Generic Mechanisms to Extend Object-Oriented Programming Languages
The Reverse Inheritance Class Relationship

PhD Thesis

Author: asist. univ. ing. Ciprian-Bogdan Chirilă

PhD Supervisors:
prof. dr. ing. Ioan Jurca
prof. Philippe Lahire (University of Nice, France)

Faculty of Automation and Computer Science

University Politehnica of Timişoara

February 5, 2010

Abstract

Reverse inheritance is a potential class reusing mechanism having capabilities of creating abstact supertypes, factoring features from classes, redefining features, adapting features, adding an abstraction layer in a class hierarchy. This class relationship was not defined fully in the literature nor implemented in a programming language. To show that reverse inheritance is a feasible class relationship that helps class reusability, we defined its semantics for Eiffel by informal rules and we built a proof of a concept prototype. The semantics of reverse inheritance deals with feature signature exheritance, type exheritance, assertion exheritance, implementation exheritance, feature adaptations, feature clauses, genericity. The extended Eiffel language is modeled using Prolog facts, factbases corresponding to object-oriented systems. The semantics of the reverse inheritance concept is expressed through model transformations applied to the factbase. The factbase model is transformed using conditional transformations, which can detect transformation dependencies or can find a possible order for the transformation execution. Finally, the resulted model facts are translated automatically into pure Eiffel compilable code, in order to build the executable object-oriented system.

Acknowledgements

The current thesis is part of the PhD programme developed in the context of the collaboration between University Politehnica of Timisoara, Romania and University of Nice from Sophia-Antipolis, France, where I had several internships: three months may, june, july in 2003 and 2005, two months july and september in 2007, one month in september 2008 and two weeks in april 2009.

I would like to thank professor Ioan Jurca for his great efforts in supervising my PhD research activity, sustaining my research projects and elaborating the reviews for the research reports and thesis.

I would like to thank professor Philippe Lahire for the financial and intellectual resources invested in the thesis research. Specially, I want to thank professor Michel Riveill for financing a working visit in Bonn 2008.

I would like also to thank professor Markku Sakkinen from University of Jyväskylä, Finland, for showing points of interest for the developed problematics during one of my PhD internships and for helping with the development of the semantics.

I would like to express my gratitude to dr. Günter Kniesel from the Informatics Institute of Bonn, Germany, for the effort invested in the prototype implementation and for organizing a working visit in june 2008 at University of Bonn, Germany.

I want to express my regards to Mathieu Acher and Jean Ledesma, former students at University of Nice, for releasing the first version of ETransformer prototype module.

I want to thank also to the head of our department professor Vladimir Creţu for the punctual advices and management information given during the research activity. I would like to thank also the dean of our faculty, professor Octavian Proştean, for granting the financial support in some of the research internships.

Contents

List of Algorithms

List of Figures

List of Tables

Part I

Problem Analysis

Chapter 1

Introduction

1.1 Motivation

One of the most important factors on which the software quality depends is reusability. The benefits of reusability are increased speed of executing projects, decreased maintenance effort, reliability [Mey97]. In object-oriented technology, one way to achieve reusability is by organizing the classes in hierarchies. Currently, class organization is done by inheritance, which is considered one of the basic concepts in the object-oriented paradigm. Inheritance is an incremental modification mechanism which allows the transformation of the ancestor class into a descendant class by augmentation [Fro02]. In practice it has several uses, it can be used for **subtyping** as well as for **subclassing**. From the modeling point of view inheritance can be used either for **classification** or for **implementation**. A very close concept to the concept of inheritance is the reverse relationship, namely **reverse inheritance**.

The idea of reverse inheritance seems to have appeared in the world of objectual database [SN88], where the main goal is object reuse. Then it was integrated in the context of object-oriented programming languages as **generalization** [Ped89], in order to reuse classes. After that, some ideas of integrating it in Eiffel language can be found in [LHQ94], which we admit to be the most advanced approach at the moment. Finally, the same concept is discussed in several aspects related to multiple programming languages in the work of [Sak02].

The works of [SN88, Ped89, LHQ94, Sak02] argue about the idea that the reverse inheritance concept favors software reusability in the case of object-oriented systems. The creation of a generalized class which plays the role of supertype and contains all the common features of subclasses is a way of achieving class homogeneity and a better reuse[SN88]. The interest for such a class relationship can grow when we are dealing with subclasses which belong to a library and have read-only source code or even worse, the source code is not available [Sak02]. The reason for which a library is read-only may vary: copyright reasons, maintenance reasons. We can mention also that this class relationship was neither completely developed in the literature, nor integrated in a programming language[Sak02]. Next, several ways in which reverse inheritance can be useful in class hierarchy reorganization are presented in more details [**C**CL05a, **C**CL07a, **C**CL07b].

1.1.1 Designing in a More Natural Way

In [Ped89] it is stated that reverse inheritance is a more natural way for designing class hierarchies. When modeling classes, it is considered that it is more natural to design each class with its own features and only then to notice commonalities and factor them in a common superclass. This will lead to avoidance of data and code duplication, which in object-oriented philosophy is error prone.

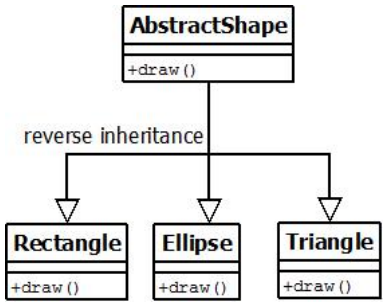


Figure 1.1: Capturing Common Functionalities

1.1.2 Capturing Common Functionalities

In some applications classes belonging to different contexts need to be used together. Sometimes they have even common functionalities which could be factored in one place to avoid duplication. There are several ways to achieve class adaptation and reuse. When the source code of classes is available and modifications are allowed, inheritance is the right choice[1]. An abstract superclass can be created by ordinary inheritance and all common code can be placed in the newly created superclass. One of the benefits of this solution is the type polymorphism and dynamic binding of common features. Any instance of the subclass can be referred using references of the superclass type. Common features can be called using superclass references and the code which will be executed is chosen at runtime.

We will address the situation of dealing with read-only code or precompiled class libraries where no modifications are possible. In this case reverse inheritance could be one solution for the unified management of the reused classes. In figure 1.1 we present the case of having three classes *Rectangle*, *Ellipse*, *Triangle* which were supposed to be developed in different contexts. A new abstract class *AbstractShape* was created which contains an abstract common feature *draw()*. The benefits discussed in the previous paragraph are still available in this solution, too. The programmer can manipulate instances of shapes through *AbstractShape* references. Of course, in practice, common features may exhibit different signatures, so they may need adaptations.

1.1.3 Inserting a Class Into an Existing Hierarchy

In this subsection is discussed the typical case of a class hierarchy which originally had two abstraction layers and later on was decided that a new middle abstraction layer is necessary. One choice is to affect the original classes and to make the modifications in order to reflect the new hierarchy. Of course, if other clients are already depending on the old class hierarchy, another solution must be considered. The use of reverse inheritance in such cases is recommended because it implies no modification of the original classes.

In the use case of figure 1.2 we present a class hierarchy which at design time had only two classes *Shape* and *Rectangle* in a subtype relationship. Later it was decided that a new class *Parallelogram* had to be added to the hierarchy. As it is known that any parallelogram is a shape and any rectangle is a parallelogram, so hierarchically class *Parallelogram* has to be between *Shape* and *Rectangle*. The solution proposed is to inherit the new class *Paralelogram* from *Shape* and to reverse inherit from *Rectangle*. This way the natural subtyping relations are preserved.

[1] Even if the reused classes have superclasses, in Eiffel multiple inheritance is allowed and should be used in this case. In programming languages like Java where no multiple inheritance between classes is allowed, the solution would be more complicated.

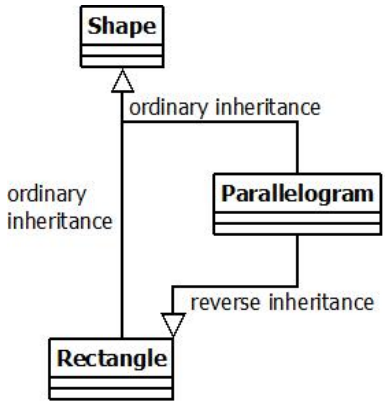


Figure 1.2: Inserting a Class Into an Existing Hierarchy

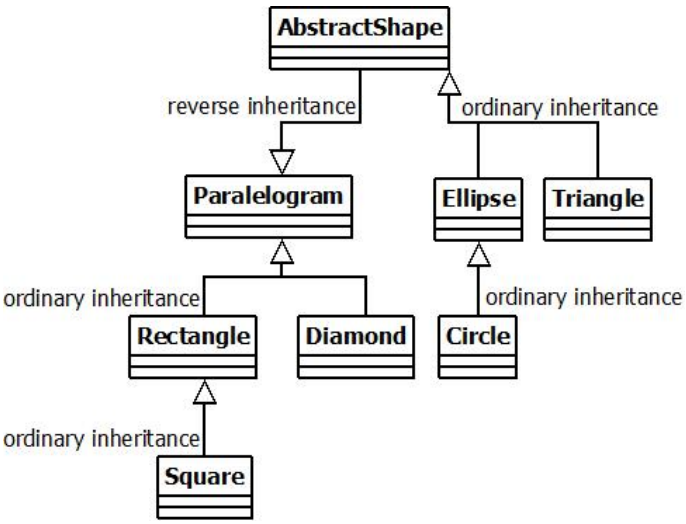


Figure 1.3: Extending a Class Hierarchy

1.1.4 Extending a Class Hierarchy

In some applications the integration of a class hierarchy into a more general one could be of real help. The idea of connecting two (or more) class hierarchies together under a common superclass without affecting any of existing classes is achievable by reverse inheritance. The part of the system which is newly developed can use ordinary inheritance but the link to the read-only hierarchy has to be made through reverse inheritance.

In the use case depicted in figure 1.3 we have a situation of class hierarchy modeling shapes. Initially only the hierarchy rooted by class *Parallelogram* existed and it could not be modified. As a first step of the redesign process, an abstract superclass named *AbstractShape* is created using reverse inheritance. Afterwards, the evolution of the hierarchy comes naturally using ordinary inheritance for classes like *Ellipse*, *Circle* and *Triangle*.

1.1.5 Reusing Partial Behavior of a Class

Some classes in object-oriented systems exhibit a great quantity of behavior. Maybe in some contexts only a subset of them needs to be reused. This could be useful in situations where binary

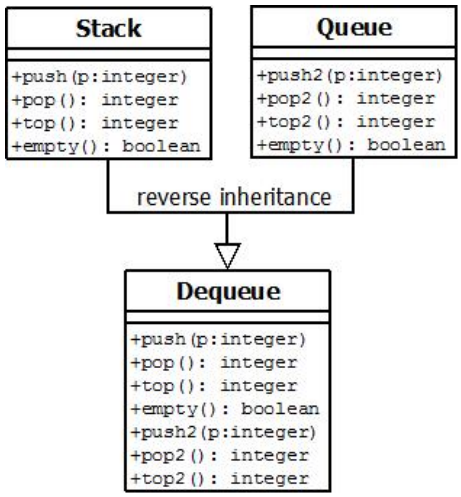


Figure 1.4: Reusing Partial Behavior of a Class

code size is critical or a supertype, containing a subset of features, is needed. On the other hand it could be good that clients are restricted to use only a part of the interface of an object and not all the features from it.

