes the enter-part in an action part of a pattern. Usually this construct is constituted by a list of parameters. We have instead decided to represent all the input parameters by means of a unique string which comes from the graphical editor. On this increment class we have defined the method to set this string with the input parameters. This method is usually invoked by one of the messages of the integration mechanism.

```
NONTERMINAL INCREMENT INTERFACE EnterPart;

  INHERIT OptionalIncrement;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Identifier;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    ABSTRACT SYNTAX
      param: Identifier;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
```

```
      "enter",
      "(",
      param,
      ")",(NL)
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):EnterPart;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);

      // This method sets the list of enter parameters
      // of a  pattern (represented as a string) with
      // the string received as parameter.
      METHOD set_param(str:STRING):BOOLEAN;
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE EnterPart.
```

C.11 DoPart

The class DoPart is a subclass of the abstract increment class Do which has been introduced to include both the class DoPart and DoInner. These two classes represent respectively the do-part of a functional (or procedure) pattern or of a virtual pattern. The do-part is a list of imperatives that describes the actions to be performed when the enclosing object is executed. On this class we have defined the method make_do_inner which transforms a DoPart template in a DoInner template. This method is invoked by the method make_virtual in the increment classes FunctionalPattern and ProcedurePattern in order to transform a functional or procedure pattern into a virtual pattern.

```
 NONTERMINAL INCREMENT INTERFACE DoPart;

  INHERIT Do;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Imperatives;
    IMPORT DoInner;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
```

```
    ABSTRACT SYNTAX
      TheImperatives : Imperatives;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      "do",(NL),
      TheImperatives,(NL),
      TheImperatives
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):DoPart;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);

      // This methods transforms a DoPart template in
      // a DoInner template. It is launched by the
      // method make_virtual in the classes
      // FunctionalPattern or ProcedurePattern.
      METHOD make_do_inner():DoInner;
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE DoPart.
```

C.12 ExitPart

The increment class `ExitPart` is declared as a nonterminal increment class. Usually a nonterminal class serves to create further terminal and nonterminal increment classes. In this case this class does not allow the generation of any other class since its goal is only to visualize its unparsing scheme. However, we have used this kind of increment to make possible the declaration of the unparsing scheme, instead of a simple regular expression to represent it. On this class we have not defined any method.

```
NONTERMINAL INCREMENT INTERFACE ExitPart;

  INHERIT OptionalIncrement;

  IMPORT INTERFACE
    IMPORT Increment;
  END IMPORT INTERFACE;
```

```
  EXPORT INTERFACE

    UNPARSING SCHEME
      "exit result;",(NL)
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):ExitPart;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE ExitPart.
```

C.13 Associations

This abstract increment class is the superclass of every kind of association. This represents the implementation of an OMT association in one of the patterns representing a class involved in the association. Since each association has a name, this attribute has been declared in the abstract syntax of this class. Moreover, in order to specialize the reference implementing the association with the name of the other class involved in the association (in case of One_to_One and One_to_Many associations) also the attribute `pattern` and the method `expand_pattern` have been declared. On this class we have also defined the methods `expand_name` e `change_name` to set and change the name of the association.

```
ABSTRACT INCREMENT INTERFACE Associations;

  INHERIT Declaration;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Identifier;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    ABSTRACT SYNTAX
      name : Identifier;
      pattern: Identifier;
    END ABSTRACT SYNTAX;
```

```
    METHODS
      DEFERRED METHOD isolate();
      DEFERRED METHOD collapse();
      DEFERRED METHOD parse(Str:STRING):Associations;
      DEFERRED METHOD check();
      DEFERRED METHOD unparse():STRING;

      // This method sets the name of the association
      // with the string received as parameter.
      METHOD expand_name(str:STRING):BOOLEAN;

      // The following method allows to change the
      // name of the association.
      METHOD change_name(str:STRING):BOOLEAN;

      // This method specializes the reference
      // implementing the association with
      // the name of the other class
      // involved in the association.
      METHOD expand_pattern(str:STRING):BOOLEAN;
    END METHODS;

  END EXPORT INTERFACE;

END ABSTRACT INCREMENT INTERFACE Associations.
```