In the example located in figure 1.4, a *Dequeue* class is analyzed. Originally it was designed as a double ended queue, having operations for each end: *push*, *pop*, *top* (for one end) and *push2*, *pop2*, *top2* (for the other end). A new class *Stack* is created which is interested only in the operations related to one end of the *Dequeue* class. A new class *Queue* is then created to get the operations related to queue abstract data type. In conclusion, the programmer has the choice of reusing several parts of the code written in a class.

1.1.6 Creating a New Type

Another facility offered to the programmer by the use of reverse inheritance and like-type class relationship (which will be presented in section 2.2) is the creation of a new type starting from existing classes. Using reverse inheritance we can create a common superclass for the existing classes, like it was presented in subsection 1.1.2. Ordinary inheritance allows only direct inheritance of all features from the superclass while a like-type class relationship allows importing features selectively from other classes. In figure 1.5 starting from two terminal classes *Terminal1* and *Terminal2*, it was built a *TerminalANSI* class which gathers all common behavior and data. Later on, a new type is created, named *Terminal3*. This new type is created by ordinary inheritance from class *TerminalANSI*. It can be noticed that class *Terminal3* may import directly some features from *Terminal1* and *Terminal2* through the like-type class relation.

1.1.7 Decomposing and Recomposing Classes

Sometimes, in object-oriented systems a part of a class could be used to create a new class. This idea was presented also in subsection 1.1.5 where the reuse of the partial behavior of a class was discussed. In this use case it is proposed to facilitate better class design by decomposing classes and creating new ones by recomposing with the decomposed parts. In figure 1.6 it is presented such a situation where class *CalculatorWatch* was decomposed into two abstract classes *Calculator*, which contains the mathematical functions and *Watch*, which includes the list of clock functionalities. It was decided to exherit just the feature signatures into the abstract classes but not the implementation because in the two abstract classes there cannot be added new functionalities. It is more natural to extract the behavior using the like-type class relationship into classes *Calcu-*

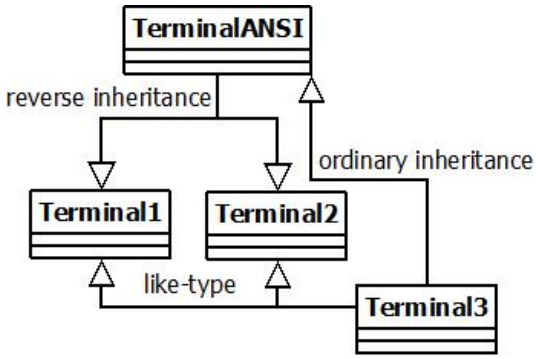


Figure 1.5: Creating a New Type

latorImplementation and *WatchImplementation*. Each implementation class is a subclass of the corresponding abstract exherited class: *CalculatorImplementation* is the subclass of *Calculator* and *WatchImplementation* is the subclass of *Watch*. Next, class *Watch* is combined with class *Cronograph* using multiple inheritance. Thus we showed a way of decomposing a class and recomposing it back with another class. It can be noticed that any eventual new features required in classes *Calculator* or *Watch* can be added in *CalculatorImplementation* or *WatchImplementation*. Adding new functionalities directly in classes *Calculator* or *Watch* would be inherited in class *CalculatorWatch* affecting its original behavior.

1.2 Overview on the Inheritance Class Relationship

The origin of inheritance dates from 1960 and was introduced in the Simula language where it was known under the name of **concatenation** [BDMN79]. The inheritance concept cannot exist without the concept of class through which the objects are defined. The class is considered to be the building brick of every object-oriented system having the role of both type and module [Mey97]. Classes are organized in hierarchies representing the backbone of almost every object-oriented system. They contain the state and the behavior of objects. There is no final definition for inheritance and its implementing mechanisms. Next, several informal definitions of inheritance are provided from the literature.

Inheritance is a class relationship where one class shares the state and behavior defined in one or more classes, so classes can be defined in terms of other classes. A subclass redefines or restricts the existing structure and behavior of the superclass [Boo94]. Inheritance in [Mey97, Int06] is defined as module extension mechanism because it makes possible to define new classes from existing ones by adding or adapting features, and as a type refinement mechanism which allows the definition of new types as specializations of already existing ones. Inheritance is commonly regarded as the feature that distinguishes object-oriented programming from object based programming or other modern programming paradigms [Tai96]. It supports the construction of reusable and flexible software. In the sense of object-oriented programming, inheritance is an incremental modification mechanism that transforms an **ancestor** class into a **descendant** class by augmenting it in various ways [Fro02]. The ancestor class is known also as **base** class, **parent** class or **superclass** and the descendant class as **derived** class, **child** class, **heir** class or **subclass**. A class is **abstract** if it has a partial implementation and as a consequence it can have no instances. Such a class may contain abstract and concrete members. In Eiffel language [Mey02], an abstract class is known also as a **deferred class**, an abstract feature as a **deferred feature** and a concrete feature as an **effective feature**.

Inheritance brings several benefits like code and data reuse, class hierarchy conceptual organization, rapid prototyping. There are also some drawbacks of inheritance class relationship. Execution speed is affected because object-oriented programs must include code implied by sev-

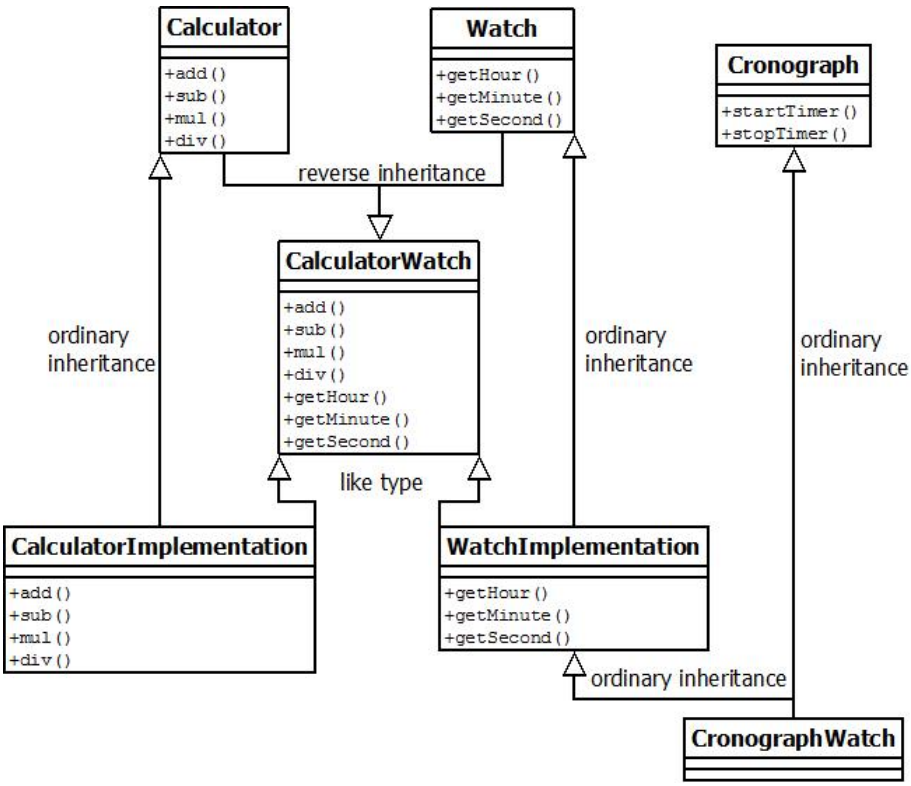


Figure 1.6: Decomposing and Recomposing Classes

eral supporting mechanisms like: constructors, method calling mechanism (polymorphism and parameter transmission), garbage collectors, run-time type checkers. Another consequence of the supporting mechanisms in object-oriented systems is program size. All of the mechanisms mentioned earlier imply routines which will be executed at runtime. So their code is added to the object-oriented system. The complexity of the object-oriented systems is higher compared to other systems designed using non object-oriented based paradigms. The complexity of the software systems is managed by the complexity of the object-oriented paradigm concepts.

In [Mey97] a taxonomy of inheritance uses is presented. Each possible purpose of inheritance is individually analysed.

Subtype inheritance is applied when: i) the heir classes represent sets of external objects; ii) heir classes correspond to subsets of ancestor class; iii) all heir classes must be mutually disjunctive. In this case the parent class has to be deferred. This kind of inheritance is very close to hierarchical taxonomies of botany, zoology and other natural sciences.

View inheritance is the type of inheritance that is used to manage the multiple criteria of classification between objects. The classes will represent non-disjoint partition sets. This kind of inheritance is based on multiple inheritance mechanism that enables an object having multiple views. All involved classes must be deferred.

Restriction inheritance where the heir class instances have additional constraints expressed through parts of the invariants. The ancestor and heir classes have to be both abstract or both concrete.

Extension inheritance involves adding new features to the superclass, thus creating a new enhanced subclass.

Variation inheritance (functional or type variation) involves either providing new implementations in the subclasses keeping the signatures intact or changing the signatures in a covariant way, but no other features in the subclasses should be added.

Uneffecting inheritance happens in the case when effective features from the superclass are redefined as deferred in the subclass.

Reification inheritance applies to cases in which the deferred superclass defines the specification of a data structure and the subclass implements it completely or partially. The superclass is deferred in this case and the subclass can be effective or deferred.

Structure inheritance applies between a deferred superclass defining a property and a subclass which models an object having that property. For instance a class COMPARABLE will be the superclass of all classes which support the comparing functionality.

Implementation inheritance facilitates the subclass to obtain a set of features (except constant attributes[2] and once functions[3]) from the superclass in order to implement the abstraction of the subclass.

Facility inheritance involves constant inheritance and machine inheritance. The purpose is to provide to the subclass a set of logically related features. Machine inheritance means that the set of features are routines viewed as operations on an abstract machine.

In [AAS01] it is considered that inheritance has mainly two different viewpoints: extension and specialization. The class relation is used by programmers in modeling, to express conceptual relations between classes and to share code between classes. It is presented a new abstraction mechanism named component, a solution that integrates both views of inheritance in an object-oriented language. The component is a non-instantiable collection of data and related operations. Classes can be composed of such components. Inheritance works at two levels: at component level for code-reuse and at class level for subtyping.

Inheritance represents an important reason for divergence in the community of researchers because of its different uses and implementations in the programming languages. There are a lot of works showing positive and negative examples of how inheritance must be used [LW94, Tai96].

[2]Constant attributes are those attributes that hold a read-only value.

[3]A once function differs from an ordinary function in the sense that it's body is executed only once on an instance, no matter how many times it is called.

1.3 Inheritance in Object-Oriented Programming Languages

In this section we will discuss about how the features of the inheritance mechanisms are implemented in several programming languages. In C++ [Str97, Sch98] there are mainly two kinds of inheritance: public and private (protected). The public inheritance is allowing the inheritance of superclass members with their default visibilities. Private inheritance hides the public and protected inherited members making them private in the subclass. Protected inheritance affects only the visibility of the public members, being protected in the subclass. On the other hand private inheritance involves that the subclass will be no longer a subtype of the superclass. That means that the type conformance relationship between the classes is disabled. In the case of multiple inheritance there is a special kind of inheritance called virtual inheritance. This is due to the virtual declarations of the base class which imply a special sharing behavior of the inherited features when multiple inheritance paths are available. These aspects will be detailed in a later section.

In Eiffel [Mey02, Int06] there are two kinds of inheritance: conforming and non-conforming. The feature inheritance mechanism is the same in both kinds of inheritance. The difference appears related to the subtyping relationship between the superclass and subclass. With conforming inheritance the subtyping relationship holds, while with non-conforming it doesn't. A special capability in the context of inheritance in Eiffel is the feature redeclaration. This may imply feature renaming, since it is considered that in the subclass, a new name may increase clarity, or redefinition, meaning change of signature or implementation. Of course, signatures may be changed in conformance with the rules of covariance. For this, a set of keywords are used: **rename**, **undefine**, **redefine**, **select**.