C.14 One and One_to_Many

These classes represent the OMT concepts of One_to_One and One_to_Many associations. We have also declared the class `Ordered` to distinguish between ordered and not-ordered One_to_Many associations. All these classes are declared in a very similar way: they have the same import and export section. The only difference is in the declaration of the unparsing scheme. Since all the needed methods are inherited from the superclass `Associations`, on these classes we have not defined any other methods. Following the code for the three classes is shown.

```
NONTERMINAL INCREMENT INTERFACE One;

  INHERIT Associations;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Identifier;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    UNPARSING SCHEME
```

```
      name,
      ":^",
      pattern,
      ";",(NL)
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):One;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE One.


NONTERMINAL INCREMENT INTERFACE One_to_Many;

  INHERIT Associations;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Identifier;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    UNPARSING SCHEME
      name,
      ":@Set",(NL),
      "  (# element:<",pattern,
      " #);",(NL)
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):One_to_Many;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE One_to_Many.
```

```
NONTERMINAL INCREMENT INTERFACE Ordered;

  INHERIT Associations;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Identifier;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    UNPARSING SCHEME
      name,
      ":@List",(NL),
      "  (# element:<",pattern,
      " #);",(NL)
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):Ordered;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE Ordered.
```

C.15 Many_to_Many and Ternary

Both Many_to_Many and ternary associations have been implemented by means of two defined data structures representing two different binary trees as described in section 5.1. Therefore these two increment classes are only used to declare the association in the involved classes. This declaration consists only in setting the name, by means of the corresponding inherited method, and in declaring the association as a pointer to the proper data structure. Hence, the implementation of this declaration is mostly generated by the unparsing scheme section and no methods have been declared. The code correspondent to these classes is shown below.

```
NONTERMINAL INCREMENT INTERFACE Many_to_Many;

  INHERIT Associations;
```

```
  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Identifier;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    UNPARSING SCHEME
     name,
      ":^Tree;",(NL)
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):Many_to_Many;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE Many_to_Many.


NONTERMINAL INCREMENT INTERFACE Ternary;

  INHERIT Associations;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Identifier;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    UNPARSING SCHEME
      name,
      ":^",
      "TernaryTree",
      ";",(NL)
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):Ternary;
```

```
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE Ternary.
```

C.16 InstanceDecl

The increment class `InstanceDecl` is generated by the methods `expand_with_instance` or `add_instance` defined on the class `DeclList`. It is used to generate an instance of the two kinds of binary trees implementing Many_to_Many or ternary associations. This nonterminal class allows the derivation of both the instances. A declaration of an instance is constituted by a `name` (corresponding to the name of the association) and by an `Instance` class, that corresponds to the abstract increment class for the two tree instances. The methods to handle the name are inherited from the superclass `Decl` and it adds only the methods to expand the `Instance` class with the desired kind of instance.

```
NONTERMINAL INCREMENT INTERFACE InstanceDecl;

  INHERIT Decl;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Identifier;
    IMPORT Instance;
    IMPORT BiTree;
    IMPORT ThreeTree;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    ABSTRACT SYNTAX
      TheInstance : Instance;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      name,
      ":@",
      TheInstance,(NL)
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
```

```
    IMPLICIT METHOD collapse();
    IMPLICIT METHOD parse(Str:STRING):InstanceDecl;
    IMPLICIT METHOD check();
    IMPLICIT METHOD unparse():STRING;
    IMPLICIT METHOD unparse_to_file(filename:STRING);

    // this method expands the declaration
    // of the instance with a BiTree template.
    METHOD expand_bi();

    // this method expands the declaration of the
    // instance with a ThreeTree template.
    METHOD expand_three();
  END METHODS;
 END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE InstanceDecl.
```

C.17 Instance

The abstract increment class `Instance` is the father of the classes generating the instance of the two kinds of binary trees. This class contains the declarations of the attributes `type1` and `type2` that are defined in order to specialize the names of the classes involved in the associations. Moreover, also the methods to set these attributes are defined. The third type representing the third class involved in a ternary association is specialized in the class `ThreeTree`.

```
ABSTRACT INCREMENT INTERFACE Instance;

 INHERIT Increment;

 IMPORT INTERFACE
  IMPORT Increment;
  IMPORT BiTree;
  IMPORT ThreeTree;
  IMPORT Identifier;
 END IMPORT INTERFACE;

 EXPORT INTERFACE

  ABSTRACT SYNTAX
   type1 : Identifier;
   type2 : Identifier;
  END ABSTRACT SYNTAX;


  METHODS
   DEFERRED METHOD isolate();
```

```
      DEFERRED METHOD collapse();
      DEFERRED METHOD parse(Str:STRING):Instance;
      DEFERRED METHOD check();
      DEFERRED METHOD unparse():STRING;

      // This two methods specialize the types
      // "type1" and "type2" with the names
      // of the classes involved in a many_to_many
      // or ternary association.
      METHOD expand_type1(st:STRING):BOOLEAN;
      METHOD expand_type2(st:STRING):BOOLEAN;
    END METHODS;

  END EXPORT INTERFACE;

END ABSTRACT INCREMENT INTERFACE Instance.
```

C.18 BiTree and ThreeTree

These nonterminal classes allow the declaration of instances of the two kinds of binary trees. Both the classes contain the declaration of an attribute `Atr` that represents the class containing the Link Attributes associated with the correspondent association. The class `BiTree` does not contain any other declaration of attributes or methods since they are all inherited from the superclass `Instance`. Instead the class `ThreeTree` instead contains the declaration of the attribute `type3` representing the name of the third class involved in ternary associations and the methods to set this attribute. Following the code correspondent to the classes is shown.

```
NONTERMINAL INCREMENT INTERFACE BiTree;

  INHERIT Instance;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Atr;
  END IMPORT INTERFACE;

  EXPORT INTERFACE

    ABSTRACT SYNTAX
      TheAtr : Atr;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      "many",
      "(#",
      "type1:<",
      type1,
```

```
      ";",(NL),
      "type2:<",
      type2,
      ";",(NL),
      TheAtr,(NL),
      "#)",
      ";",(NL)
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):BiTree;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE BiTree.


NONTERMINAL INCREMENT INTERFACE ThreeTree;

  INHERIT Instance;

  IMPORT INTERFACE
    IMPORT Increment;
    IMPORT Identifier;
  END IMPORT INTERFACE;

  EXPORT INTERFACE
    ABSTRACT SYNTAX
       type3 : Identifier;
    END ABSTRACT SYNTAX;

    UNPARSING SCHEME
      "ternary",
      "(#",
      "type1:<",
      type1,
      ";",
      "type2:<",
      type2,
      ";",
      "type3:<",
      type3,
      ";",
      "#)",
      ";"
```

```
    END UNPARSING SCHEME;

    METHODS
      IMPLICIT METHOD init(f:Increment);
      IMPLICIT METHOD expand();
      IMPLICIT METHOD isolate();
      IMPLICIT METHOD collapse();
      IMPLICIT METHOD parse(Str:STRING):ThreeTree;
      IMPLICIT METHOD check();
      IMPLICIT METHOD unparse():STRING;
      IMPLICIT METHOD unparse_to_file(filename:STRING);

      // This method specializes type3 with the name
      // of the third class involved in the ternary
      // association.
      METHOD expand_type3(st:STRING):BOOLEAN;
    END METHODS;
  END EXPORT INTERFACE;

END NONTERMINAL INCREMENT INTERFACE ThreeTree.
```

Appendix D

Defined Messages

This Appendix gives a brief presentation of the messages we have defined in the specification of the communication protocol that implements our mapping from OMT to Beta. For each of these messages we have specified a *Message*, representing a service request, and a *Service*, representing the object that executes the request by interacting with the textual editor. Each Message has been declared as a specialization of the predefined class *GenesisMessage*, and it simply adds its own parameters and the methods to handle them. Instead, each Service is a specialization of the predefined class *GenesisService*. A Service declares the methods to create a Service object and specializes a virtual method - *Execute* - which physically implements the execution of the requests. In order to present these messages we have decided to divide them on the basis of the OMT concepts they implement. However, we will see that sometimes it is not possible to follow this division since some messages are used to implement more the one OMT concept.