In Java [AG00] the inheritance mechanism always involves subtyping. The inheritance mechanism involves single subclassing and multiple subtyping. In other words, classes may have only one superclass while interfaces can have multiple superinterfaces. On the other hand, classes may implement multiple interfaces. By interface we refer to a special concept, which behaves like a pure abstract class and has only abstract methods. This approach avoids all problems encountered in some of the complex cases of multiple inheritance. In C# [FPB+02] we will find the same behavior as in Java, but some concepts may be named differently.

In the context of inheritance we can discuss also about inherited features or members. In an Eiffel class a feature has an unique name, which cannot be overloaded. In C++ and Java it is possible to define several methods with the same name but they are required to have different signatures. By signature it is meant member name, parameter number and types. Return types do not belong to the signature.

1.4 Thesis Objectives

The main goal of this thesis is class reuse. In order to achieve our goal we have to fulfill several objectives. The first objective is the design of a class reuse mechanism in order to facilitate reuse of already existing class libraries, by defining its semantics informally by principles, rules and examples. Afterwards we intend to implement it in an industrial strength language in order to experiment with it. The final objective is to show that the newly designed mechanism has the necessary features in order to help in the evolution and adaptation of class hierarchies.

1.5 Document Outline

In part I we propose to analyze the features of reverse inheritance class relationship from the conceptual point of view. Chapter 2 presents the most important reuse mechanisms of object-oriented technology. In section 3.1 we discuss generalities regarding reverse inheritance, like basic principles, notations in different approaches. Section 3.2 deals with exheritance at class interface level. We analyze which features from the class interface can be exherited and what major problems are encountered. In section 3.3 implementation exheritance issues are discussed. In section 3.4,

some interesting combinations of ordinary and reverse inheritance are studied. Section 3.5 points out the conclusions of our analysis.

In part II we define the semantics of reverse inheritance for Eiffel programming language. Chapter 4 presents the basic elements of our approach of reverse inheritance. Aspects like cardinality, feature factorization, type conformance are discussed in detail. In chapter 5 are presented main mechanisms through which feature adaptations can be performed. In chapter 6 are discussed dynamic binding aspects in the context of ordinary and reverse inheritance and constraints which must be imposed on foster classes. In section 6.5 the approach is reviewed.

In part III we present an implementation of the Eiffel reverse inheritance class relationship in Prolog. In chapter 7 we will present the Prolog reification of Eiffel and of the reverse inheritance extension and the architecture of the implementation prototype. Section 7.3 presents the model transformations in Prolog that expresses the semantics of reverse inheritance in a formal manner. Main aspects referring to feature exheritance, type exheritance, assertion composition, feature redefinition, feature adaptation, feature migration and exheritance facts removal are presented in the containing sections. Chapter 8 evaluates the approach by comparing it to other reuse mechanisms and describes an adaptation experiment on the Eiffel Kernel Library. In chapter 9 we draw the conclusions and we set the perspectives.

Chapter 2

Reuse Mechanisms in Object Technology

In this chapter we will tackle the principles behind the most significant reuse mechanism in object-oriented programming. The discussed aspects are mechanism design, encountered problems, possible compromising solutions and their supporting motivation. Implementation issues are also a goal for this section, as they could be reused or similar ideas could be developed starting from them. The proposed mechanism for analysis refer to central concepts of object-oriented technology like inheritance, mixins, traits, roles, separation of concerns with its object paradigms (aspect oriented programming, composition filters), classboxes and expanders. The main focus is set on several design criteria offered by these mechanisms:

i) to create new abstract supertypes;
ii) to factor features from classes;
iii) to combine the implementations of features;
iv) to redefine the implementation of features;
v) to adapt the implementation of features;
vi) to cancel the implementation of features;
vii) to add an abstraction layer into a class hierarchy.

2.1 Multiple Inheritance

In this subsection we will discuss several issues about multiple inheritance since it is a special form of inheritance, which is the base concept of the object-oriented paradigm. Multiple inheritance is one of the most powerful facilities offered for the software development allowing to combine several concepts in one abstraction. Disallowing inheritance to accept multiple parents would limit the potential of inheritance in general [Mey02]. Like single inheritance, multiple inheritance is used to extend the module (class) and to create a powerful type system in applications. Inheritance is **single** if the subclass has one parent or **multiple** if the subclass has multiple parents. Because a class can inherit parents in more than one way, the case of **repeated inheritance** occurs. We will focus on the implementations of Eiffel, C++, Java and C# statically typed programming languages, analyzing the problems and the existing solutions. The interesting points of discussion are name clashes, duplicating and sharing features, dynamic binding issues.

In Eiffel, multiple inheritance occurs even when the same parent class is inherited twice by the same subclass. This is also known as the repeated inheritance case [Mey02]. If a class has no parents declared, implicitly is considered that it inherits from class *ANY*, which is the base class of any user defined class. In practice multiple inheritance is used in describing the basic data structures implemented in the base library of the Eiffel language. Some researchers sustain the idea that multiple inheritance is a dangerous and destructive concept [Mey02]. This is not a justified opinion, but in practice it results from imperfect implementations and its improper

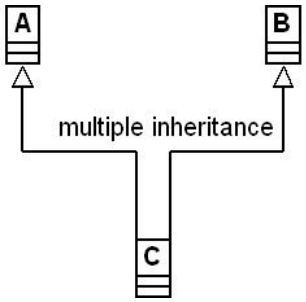


Figure 2.1: Multiple Inheritance

Example 1 Multiple Inheritance Name Clashes

```
class LONDON
 feature foo:INTEGER;
end
class NEW_YORK
 feature foo:REAL;
end
class SANTA_BARBARA inherits
 LONDON
 NEW_YORK
feature
 ...
end
```

uses. When using it properly, it permits combining abstractions, being a key technique in object-oriented development. As graphical convention, multiple inheritance can be represented like in figure 2.1. We used the UML notation to show that class C has two parents A and B.

In C++ this kind of multiple inheritance is named also as **independent multiple inheritance** [Str02], because there is no dependency between the superclasses. This name allows separating from the case of repeated inheritance.

In multiple inheritance one technical problem is the **name clash**. This happens when several features with the same name are inherited from different parents. In the Eiffel philosophy, features cannot be overloaded within a class [Mey02], each feature has a unique name, but even in languages which support overloading (like C++ or Java) the conflict persists for features with identical signatures. Such a name clash situation is represented in example 1, taken from [Mey97].

Name clash is produced if both classes *LONDON* and *NEW_YORK* have a same named feature *foo*, for example. Because the problem appeared in the descendant class, it is motivated that the solution's place is also in the descendant. So the renaming mechanism can be used to solve such a conflict. There are several solutions for the problem of the example: to rename the feature inherited from *LONDON*, to rename the inherited feature from *NEW_YORK* or to rename both inherited features. Of course, the new names chosen have to be unique at the subclass visibility level, otherwise another name conflict could be caused. It is worth mentioning that as long as the conflicting features are not used there is no conflict declared by the compiler.

In C++, multiple inheritance presents the same problem of name clashes, but a different solution is used, the one of explicit designation [Str02, Str97]. The individual selection of one or another inherited feature with the same name, is made with the help of the full qualification. So, the resolution operator "::" is used in this sense. We will revisit the same example in the context of C++ (example 2):

It can be noticed that in the subclass, using the name of the superclass for the inherited

Example 2 Multiple Inheritance Conflict Resolution in C++

```
class London
{
 int foo;
};
class NewYork
{
 double foo;
};
class SantaBarbara: public London, NewYork
{
 // access example London::foo;
 // access example NewYork::foo;
};
```

Example 3 Multiple Inheritance Conflict Resolution in Java

```
interface BaseColors
{
    int RED = 1, GREEN = 2, BLUE = 4;
}
interface RainbowColors extends BaseColors
{
    int YELLOW = 3, ORANGE = 5, INDIGO = 6, VIOLET = 7;
}
interface PrintColors extends BaseColors
{
    int YELLOW = 8, CYAN = 16, MAGENTA = 32;
}
interface LotsOfColors extends RainbowColors, PrintColors
{
    int FUCHSIA = 17, VERMILION = 43, CHARTREUSE = RED+90;
}
```

features, they can be distinguished without any problems. In case of a more complex hierarchy the resolution operator can be used repeatedly.

In Java[AG00] multiple inheritance is possible for types but not for classes. In other words this means that interfaces can be multiply inherited while classes cannot. For classes there can be used only single inheritance, this being a way of avoiding the multiple inheritance problems. In the case of interfaces conflicts may appear. If two superinterfaces declare a field[1] with the same name in each of them, then in the common subinterface any reference to that field causes ambiguity errors [AG00]. Multiply inherited interface fields through different inheritance paths are unified into a single feature [AG00].

We took an example from Java Language Specification book [AG00] in order to exemplify the two possible conflict situations that may arise. Fields *RED*, *GREEN*, *BLUE* are multiply inherited by both interfaces *RainbowColors* and *PrintColors* from *BaseColors* interface. Then they are inherited into *LotsOfColors* interface through multiple paths. The access "*CHARTREUSE = RED+90*" will not create ambiguities since *RED* member is unified at the subinterface level. It is not the same case for *YELLOW* member defined in *RainbowCollors* and in *PrintColors* with different values. A potential reference to *YELLOW* member in the subinterface will determine a name conflict. It is not clear which features should be taken into account: the one with value 3

[1] In Java all fields declared in interfaces are public, static and final, meaning that they behave like constants.

Figure 2.2: Direct Repeated Inheritance

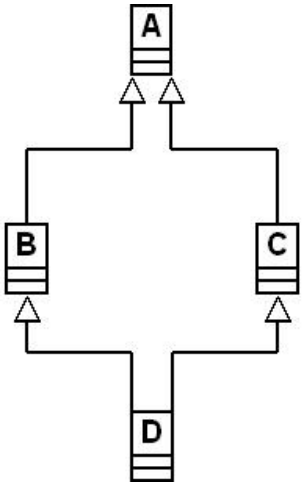


Figure 2.3: Indirect Repeated Inheritance

or the one with value 8.