D.1 Class

On classes we have defined the messages to:

- Create and delete a class;
- Add attributes and operations;
- Add virtual operations;
- Change the name of an attribute or operation;
- Delete a declaration;

When the user graphically adds a new class and inserts its name, the graphical editor sends the message `CreateClassMessage` to the textual editor in order to perform the same operation also in the textual representation. The message and its parameters are shown below:

```
CreateClassMessage(Document, ClassName)
```

where: `Document` is the name of the document (see section 9.3);
`ClassName` is the name of the class to insert.

The corresponding Service, `CreateClassService`, receives the message and executes the request. It uses the parameter `Document` in order to find the correct textual document where it has to add the new class. Then, it selects the list of all the declared classes and requests the textual editor to add a new one at the end of this list. Finally, it asks the textual editor also to set the name of this new class with the string `ClassName` received as parameter.

To change the name of a defined class the user selects the corresponding method in the graphical editor and inserts the new name. The same method in turn sends the message `ChangeDeclNameMessage` to the textual editor to update also the textual representation. The message is shown below.

```
ChangeDeclNameMessage(Document, OldName, NewName)
```

where: `Document` is the name of the document;
`OldName` is the old name of the class;
`NewName` is the new name of the class.

The class `ChangeDeclNameService` receives this message and executes the request. It selects the right document and the right declaration, respectively identified by the parameters `Document` and `OldName`. Obviously, in this case the selected declaration is the declaration of a class. Then it asks the textual editor to change the name of the declaration with the string `NewName`.

In order to allow the insertion of attributes and operations we have defined other four messages. When the user adds a new attribute it has to specify if he wants to insert a “simple” or a repetition attribute. In the first case the graphical editor sends the message `AddAttributeMessage`, otherwise the message `AddRepetitionMessage`.

```
AddAttributeMessage(Document, ClassName, AttributeName, AttributeType)
```

where: `Document` is the name of the document;
`ClassName` is the name of the class in which we want to insert the new attribute;
`AttributeName` is the name of the new attribute;
`AttributeType` is the type of the new attribute.

To execute this request the class `AddAttributeService` first selects the proper textual document and class, by means of the parameters `Document` and `ClassName`, and afterwards it asks the textual editor to create a new part object template and sets its name and type with the strings `AttributeName` and `AttributeType` received as parameters. Instead to add a repetition attribute we have defined the message:

```
AddRepetitionMessage(Document, ClassName, AttributeName, Range,attribute-
                    Type)
```

where: `Document` is the name of the document;
`ClassName` is the name of the class in which we want to insert the new attribute;
`AttributeName` is the name of the new attribute;
`Range` is the range of the repetition attribute;
`AttributeType` is the type of the repetition attribute.

In this case the Service `AddRepetitionService` performs almost the same actions of the `AddAttributeService` class but it also sets the range of the repetition.

When the user adds a new operation on a class, the graphical editor checks if the operation returns a value or not. In the first case it sends the message `AddFunctionMessage`, otherwise the message `AddProcedureMessage`.

```
AddFunctionMessage(Document, ClassName, FunctionName, Parameters, Enter)
```

where: `Document` is the name of the document;
`ClassName` is the name of the class in which we want to insert the new function;
`FunctionName` is the name of the new function;
`Parameters` represents the declaration of all the parameters in one string;
`Enter` contains the string to be inserted in the enter part of a functional pattern.

```
AddProcedureMessage(Document, ClassName, ProcedureName, Parameters,
                   Enter)
```

where: `Document` is the name of the document;
`ClassName` is the name of the class in which we want to insert the new procedure;
`ProcedureName` is the name of the new procedure;
`Parameters` represents the declaration of all the parameters in one string;
`Enter` contains the string to be inserted in the enter part of a procedure pattern.

The Messages and the corresponding Services have almost the same structure. The only difference is in the methods the two Services invoke on the textual editor. They first select the proper textual document and class, by means of the parameters `Document` and `ClassName`. Then they ask the textual editor to add a new functional or procedure pattern template and set the name, the declarations of the parameters and the enter part with the strings they received as parameters. Moreover, `AddFunctionMessage` has not to bother about the exit-part of the functional pattern, since the textual editor always produce a pattern which returns a fixed variable *result*. The declaration of this variable is then part of the `Parameters` argument.

If the operation the user wants to insert has been already defined in a superclass, then the graphical editor does not allow the insertion of the operation (with the two messages above) but decides to create a binding operation. Therefore it sends one of the messages `AddBindingFunctionMessage` or `AddBindingProcedureMessage` as shown below.