2.1.1 Repeated Inheritance

In the context of multiple inheritance we encounter the case of repeated inheritance, because a class can be the descendant of another in several ways [Mey02]. There are two cases of repeated inheritance: **direct repeated inheritance** and **indirect repeated inheritance**, like in figures 2.2 and 2.3:

Repeated inheritance arises when two or more parents of D have a common parent A. We have to state that direct repeated inheritance is not allowed in all languages. For example in Eiffel it is possible but in C++ it is not. Class D is the repeated descendant of A and A is the repeated ancestor of D. There are two problems about repeated inheritance which must be solved: the fate of the repeatedly inherited features and the solutions in the resolution of dynamic binding ambiguities [Mey02]. The repeatedly inherited features could be **replicated** meaning that there will be one copy for each inheritance path or **shared** meaning that one unique copy will be inherited. The solution proposed in Eiffel [Mey02] is to be able to decide for each feature independently how to deal with it. This can be done using the renaming mechanism in the following way: shared features will have the same name in the subclass and replicated features will have different names. If we want to replicate repeatedly inherited features we have to change their names using the renaming mechanism. The implicit behavior is the sharing of repeatedly inherited features. In [Mey02] there is an example about a situation where one feature should be replicated and one should be shared (see figure 2.4). Class *HOUSE* has two members *street_address* and *insured_value*. Then two

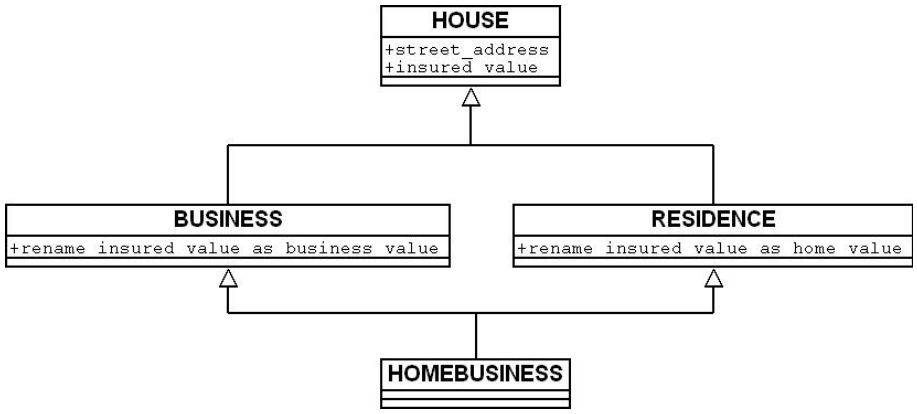


Figure 2.4: Replicated and Shared Features in Repeated Inheritance

Example 4 Repeated Inheritance in C++

```
class L {...};
class A: L {...};
class B: L {...};
class C: A, B {...};
```

subclasses are created *BUSINESS* and *RESIDENCE*. Later a class *HOMEBUSINESS* is built as a subclass of both *BUSINESS* and *RESIDENCE*. It is a natural fact that one person can have a business at his residence so the *street_address* can be unique in the *HOMEBUSINESS* subclass but *insured_value* should be duplicated since two insurance policies have to be made one for the residence and one for the business. In order to obtain such an effect feature *insured_value* will be renamed, using names as *business_value* and *home_value*, in both *BUSINESS* and *RESIDENCE* classes.

In C++ there is one global selection possibility for the multiply inherited features [Sch98, Str97]. Implicitly multiple sub-objects are created when such a subclass is instantiated. Example 4 presents such a situation where class L is the superclass of A and B. Later class C is created as a subclass of A and B. Under these circumstances an instance of C will contain a sub-object L corresponding to A and another sub-object L corresponding to B.

C++ offers also the other possibility of sharing features. This can be done by declaring the base classes as virtual, so any virtual base class will generate a single sub-object [Str02]. In example 5 we studied the behavior of the virtual base classes mechanism. In this case we decided to obtain in class C only one copy of the features from the base class L and in class D two copies: one inherited from C and the other directly inherited from L.

This mechanism has two particularities. One is related to the lack of flexibility because it is not possible to have at the same time features which are replicated and features which are shared. The second observation is the fact that in order to obtain the sharing behavior in the subclass it is necessary to affect the superclasses by declaring them as inherited virtually. The problem is that

Example 5 Virtual Base Classes in C++

```
class A : virtual L {...};
class B : virtual L {...};
class C : A, B {...};
class D : L, C {...};
```

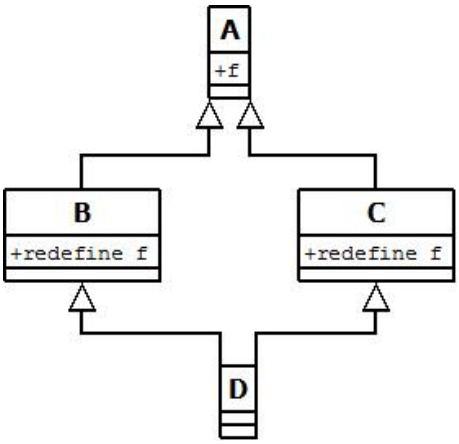


Figure 2.5: Redefined Features in Repeated Inheritance

Example 6 Deferring Multiple Inherited Features

```
class D inherit
 B undefine f end
 C
end
```

not always such a decision can be foreseen.

Dynamic binding problems occur in situations like the one presented in figure 2.5 where the redefinitions of a repeatedly inherited features are made. In Eiffel features can be redeclared [Mey02], this can imply redefinition or effecting. Redefinition means that a feature which lives in the superclass, gets a new implementation or a new covariant signature or a new set of assertions in the subclass. Redeclaration happens even if a deferred feature in the superclass is effected in the subclass. The problem of dynamic binding appears when on an instance of class D, referenced through a variable of type A, the feature f is called. Because feature f is redefined in B and C classes it means that there are two implementations available and name conflict arises. There can be analyzed two cases: one in which features are intended to be **shared** and one in which features are intended to be **duplicated**. When **sharing** features the only possibility to eliminate the name conflict is to deffer (make abstract) all conflicting features except maybe one. In this way only one implementation will be available for feature f in class D.

In example 6 we show one possibility of deferring feature f at the level of class D. It is undefined the feature coming from class B, while in class D the implementation of class C will be used. So, any call to feature f on a D instance will be linked to the implementation of its non-undefined version. The choice of undefining all f features makes the class valid, but if there are at least two implementations of feature f propagating in class D, it will be invalid. Of course, as an alternative, we could undefined feature f on the branch of class C.

The second case, the one of **duplication** implies the renaming of the different implementations of feature f in D subclass (see example 7). In class D, the implementations written in classes B and C are renamed as fb, respectively as fc. Thus there are no name clashes at the level of class D.

Let's analyze all the possible dynamic linking possibilities depending on the reference used on the D typed instance. Example 8 uses the same reference type as the type of the object. In this case the possible calls are to fb and fc features which will be linked to the versions of class B and respectively C.

Example 9 uses references of type B and C, so the versions called on the D object are determined

Example 7 Replicating Multiple Inherited Features

```
class A
 feature f
end
class B inherit
 A redefine f
 ...
end
class C inherit
 A redefine f
 ...
end
class D inherit
 B
  rename f as fb end
 C
  rename f as fc end
end
```

Example 8 Multiple Inheritance Dynamic Binding Case (1)

```
d:D;
create d;
d.fb; -- calls the version of f from class B
d.fc; -- calls the version of f from class C
```

by the type of the reference.

Example 10 uses an A typed reference on a D typed object. The call to the f feature will be ambiguous.

The dynamic binding problem is more severe in this case since there are two versions of the feature in the subclass. A call to the f feature on a D typed object will remain ambiguous unless some criteria is used in favouring one of the implementations. Several solutions are discussed in [Mey02]. The first solution is to use the implementation of the class which is the first listed in the inheritance clause. This approach would change the semantics of a class when changing the order of the inheritance clauses. On the other hand when dealing with more complicated class hierarchies having several features it will lead to impossible situations of selecting different implementations from different superclasses. The second solution is to use a special **select** keyword in the language, which allows to declare directly the choice in the favor of one implementation. The third approach comes with the idea of disabling polymorphism on the several inheritance paths, except one from where the implementation will be achieved. In example 11 on the inheritance path corresponding to class C the polymorphism is disabled using the **expanded** keyword. This means also that the subtyping class relationship between instances of D and C are cut.

Example 9 Multiple Inheritance Dynamic Binding Case (2)

```
d:D;
create d;
b:B;c:C;
b=d;c=d;
b.f; -- calls the version of f from class B
c.f; -- calls the version of f from class C
```

Example 10 Multiple Inheritance Dynamic Binding Case (3)

```
d:D;
create d;
a:A;
a=d;
a.f; -- this call is ambiguous
```

Example 11 Disabling Polymorphism

```
class D inherit
 B
  rename f as fb end
 expanded C
  rename f as fc end
end
```

2.1.2 Implementations of Multiple Inheritance

Several implementation techniques of the multiple inheritance concept in different types of programming languages are presented in [CMR02]. These techniques involve class hierarchy transformations in order to integrate this concept in languages with single inheritance and even with no inheritance. These transformations intend to maintain as much as possible the model of the original class hierarchy, to respect the polymorphic behavior of strongly typed languages and to avoid excessive code repetition.

There are several basic transformations available for different kinds of inheritance models. For languages having no inheritance there can be performed translations like emancipation, variant types or simulation using flags, composition. When dealing with a language having single inheritance there are possible techniques of expansion or mixed techniques. In case of single subclassing and multiple subtyping (which is the case of Java [AG00] and C# [FPB+02]) a mixed strategy has to be used. In the case of languages with multiple inheritance the only concern is the conflict resolution mechanism. Of course, the techniques presented in the context of languages with no inheritance can be applied also to languages with single inheritance, single subclassing and multiple subtyping, multiple inheritance. These basic transformations will be exercised on a demonstrative multiple inheritance class hierarchy.

Figure 2.6 presents a representative hierarchy which will be transformed using each of the transformations previously enumerated. It can be noticed that there is a complex case of repeated inheritance with multiply inherited features through several inheritance paths. Attribute *atrA* is inherited from A to D through B and C classes. On the other hand method *methA* is overridden and has different implementations in each subclass of the hierarchy. We have to mention also that from the typing system point of view there are some subtyping class relationship in this class hierarchy: B and C are subtypes of A, and class D is a subtype of both classes B and C.

Emancipation

The strategy of emancipation [CMR02] involves cutting all inheritance links between classes and including all exhibited features as own resources. This strategy is also known as flattening [Mey97]. A special attention has to be given to the several versions of a method and its "super" like calls. All these have to be renamed in order to keep the behavioral consistency of the methods. In figure 2.7 we can see the effects of the transformation. Each class is independent, it has no inheritance links, inherited attributes are duplicated for each class. Inherited methods are included in the subclasses as own resources and renamed at the same time. The delegation of the "super" like calls are not visible in this representation. One can notice that the natural subtyping class relationship between the classes is lost by using such a transformation.

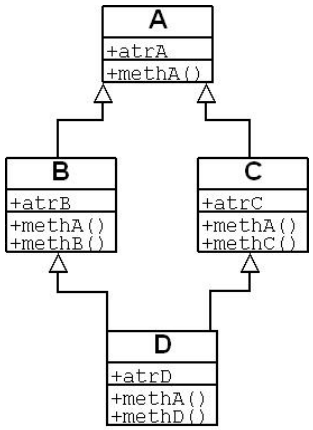


Figure 2.6: Multiple Inheritance Class Hierarchy

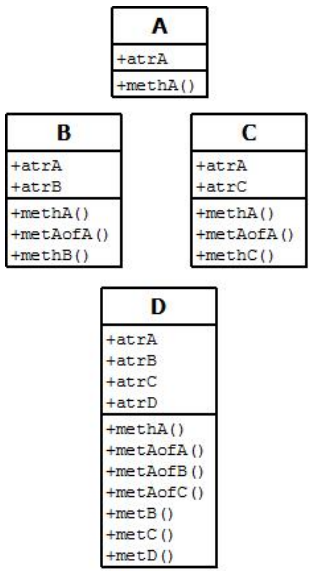


Figure 2.7: Emancipation

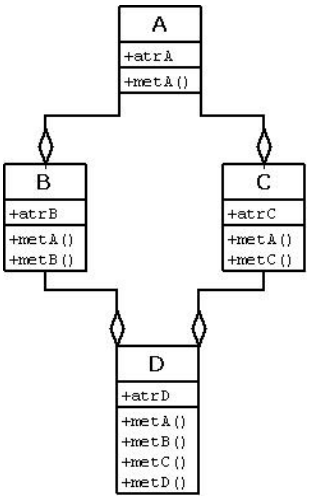


Figure 2.8: Composition

Composition

A different approach to transform multiple inheritance into something more simple is to use composition [CMR02]. This approach is based on the fact that if a class needs some services from another class it is either a subclass or it is a client of that class [Mey97]. The transformation consists in transforming all inheritance links in composition links. The former subclass will be composed out of references to instances of the former superclass. Obviously, all "super" like calls have to be delegated to the corresponding component objects. The subtyping relationships between classes is lost also with this kind of transformation. In figure 2.8 it can be noticed that each inheritance link is replaced with a composition link. This transformation is based also on the fact that a class can have an unlimited number of composition links while the inheritance links could be limited to one (in single inheritance based language) or even zero (in the case of procedural languages). Class D is exhibiting all inherited methods from superclasses implementing them by delegation. The advantage of this transformation is that there will be no code or data duplication, subtyping being lost though.

Expansion

The idea of this transformation is to use the benefits of single inheritance and to transform the multiple inheritance DAG (directed acyclic graph) into a tree or forest in the general case. Each inheritance path is isolated by duplicating the multiply inherited class. In figure 2.9 the hierarchy is transformed using expansion. Class D is duplicated into *D1* and *D2* on each inheritance paths. It can be noticed that features which should be inherited normally from the other branch are added as a own resource into the subclass. It is the case of method *metC* and *metB* of class C and respectively B which has to be added into class *D1* respectively *D2*. The other features are inherited using the single inheritance: *atrB*, *metA*, *metB* for class *D1* and *atrC*, *metA*, *metC* for class *D2*. In conclusion we can say that some of the subtyping class relations are kept, naming the ones between *D1* and B or *D2* and C, but the others not.

Variant Type

The variant type idea or simulation of variant type comes from procedural programming languages where no polymorphism is available. Simulation is made using a single monitor class which has all the features of all classes and by switching a flag the different types can be achieved [CMR02]. Depending on the current object type a certain set of features is exhibited. The obtained structure

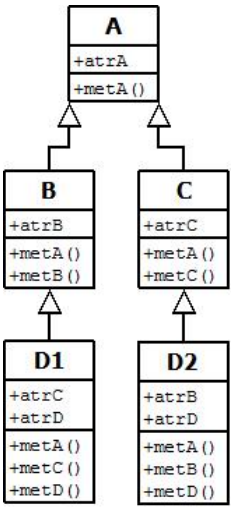


Figure 2.9: Expansion

Example 12 Delegation Example in C++

```
class B { int b; void f(); };
class C : *p { B* p; int c; };
```

is relatively complex and it involves no data or code duplication. In figure 2.10 we have the transformation applied and a monitor class is created instead of the whole multiple inheritance hierarchy. There has been added a flag called *whoAmI* which can set the difference between the several object types simulated by this class. All the attributes and methods (including all variants) of the hierarchy are centralized in this class. In order to emulate the original behavior of the multiple inheritance hierarchy the set of exhibited methods *metA*, *metB*, *metC*, *metD* will call the appropriate versions depending on the state flag.

2.1.3 Delegation

As a similar concept with the concept of inheritance, we will discuss about the delegation class relationship in C++ [Agh86, Str02]. The idea is that in the base class list of a class declaration there can be specified a pointer to some other class. Example 12 shows such a mechanism.

Class C is defined as having superclass class B, but this link is expressed using a pointer to the superclass sub-object. Example 13 explains that any call to an inherited member of the subclass instance will be treated as if it would be defined in that subclass.

The advantage of this technique is the possibility of changing the superclass sub-object at runtime. With normal inheritance there is no such facility, the superclass sub-object can be referred using the *this* pointer which cannot be assigned with the address of the new sub-object. Because of the bugs and confusion encountered by the users of this mechanism, it was never included in the C++ language.

Example 13 Delegation Usage in C++

```
C* q;
q->f(); // is equivalent with q->p->f();
```

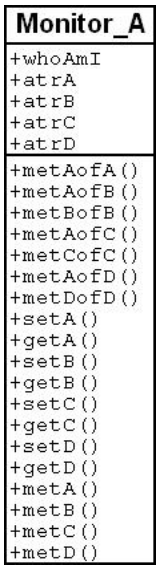


Figure 2.10: Variant Type

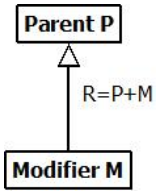


Figure 2.11: Incremental Modification by Inheritance

2.2 The Like-Type Class Relationship

Incremental modifications can be used in reusing conceptual or physical entities and in the construction of new similar ones [WZ88]. As natural and computational systems evolve, incremental mechanisms controls their evolution. Inheritance class relationship is a particular kind of incrementation mechanism which transforms the parent P with the help of a modifier M into a result entity R=P+M.

The "+" composition operator is asymmetrical since P and M have different roles in this class relationship. The features of M may overlap the features of P.

Compatibility rules can be set between subclass and superclass [WZ88].

Cancellation allows operations from the superclass to be deleted from the subclass.

Name compatibility allows changing the names of features but no features are deleted.

Signature compatibility guarantees the compatibility between superclass and subclass interfaces.

The **behavioral compatibility** assumes that the subclass will not define a radically different behavior from the one in the superclass.

The first three rules refer more to the syntactical part of inheritance and can be easily checked. This form of inheritance is known as **non-strict inheritance**. The fourth rule cannot be easily guaranteed. Some assumptions regarding the subclass behavior can be issued, but not an absolute verification. Assertion mechanism is a step in this direction of guaranteeing behavioral compatibility. This form of inheritance is known as **strict inheritance**.

Table 2.1 presents the four incremental mechanisms and their corresponding class relations

Incremental Mechanism	Class Relations
behavioral compatibility	R subtype P or R is-a P
signature compatibility	R subsig P
name compatibility	R subclass P
cancellation	R1 like R2

Table 2.1: The Four Incremental Mechanisms

Example 14 General Form of Mixin in C++

```
template<class Super>
class Mixin : public Super
{
 /* body of the mixin */
};
```

[WZ88]. It can be noticed that **like** is the most general relation which includes all the rest. Types which are related by the like relationship are called **liketypes**, as subtypes is the name for types related by the subset relation. The subtype relation is asymmetrical while the liketype is symmetrical. If any type R1 like R2 then R2 will be like R1.

2.3 Mixins

Software complexity gave birth to methodologies which divide the problem in solvable parts, after they have to be composed in the final software product[SB00]. The mixin mechanism is based on two other concepts: **inheritance** and **genericity**. Mixins are derived from generic programming and they are generic classes which have as generic parameter types their superclasses. The basic idea was to specify an extension without being obliged to specify the unit to be extended. This is equivalent to specifying the subclass letting the superclass as a parameter which will be specified later.

Mixin in C++ is a high power technique for using multiple inheritance with abstract virtual base classes to enable incremental development of both interfaces and implementations of classes [Ska93]. Mixins are considered a great achievement in C++, although it was not intended to enable it in the language.

The advantage of this approach is in the fact that there is only one class used for a valid incremental extension specification for a variety of classes. In [VN96] it is shown how mixins are used to create a role based design. Mixin layers is a derived mechanism from the mixin concept and it was made for concern modeling. They are nested mixins in which the parameter of the external mixin will determine the parameters of the internal mixin [SB02].

2.3.1 The Mixin Concept

In this subsection we will focus on the implementation of mixins in the C++ programming language. The basic idea is to define an extension without knowing a priori what is extended. This implies the specification of a subclass while the superclass will be specified later as a generic parameter [SB00]. Mixins can be implemented using parameterized inheritance. The superclass of the mixin will be specified as a parameter which will be specified at the instantiation moment. In C++ such a mechanism can be expressed like in example 14.

We will exemplify the counting operation on a graph data structure. The operation involves counting how many nodes and edges were visited during the execution of the example.

Example 15 shows how the operations of the designed mixin interfere with the operations of the graph modeling class. The counting operations from the mixin in the example can be applied

Example 15 Graph Counting Mixin Example in C++

```
template <class Graph>
class Counting : public Graph
{
 int nodes_visited,edges_visited;
 public:
 Counting():nodes_visited(0),edges_visited(0),Graph(){}
 node succ_node(node n)
 {
  nodes_visited++;
  return Graph::succ_node(n);
 }
 edge succ_edge(edge e)
 {
  edges_visited++;
  return Graph::succ_edge(e);
 }
 ...
};
```

Example 16 Using Mixins Example in C++

```
Counting <UGraph> counted_ugraph;
Counting <DGraph> counted_dgraph;
```

to all classes having the same interface (see example 16).

In example 16 there are instantiated two graph objects: one undirected and the other directed. Both objects have the facility of counting nodes and edges. These counting facilities were not included originally in the graph modeling classes but were achieved by setting the actual parameter of the *Counting* mixin to *UGraph* and *DGraph* classes. The mixin is suitable to all classes which have the same interface as the graph modeling classes.

2.3.2 The Mixin Layer Concept

In this subsection we will focus on the implementation of mixin layers in the C++ programming language. Mixin layers are a particular form of mixins. They are designed to encapsulate refinements for multiple classes [SB00]. They are nested mixins so the parameters of the external one will determine the parameters for the internal mixin. The general form of a C++ mixin layer is presented in example 17.

The conceptual unit here is not the object or parts of it. The mixin layer can specify refinements for more than one object. Inheritance is used in order to compose extensions. Mixin layers are used for implementing roles. Each layer will capture one collaboration. The roles for all the

Example 17 General Form of Mixin Layers in C++

```
template <class NextLayer>
class ThisLayer : public NextLayer
{
 public Mixin1:public NextLayer::Mixin1{...};
 public Mixin2:public NextLayer::Mixin2{...};
};
```

participant classes are represented by the internal classes of the mixin layer. Inheritance works at two levels. First the layer inherits all the inner classes from the superclass. Then, the internal classes inherit attributes, methods and even classes from the internal classes of the corresponding mixin layer superclass. In this context the layer behaves like a name space.

2.4 Traits

Traits are simple mechanisms for object-oriented systems organization based on mixin components. A trait is a parametric set of methods, which can be assembled in classes, representing the primitive entity of reuse. Using traits, classes can be organized in hierarchies based on single inheritance and can be used also in specifying the incremental difference between subclass and superclass. For this mechanism, inheritance is not the composing operator like for multiple inheritance or mixins, because it has its own composition operators.

2.4.1 Motivations

Motivations around this concept attack the weaknesses of the concept of inheritance [SDN02]. First of all inheritance cannot factor common features from complex class hierarchies. This gave birth to the multiple inheritance class relationship. Mixins, discussed in subsection 2.3, are a way of composing classes incrementally starting from sets of members. It is admitted in [SDN02] that in practice there are a lot of problems with these mechanisms. One cause are the conflict resolutions when inheriting the same feature on several inheritance path. As it was presented in subsection 2.1, the solutions involve linearizing or renaming which makes the desired behavior hard to achieve. It is stated also that reusable artifacts are hard to design without conflicts. Class hierarchies based on inheritance suffer also from the fragile base class problem [MS98]. Changes in the class hierarchies affect the conflict resolution mechanisms causing anomalies.

2.4.2 Classes and Traits

In [SDN02, SDNB03] is presented a solution to the earlier invoked problems. There is a separation between the concepts of traits and the concept of class. Traits offer a set of services (methods) which implement the behavior but not state (attributes). Traits can depend further on other traits. Traits have no direct access to state, but using accessor methods. In conformance with this model, a class can be built starting from a set of traits and providing the necessary state and the missing services. The missing services represent the linking code which specifies how traits are connected and how possible conflicts have to be solved.