`AddBindingFunctionMessage(Document, ClassName, FunctionName)`

where: `Document` is the name of the document;
`ClassName` is the name of the class in which we want to insert the new function;
`FunctionName` is the name of the new binding function.

`AddBindingProcedureMessage(Document, ClassName, ProcedureName)`

where: `Document` is the name of the document;
`ClassName` is the name of the class in which we want to insert the new procedure;
`ProcedureName` is the name of the new binding procedure.

The two corresponding Services perform exactly the same actions of the Services `AddFunctionService` and `AddProcedureService` respectively, but they ask the textual editor to create a new binding functional or procedure pattern template.

To allow the deletion of attributes, operations and, as shown later, associations, we have defined only one message, `DeleteDeclMessage`.

`DeleteDeclMessage(Document, ClassName, DeclName)`

where: `Document` is the name of the document;
`ClassName` is the name of the class containing the declaration the user wants to delete;
`DeclName` is the name of the declaration to delete.

This message is received by the Service `DeleteDeclService` which executes the deletion. It first selects the textual document and the class corresponding to the parameters

`Document` and `ClassName` and, afterwards, interacts with the textual editor to select the declaration whose name is `DeclName` and deletes it.

The last message we have defined on classes is the one the textual editor sends when the user wants to delete a class. We have therefore defined the message `DeleteClassMessage` shown below.

```
DeleteClassMessage(Document, ClassName)
```

where: `Document` is the name of the document;
`ClassName` is the name of the class to delete.

This message is received by the Service `DeleteClassService` which selects the textual document `Document` and then looks for the class `ClassName`. When the textual editor returns the pointer to this class, the Service invokes the method `delete_decl` to delete the class.

D.2 Association

On associations we have defined the messages to:

- Create a One_to_One association;
- Create an ordered or not-ordered One_to_Many association;
- Create a Many_to_Many association;
- Create a ternary association;
- Change the name of an association.

To delete an association or to add and delete a Link Attribute we have used the same messages already seen for classes.

When the user graphically adds a new association, the graphical editor sends one of the following messages to allow the creation of the corresponding association.

```
CreateAss_1_1Message(Document, Class1_Name, AssName, Class2_Name)
```

where: `Document` is the name of the document;
`Class1_Name` is the name of the class in which the new association has to be inserted;
`AssName` is the name of the new One_to_One association;
`Class2_Name` is the name of the other class involved in the association.

This message is received by the Service `CreateAss_1_1Service` that implements the creation of the association. The Service selects the textual document correspondent to the parameter `Document`. Then it looks for the class `Class1_Name` in which it has to insert the declaration of the association and asks the textual editor to add a new `One_to_One` template at the end of the declarations of that class. Afterwards it sets the name of the association with the string `AssName` and the pointer of the association with the name of the other class involved in the association, that is the last parameter, `Class2_Name`.

`CreateAss_1_ManyMessage(Document, Class1_Name, AssName, Class2_Name)`

where: `Document` is the name of the document;
`Class1_Name` is the name of the class in which the new association has to be inserted;
`AssName` is the name of the new association;
`Class2_Name` is the name of the other class involved in the association.

`CreateOrderedAss_1_ManyMessage(Document, Class1_Name, AssName, Class2_Name)`

where: `Document` is the name of the document;
`Class1_Name` is the name of the class in which the new association has to be inserted;
`AssName` is the name of the new association;
`Class2_Name` is the name of the other class involved in the association.

These messages are received respectively by the Services `CreateAss_1_ManyService` and `CreateOrderedAss_1_ManyService` whose implementation is almost the same of the Service above. The only difference is that in this case a new `One_to_Many` or `Ordered` template is created.

When the user adds a new Many_to_Many or ternary association the graphical editor sends one of the messages `CreateAss_Many_ManyMessage` or `CreateTernaryMessage` in order to create the same association also in the textual document.