The traits model can be applied to several types of programming languages. The traits description will be made in the context of a single inheritance programming language. The model of traits is presented in figure 2.12. In this model, traits are designed to be the most primitive reuse code unit. Traits are designed to offer and to request services. The requested services are named **connectors**, while the offered services are named **sockets**. The sockets in the example are *area*, *bounds*, *scaledBy*, and the connectors are *center*, *center:*,*radius*, *radius:*.Between traits there is no inheritance allowed. Connectors have to be connected in the moment of using the trait. A class can be obtained by the composition of zero or several traits. The class will have to offer the state plus the extra functionality in order to assimilate such a trait.

From the semantical point of view the whole trait functionality will be incorporated in the interior of the class as if it would be declared there initially. There are exceptions in the case of a method which is implemented in a class and in a trait, the method implemented in the class has the priority. It was decided that all traits have the same priority in case of name collisions.

2.4.3 Composing Traits Use Case

In figure 2.13 is analyzed an example from [SDN02] of how traits can be composed. Also, the case of a conflicting feature is considered.

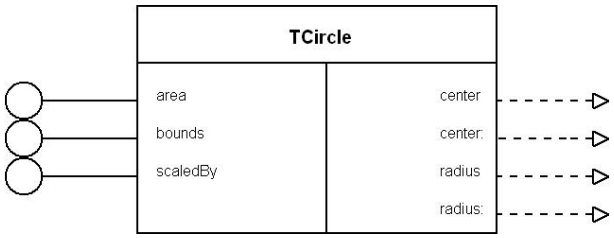


Figure 2.12: Traits Model

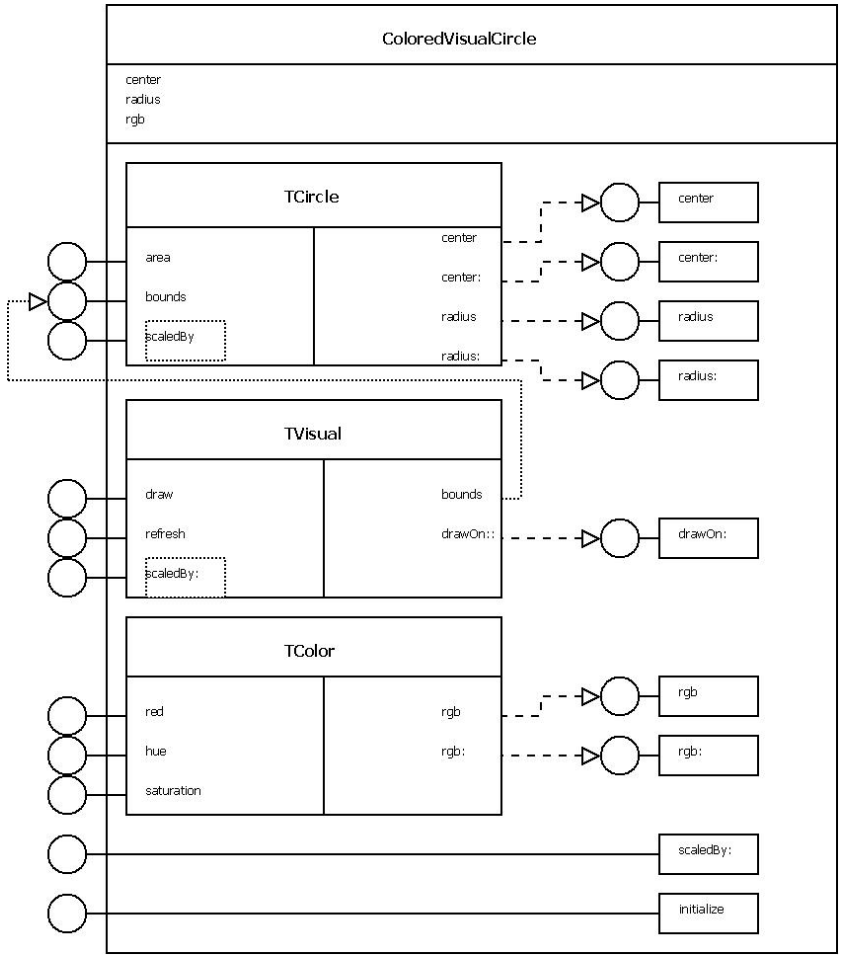


Figure 2.13: Traits Use Case

In the example depicted in figure 2.13 a class is built starting from three traits: *TCircle*, *TVisual*, *TColor*. Each trait has embedded the necessary functionality to produce a *ColoredVisualCircle* class. Trait *TCircle* has three sockets: *area*, *bounds*, *scaledBy* and three connectors *center*, *center:*, *radius*, *radius:*. The center and radius related connectors will be plugged into the class defined features. Trait *TVisual* has three sockets *draw*, *refresh*, *scaledBy* which are also exhibited at the class level. In exchange, it needs services like *bounds*, which are provided by the *TCircle* trait and a *drawOn:* method which is implemented in the class. The *TColor* trait connects only with the class features *rgb* and *rgb:*. Separately from the already presented features, class *ColorVisualCircle* has two features *scaledBy:* and *initialize*. There can be noticed that the *scaledBy:* glue feature is provided by two of the traits *TCircle* and *TVisual* so in the composing class there will be defined a new version, thus eliminating the name conflict.

2.4.4 Traits vs. Multiple Inheritance

In this subsection we compare the traits mechanism with another reuse mechanism of the object-oriented technology: multiple inheritance. Traits and inheritance can be combined together in a constructive way. Since there is no inheritance between traits, any "super" like call from one of its methods, is linked to the method of the composing class parent. Comparing the traits mechanism with multiple inheritance it can be admitted that there are some similarities and some differences. The starting point for both mechanisms is the combination of reuse entities. The composing mechanism is semantically the same: feature reunion. It was admitted that features from traits will be incorporated in the composing class. With multiple inheritance, which involves subtyping and also subclassing, the same feature composition can be obtained. There can be noticed a difference from the technical point of view. A class in order to be valid is obliged to have all its external calls resolved, while a trait will be connected in the moment of composition. A potential problem of traits model is that it implies developing from scratch all the reuse artifacts, there is no decomposition mechanism for the already existing classes. The "diamond problem" of multiple inheritance analysed in subsection 2.1 appears in the case of traits model. The authors of [SDN02] claim that since there are no attributes in the structure of traits, there are no conflicts. The possible method conflicts are solved by declaring the conflicted method in the composing class or allows the programmer to favor one mixin to implement a certain service [SDN02].

2.5 Role Programming

In this section are presented the main concepts of role programming: roles and collaborations. Several implementation techniques are discussed also.

2.5.1 Roles

Role programming allows the decomposition of the object into several roles. Roles are abstracting the concerns and formalizing their separation. From the collaboration point of view, roles are parts of objects which fulfill their responsibilities in the collaboration [HN96]. Roles are encountered in many practical situations. Taking the example of a university student, sometimes he can be a football and a basketball player. In some special situation he can became a member in the university council. After a period he can quit this memberships. So objects in object-oriented systems may behave the same way like real-world ones do. Dynamically, during their life time several behaviors can be attached and detached to them [TUI05]. Roles are a volatile concept in the implementation since they do not generally exist as an identifiable component [HN96]. In [Kri96] the properties of roles are presented. **Abstractivity** facilitates the roles to be organized in hierarchies. **Aggregational composition** is the property of roles which results from the fact that roles can be composed with other roles. **Dependency** property states that a role cannot exist without an object. **Dynamicity** refers to the fact that roles can be added or removed during the lifetime of an object. **Identity** of role is the same with the identity of object. **Inheritance**

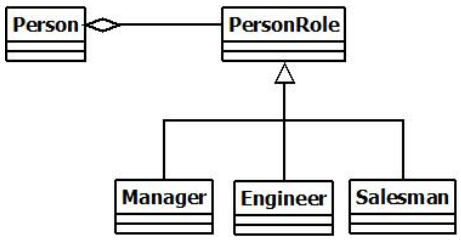


Figure 2.14: Role Object

in the context of roles refers to the fact that a role for a class will be a role for any subclass. The **locality** property gives meaning to a role only in a role model. **Multiplicity** property states that several instances of a given role can exist for an object at a given time. **Visibility** property of roles means that a role can restrict the access to an object.

2.5.2 Collaborations

In relation to roles, collaborations also have to be discussed. Collaborations involve the cooperation of a group of objects which perform a task or maintain an invariant [HN96]. The main objectives of roles are to describe the collaboration of objects and to delimit well their boundaries [TUI05]. Objects may be involved in multiple collaborations having different roles. Initially collaborations were described by specifying use cases and observable behavior of the objects. The use case idea originated in [JCJO92] was adopted by UML [OMG04]. In UML roles were denoting directions of static associations (Association Roles) and afterwards they were related to collaborations (Collaboration Roles). Roles by a set of behavioral functions are able to delimit the boundaries of objects and thus their granularity is smaller, being very close to the level of methods.

2.5.3 Role Implementation Techniques

In the work of [Fow97] several role representations and corresponding design patterns for implementation are presented. A design pattern, as presented in [GHJV97], is a general solution to a class of software architectural problems. The proposed role representations are: single role type, separate role type, role subtype, role object, role relationship. **Single role type** is a solution where all features of the roles are combined in a single type. **Separate role type** implies that for each role a separate type has to be created. **Role subtype** solution organizes the roles in a hierarchy. Each role has its own type and common behavior can be put in the supertype. **Role object** pattern involves the existence of a host object which has several sub-objects, one for each role. The clients will ask the host object for a specific role feature. In figure 2.14 is presented such a situation. Class *Person* has several roles like manager, engineer and salesman. For this case a role class hierarchy is designed having as superclass *PersonRole*. Class *Person* will have to use one reference of type *PersonRole* which will refer any role instance. The interface of the *PersonRole* with polymorphism and dynamic binding will allow transparent access to different role implementations of the person's behavior.

In **role relationship** pattern the idea is to make each role a relationship with an appropriate object. Several techniques can help in implementing a role mechanism. One implementation uses only the ordinary features of a regular object-oriented programming language and no other special language extensions. The implementation is based on interface, class inheritance and polymorphism combined with dynamic binding. The proposed strategies are: internal flag, hidden delegate, state object. **Internal flag** pattern, as its name suggests, the object uses the flag to do the behavior selection upon the selected role. **Hidden delegate** supposes that a subobject knows all behaviors implied by roles and can be selected using appropriate messages.

Mixin layers are another way of implementing roles [SB02]. The model of the mixin layer and

Example 18 Role Implementation

```
template <class ChildType,
          class MotherType,
          class Supertype>
class FatherRole : public SuperType
{
 ChildType *child;
 MotherType *mother;
};
```

its role implementation capabilities were presented in subsection 2.3. In the approach of [HN96], a model similar to mixins is presented. Roles are defined using parameterized types. In C++ the implementation can be made with the help of templates. In example 18 class *FatherRole* has two generic parameters *ChildType* and *MotherType* denoting the two collaborating roles. The *SuperType* parameter is used with the same purpose as it is used in mixins, to have a link with the class which will contain the father role.

In [GSR96] in order to allow objectual database evolution, along with the class hierarchy of the objectual paradigm, a role hierarchy is created. A role hierarchy is a tree of special types, named role types. The root of the tree defines the invariant properties of an entity while the roles types reflect refinements. An entity is represented by an instance of the root type and the instances of every role type that the entity has at a given moment. So the traditional object-oriented concepts are extended with the role hierarchy.

In [Ken99] roles are implemented using the AOP of AspectJ and there are shown the relations between the role approach and the aspect-oriented technology. In the approach of [TUI05] the role model is created on the base of some principles. One of them is the **support adaptive evolution** which means that objects evolve in environments assuming their roles, the participation can be made dynamically: objects can enter or leave environments freely, an object can belong to multiple environments at a time. The second principle is related to the **separation of concerns**, each concern is modelled by an environment. Concerns will interact using objects simultaneously assuming roles of different collaboration environments. The third principle is the **advanced reuse of roles** within environments. Environments and roles have the status of first block constructs in EpsilonJ proposed programming language.

2.6 Composition Filters

In this section a brief presentation of composition filter mechanisms is made. They address the separation of concerns issue. Composition filters are special objects that can be attached to the normal instances in an application, having the role of intercepting incoming and outgoing messages. These filters can be combined freely as they are orthogonal. The filters can change the behavior of an object so concerns can be attached to objects using them.

2.6.1 Motivations

The concept of composition filters appeared in the context of an object-oriented language database integration model [ABV92]. Motivations in this direction are given: **duality in conception** - language and database models are kept separately, **violation of encapsulation** - object queries make object structure visible and are not accessed via send messages only as the object-oriented paradigm requires, **fixed views** - relational databases support views on base tables while from the object-oriented point of view there are methods of an object which are not of interest for any client. The integration of database like features in the object-oriented programming language implied the extension of the object model.

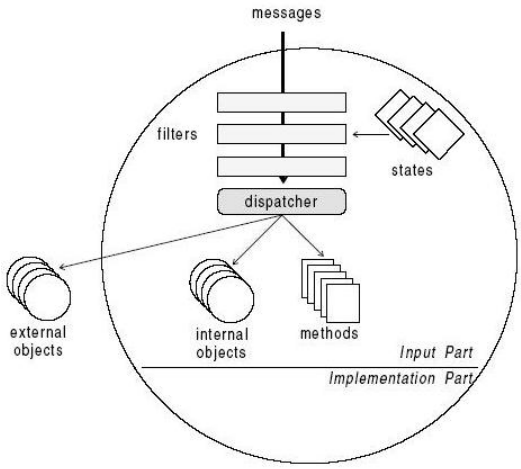


Figure 2.15: Composition Filters Model

2.6.2 The Composition Filters Model

In figure 2.15 the model of the composition filters mechanism is presented. It can be noted that the object model consists in sets of methods, interface objects and states. The interface objects are of two types: internal and external. A set of filters are a part of the object model also. The filters are used to intercept the incoming messages and to dispatch them to the appropriate methods. By methods are meant: a subset of own methods, or methods from the internal or external objects. The states are used to control the behavior of filters.

Composition filters can be used as mechanism to separate concerns.

2.7 Views

A **view** is a description of the system relative to a set of concerns from a certain point of view [Hil99]. The motivation for **multiple views** is **separation of concerns**. They were introduced to manage the complexity of software engineering artifacts like: requirements, specification and design. UML [OMG04] is the best known language which respects the point of view modeling philosophy.

In the world of object database the notion of view has emerged from the relational view solution. Views [KR93] in the relational model provide logical data independence and offers possibilities for data to be repartitioned and restructured to fit particular applications. In addition, the database object views offer the introduction of new classes (called virtual) into the class hierarchy [AB91]. Virtual classes can be populated with objects in several ways: i) the virtual class is a superclass of certain classes (generalization); ii) the virtual class contains all objects returned by a query (specialization); iii) the virtual class contains all objects having a certain behavior.

We will discuss from the several domains point of view how reusability can be achieved. In the object-oriented database world, the reuse is focused intensively on objects. They try not to reuse the structure of the object, meaning the class, but the object itself at runtime. This reuse addresses the extension of the concept modelled through a certain object. In software design and programming languages the tendency is to reuse the structure of the class at design time. This is done in modeling using different concepts like generalization, specialization, and in programming languages by mechanisms like inheritance single or multiple, genericity. So, from these two points of view, the software reuse can be seen at two levels: reuse of class or intension and reuse of objects or extension. The first level is encountered in object-oriented programming languages, while the second level of reuse is exploited in object-oriented databases.

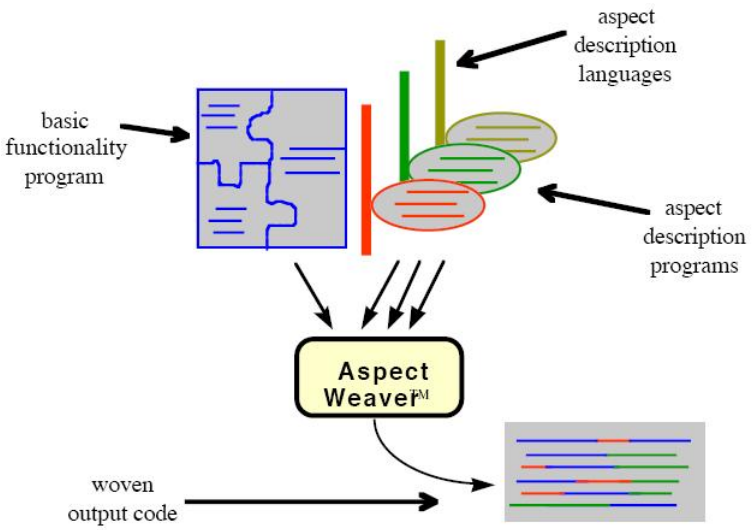


Figure 2.16: Aspect Oriented Programming Main Principle

2.8 Aspect Oriented Programming

Aspect oriented programming [KLM+97] is a separation of concerns model based on object-oriented paradigm. It deals with crosscutting concerns which cannot be well separated by pure object technology. The majority of object-oriented systems are composed out of crosscutting concerns dispersed over several modules. By concern it is meant a concept, a goal in the context of a given domain. For example a concern in the context of debugging a software system would be the logging operations. Another functionality, which can be viewed as a crosscutting concern, needed in the context of objects, is persistence.

There are several concepts of object-oriented programming which facilitate the separation of concerns. First, the abstraction principle implies the creation of separate classes for each concept from the real world [Aks96]. On the other hand the information hiding principle allows interface separation from implementation. Inheritance and delegation are ways of composing behavior. In the context of inheritance, the behavior of the subclass is composed with the behavior of the superclass [Aks96].

In figure 2.16 the main schema of aspect oriented programming is depicted [KLM+97]. Each application has a main part where the basic functionality is captured. This part is supposed to be written in a language that suits better to the application domain. Then each cross-cutting aspects are described using several specialized languages. All these programs are taken by the weaver and it produces the output code. The main property of this methodology is aspect decomposition. Thus, the aspectually decomposed program is easier to develop and to maintain.

Generally speaking, a software system is composed out of several concerns and it is responsible for multiple requirements. In [Lad02] the requirements are classified as **core module level requirements** and **system-level requirements**. The system-level requirements are crosscutting several modules. A class implementing business logic is presented in example 19 taken from [Lad02].

The first observation is that the *other data members* do not belong to the core concern of the class. The *performSomeOperation* method includes along with the core concern, some other operations like logging, authentication, multithread safety, contract validation, cache management. The two operations *load* and *save* having the role of persistence management should not be part of the core concern.

There are two symptoms which indicate the problematic implementation of crosscutting con-

Example 19 Crosscutting Concerns Example

```
public class SomeBusinessClass extends OtherBusinessClass
{
 // Core data members
 // Other data members: Log stream, data-consistency flag
 // Override methods in the base class
 public void performSomeOperation(OperationInformation info)
 {
  // Ensure authentication
  // Ensure info satisfies contracts
  // Lock the object to ensure data-consistency in case other
  // threads access it
  // Ensure the cache is up to date
  // Log the start of operation
  // ==== Perform the core operation ====
  // Log the completion of operation
  // Unlock the object
 }
 // More operations similar to above
 public void save(PersitanceStorage ps) { }
 public void load(PersitanceStorage ps) { }
}
```

cerns: **code tangling** and **code scattering** [Lad02]. Code tangling happens in modules which interact simultaneously with several requirements. Code scattering refers to concerns spread over the software system modules.

AspectJ is an implementation of the aspect oriented paradigm in Java developed by Xerox Parc, Inc. The concepts for describing the extensions are: **pointcuts**, **join points**, **advices** and **aspects** [Tea03]. Join points are certain well defined points in the execution of the programe. The pointcut is a language construction that specifies several join points. Advice is a piece of code executed when a joint point is reached, it brings together a pointcut and a body of code, to run at each join points. The aspect is a unit of modularity for the crosscutting concern.

2.9 Classboxes

In this section we present a class adaptation mechanism, called **classboxes** [BDW03], which allows adding and replacing methods in a class. The changes made by a classbox are only visible in that classbox or other classboxes importing it. This feature is named **local rebinding**. Classboxes are modules or software units that consist of imports and definitions. An import is either a class import or a classbox import. A definition can be a class or method definition. The method definition must specify the class it belongs to and also the implementation.

In example 2.17 taken from [BDW03] the *Squeak* classbox contains all the classes from the standard library. The aim of the example is to reuse the HTML node classes from the library and to add to check all broken links from a web page.

In the *HTMLVisitor* classbox each node class is augmented with the *acceptVisitor* method and a new *HTMLVisitor* class was added to perform URL link checking on the nodes in a systematical manner.

Finally, in figure 2.18 a new classbox is created importing the *HTMLParser* and *Socket* classes. The *LinkChecker* visitor class is derived from *HTMLVisitor*. In the context of the newly created classbox the *HTMLParser* will use the node classes accepting visitors, the *LinkChecker* visitor class will visit each node and if the node is an anchor then the *ping* method from the *Socket* class

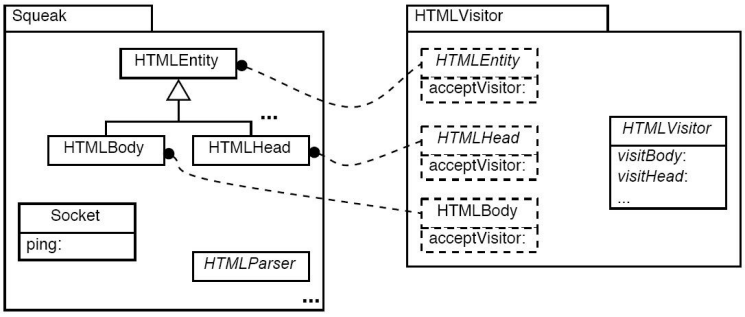


Figure 2.17: Classbox Example (1)

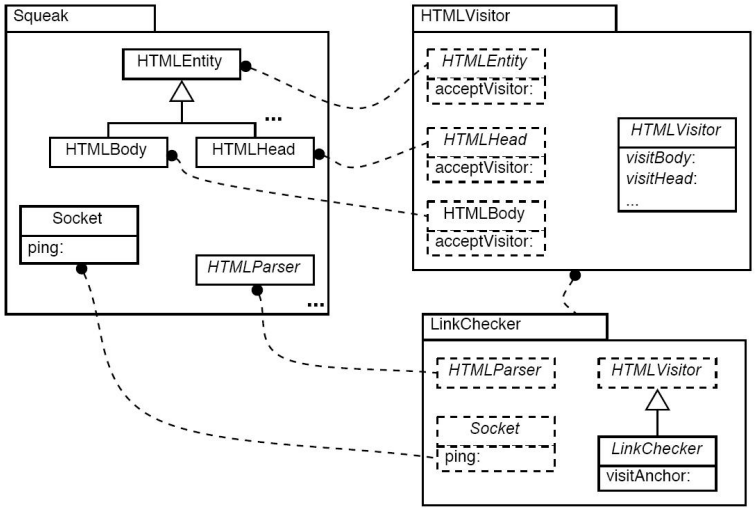


Figure 2.18: Classbox Example (2)

Example 20 An Expression Class Hierarchy

```
class Expr
{
 abstract void display(OutputStream os);
}
class Plus extends Expr
{
 Expr op1();
 Expr op2();
 void display(OutputStream os) {/*...*/}
}
class Value extends Expr {}
class Int extends Value
{
 int intValue() {/*...*/}
 void display(OutputStream os) {/*...*/}
}
class Flt extends Value
{
 float fltValue() {/*...*/}
 void display(OutputStream os) {/*...*/}
}
```

will be used to determine the broken status of a link within a certain timeout.