`CreateAss_Many_ManyMessage(Document, Class1_Name, AssName, Class2_Name)`

where: `Document` is the name of the document;
`Class1_Name` is the name of the first class involved in the association;
`AssName` is the name of the new association;
`Class2_Name` is the name of the other class involved in the association.

```
CreateTernaryMessage (Document,Class1_Name, AssName, Class2_Name,
                      Class3_Name)
```

where: `Document` is the name of the document;
`Class1_Name` is the name of the class in which the new association has to be inserted;
`AssName` is the name of the new association;
`Class2_Name` and `Class3_Name` are the names of the other classes involved in the association.

These messages are received by the Services `CreateAss_Many_ManyService` and `CreateTernaryService` whose implementation is completely different from those of the other messages used to create associations we have seen before. These Services select the textual document corresponding to the parameter `Document`. Then they look for the class `Class1_Name` in which they have to insert the declaration of the association. Therefore they ask the textual editor to add a new `Many_to_Many` or `Ternary` template at the end of the declarations of that class, and they set the name of the association with the parameter `AssName`. Then they perform the same actions but with the classes `Class2_Name` and/or `Class3_Name`. Finally, they have to add a new `BiTree` or `ThreeTree` template in order to instantiate the binary tree implementing the association. Therefore they ask the textual editor to add a new `BiTree` or `ThreeTree` template and set the types of the elements of this tree with the names of the involved classes.

To change the name of an association the user invokes the correspondent method in the graphical editor. This method in turn sends the message `ChangeAssNameMessage` to update the textual document. The message is shown below.

```
ChangeAssNameMessage(Document, ClassName, OldName, NewName)
```

where: `Document` is the name of the document;
`ClassName` is the name of the class in which the declaration has been defined;
`OldName` is the old name of the declaration;
`NewName` is the new name of the declaration.

The Service `ChangeAssNameService` receives this message and implements the requests. It selects the textual document correspondent to the parameter `Document` and the class in which the association has been inserted. Then it selects the declaration of the association corresponding to the name `OldName` and asks the textual editor to change this name with the string `NewName`.

D.3 Generalization

On Generalization we have defined the message to:

- Specialize a subpattern with the name of the superpattern;

- Transform an operation into a virtual operation.

When a new class is added in a generalization relationship, this class needs to be declared as a subclass of another class. Therefore the textual editor sends the message `SetSuperPatternMessage` to implement this service.

```
SetSuperPatternMessage(Document, ClassName, SuperPattern)
```

where: `Document` is the name of the document;
`ClassName` is the name of the subclass in which the name of the superpattern has to be inserted;
`SuperPattern` is the name of the superpattern.

The Service `SetSuperPatternService` selects the textual document `Document` and the class `ClassName`. Then it asks the textual editor to specialize the subpattern setting the superpattern with the parameter `SuperPattern`.

When a new operation is inserted in a subclass or when a new generalization is created, the graphical editor checks if the operations of the subclass have the same name of others operations in the superclass. In this case these last operations need to be transformed into virtual operations. Therefore the graphical editor sends the following message `MakeVirtualMessage` to perform this transformation.

```
MakeVirtualMessage(Document, ClassName, OperationName)
```

where: `Document` is the name of the document;
`ClassName` is the name of the superclass in which the operation has to be transformed in a virtual operation;
`OperationName` is the name of the operation to transform.

The Service `MekeVirtualService` executes the service request. It selects the textual document correspondent to the parameter `Document` and the class correspondent to the parameter `ClassName`. Then it looks for the operation `OperationName` and invokes on this operation the defined method `make_virtual`.

D.4 Aggregation

On Aggregation we have defined the message to allow the propagation of operations from an aggregate to its component classes. This message contains also as parameter a string, `Imperative`, representing the imperatives to be performed in the aggregate to propagate the operation to the component classes (see section 5.1).

```
PropagateOperationMessage(Document, ClassName, OperationName, Imperative)
```

where: `Document` is the name of the document;
`ClassName` is the name of the class in which the operation has to be propagated;
`OperationName` is the name of the operation to propagate;
`Imperative` is the list of the imperative that must be inserted in the do part of the class propagating the operation.

The corresponding Service, `PropagateOperationMessage`, receives the message and executes the requests. It uses the parameter `Document` in order to find the correct textual document where it has to propagate the operation. Then it selects the class and the operation correspondent to the other parameters, `ClassName` and `OperationName`, and inserts the imperatives, received in `Imperative`, in the textual representation of the operation.

UNIDO