2.10 Expanders

The expander is an object-oriented programming language construct which supports object adaptation [WSM06]. Classes are adapted in a non-intrusive manner by adding new attributes, methods and superinterfaces. Each client can adapt the same class in different contexts independently with different expanders. Expanders have several design properties in order to support object adaptation.

In-place Adaptation An expander adds to a class new state, behavior and superinterfaces rather than creating a new subclass. Multiple clients are allowed to view the same object in different ways.

Statically Scoped Adaptation Expanders adapt existing classes to the needs of new client without affecting the behavior of old clients. A client can explicitly import the expanders it needs in order to fulfill its task. On the other hand all other expanders are out of scope and cannot affect the client behavior. Clients can expand the same object in different ways without any conflict.

Modular Type Safety The existence of multiple versions of a class may cause undesirable behavior at runtime. In the context of inheritance the newly added features in the subclass may conflict with the features added by expanders. Because expanders have a static scope, it is possible to perform modular static type checking to assure type safety.

Expander Usage Example The way expanders work in practice is presented in example 20.

A class hierarchy modelling an AST used in an expression parser is presented in example 20. There are classes modelling expressions, plus operator, values, integers and floats. Each class of the

Example 21 Expander Example

```
// file EX.ej
package eval;
import ast.*;
public expander EX of Expr
{
 public Value eval() { throw new EvalError(); }
}
public expander EX of Value
{
 public Value eval() { return this; }
}
public expander EX of Plus {
 public Value eval()
 {
  Value v1 = op1().eval();
  Value v2 = op2().eval();
  // ...
 }
}
```

hierarchy has a method *display* for printing node information in the output stream. The intention is to augment somehow these classes to get value evaluation behavior of the parsed expression.

The *EX* expander family containing an expander for each class of the AST node hierarchy is presented in example 21. The expander family will add an *eval* method to each class in order to compute the value of the parsed expression.

In example 22 is presented a client of the nodes hierarchy parsing a string and evaluating the expression on the AST in quite a natural manner. In order to use the *EX* expander it must be imported with the special *use* directive. Thus, each node class will be equipped with the *eval* evaluation method.

Example 22 Expander Usage Example

```
// file Calculator.ej
package calc;
import parse.Parser;
import ast.*;
use eval.EX;
public class Calculator
{
 public void process(String s)
 {
  Expr e = new Parser().parse(s);
  Value ans = e.eval();
  ans.display(System.out);
 }
}
```

2.11 Summary

In this chapter we talked about inheritance in general and multiple inheritance in particular focusing on crucial aspects. There are opinions that multiple inheritance is bad and dangerous because of the resolution mechanisms which are sensible to any eventual changes in the hierarchy. Other authors consider that the approach of Java [AG00] is the right way of implementing the concept of multiple inheritance by letting multiple inheritance of types but single inheritance of classes and classes can implement many interfaces. Another important issue is the **resolution mechanism** which could be assisted like in the case of C++ [Str02, Str97] or Eiffel [Mey02] or it could be automatic like in CLOS [Kee89]. In CLOS the class which appears first in the list of superclasses is the one that has priority in the case of conflicts. In Eiffel the conflict resolution decisions can be taken for each conflicting feature separately, while in C++ the "all or nothing" approach is applied: one decision applies to all the features involved. It can be noticed that the Eiffel renaming solution for multiple inheritance name clashes in fact generates the problem of dynamic binding. Such problem does not exist in C++. In C++ to switch between replication and sharing implies placing the **virtual** keyword at a higher level than the one on which the effect occurs. It results that such conflict resolutions must be foreseen in advance, in practice this is not always possible. In Eiffel, in the case of dynamic binding problems, the **select** keyword is used in the subclass where the ambiguity arises. Implementations of multiple inheritance in several programming languages are presented, where some semantical decisions can be learned. In each implementation solution there is a balance between several issues like data and code duplication, type conformance, delegation code insertion which work together for maintaining the original semantics of reverse inheritance.

The mixin and mixin layer mechanisms are studied as possible vehicles for role programming and collaboration based designs. The combination of inheritance and genericity of C++ leads to a mechanism which allows the description of extensions which fit to several classes. In this mechanism, inheritance is the composition operator, while the class plays the role of the composable module. The only reuse restriction for mixins is that they have to be fitted for the superclass interface.

The trait concept which derives from mixins, adhere to the idea of eliminating attributes from the composable modules and letting them build a class by composition. In this case the composition operator is in fact a reunion of all the features from the composing traits. When conflicts arise resolution is provided by redeclaring the feature in the class, thus ignoring the trait originating conflicting features. It can be noticed that the traits based design has to be started from scratch, reusing existing class libraries with such a mechanism is not possible. The main advantage remains though that trait components, once defined, are simple and highly reusable.

Roles and collaborations are other issues described in this chapter. Roles are represented in several implementations as temporary features of an object during its life time. There are no general identifiable components corresponding to roles. They are highly used in object-oriented systems and in object-oriented databases. As role implementations there were presented design patterns, mixins, aspect-oriented solutions, role hierarchies and a language with special roles semantical extension.

In the last part of the chapter the separation of concerns issue is treated. First the composition filters are presented, explaining the basic functionality of the mechanism. Views is another mechanism for concern separation which has its origins in the relational databases. Aspect oriented programming principles and concepts are presented also.

Classboxes and expanders are class reuse and adaptation mechanisms that describe class extensions which are activated only in special contexts, leaving old clients unmodified.

Finally, all reuse mechanisms are compared through several criteria of interest:

- to create new abstract types;
- to factor features from classes;
- to combine the implementations of features;

- to redefine the implementations of features;
- to adapt the implementation of features;
- to cancel the implementation of a feature;
- to add an abstraction layer into a class hierarchy.

In table 2.2 we analyse several class reuse mechanisms through each proposed criteria. Creating abstract supertypes criteria is supported only by exheritance. Factoring common features from classes is supported also, but only by reverse inheritance and not by the other mechanisms.

Combining the implementations of features is possible in several mechanisms. In ordinary inheritance this is possible through language constructs like **super** (Java), **precursor** (Eiffel) where the implementation from the subclass may call the implementation from the superclass. In the context of exheritance the exherited features from different clases can be added in the same subclass where they may be combined. For mixins and mixin layer this criteria is central for the reuse. Class features call methods from the actual generic arguments instances which the class inherits. In AspectJ the join point model allows to define the dynamic structure of crosscutting concerns. Classboxes and expanders favors the creation of special context where old and new features can be combined.

Feature redefinitions are possible in classic and reverse inheritance. Classboxes and expanders allow feature redefinition in separate contexts.

Cancelling the implementation of a feature can be obtained in ordinary inheritance by making the feature deferred (Eiffel), abstract (Java, C++). In exheritance a non-selected feature from subclasses will cancel the entire feature. The other mechanisms do not correspond to this criteria.

Adding an abstraction layer into a class hierarchy is not possible with inheritance unless the whole hierarchy is redesigned. Exheritance helped by inheritance can achieve this goal.

In conclusion, looking at these class reuse mechanisms through the proposed criteria we can motivate the analysis and implementation of a reverse inheritance semantics.

Criteria \ Reuse Mechanisms	Inheritance	Exheritance	Mixins	Mixin Layers	Traits	AOP	Classboxes	Expanders
to create new abstract types	no	yes	no	no	no	no	no	no
to factor features from classes	no	yes	no	no	no	no	no	no
to combine the implementations of features	yes, using super like mechanisms	yes, using inheritance	yes	yes	yes	yes	yes	yes
to redefine the implementation of features	yes	yes	no	no	no	no	yes	yes
to adapt the implementation of a feature	yes	yes	no	no	no	yes	yes	yes
to cancel the implementation of a feature	no	yes	no	no	no	no	no	no
to add an abstraction layer into a class hierarchy	no	yes	no	no	no	no	no	no

Table 2.2: Comparison of Class Reuse Mechanisms

Chapter 3

Towards Exheritance: Main Issues

3.1 Generalities About Exheritance

3.1.1 Main Approaches of Reverse Inheritance

The idea of upward inheritance was born in the database world from the concept of database schema generalization [SN88]. A type corresponding to a database schema may be a generalization of several specialized ones. It is also the case of generalization in a global multi database view which provides a homogeneous interface to a set of heterogeneous databases. The basic idea of reverse inheritance class relation is the **generalization abstraction** [SS77], which enables a set of individual objects to be thought generically as a single named object. It is considered to be the most important mechanism for conceptualizing the real world. Generalization helps the goal of uniform treatment for objects in models of the real world.

From the development point of view of a software system, direct inheritance is a top-down approach of construction, while reverse inheritance offers the possibility of constructing software in a bottom-up manner. We adhere to the idea that it is more natural to first create the subclasses, then to observe and analyze commonalities, and after that to define the super classes [Ped89, Sak02]. The autonomous design of class hierarchies or database schema will give rise to **inhomogeneities**. Their reusability depends strongly on their capabilities of adapting their local interface to a common global interface.

3.1.2 Definition

The **reverse inheritance** class relationship is also known as **exheritance** [Sak02], **adoption** [LHQ94], **generalization** [Ped89, OMG04] or **upward inheritance** [SN88]. The source class of reverse inheritance is known as **generalizing class** [Sak02] or as **foster class** [LHQ94]. In the state of the art there are several approaches dealing with reverse inheritance issues in domains like object-oriented programming and design, databases, artificial intelligence.

We start from the definition of reverse inheritance given by Pedersen [Ped89, Sak02] which states that a class G can be defined as a generalization of A_1, A_2, ..., A_n previously defined classes. If the value of n is 1 then we discuss about **single generalization**, otherwise about **multiple generalization**. Informally it can be defined as another model of inheritance where the subclass exists and the superclass is constructed afterwards.

3.1.3 Intension and Extension of a Class

In [Ped89] is presented a simplification of the object concept. The **intension** of a class is the set of properties through which it is defined. An example is given in this sense. The "mammal" concept is analyzed. The intension of this concept refers to real-world properties like: these animals have mammae which secret milk as nourishment for their young. By **extension** of a class we mean all

the phenomena[1] that include those properties. Back to the analyzed example it can be considered that the neighbor's dog belong to the extension of the mammal concept.

Specialization can be defined in terms of intension and extension of a concept. A concept $C_{special}$ is a specialization of a concept C, if all phenomena of $C_{special}^{extension}$ belong to $C^{extension}$ [Ped89]. Concept worker is a single specialization of concept employee, since all workers have all properties of employees and eventually some extra. A worker can take the place of an employee but not necessarily the other way around. Formally this can be expressed like: a concept $C_{special}$ is a single specialization of a concept C iff $x \in C_{special}^{extension} \Rightarrow x \in C^{extension}$. The notion of **multiple specialization** can be defined in the same way: a concept is a multiple specialization of a set of other concepts if it is a single specialization of each concept in the set [Ped89]. Concept calculator-watch is a specialization of both concepts calculator and watch. Calculator-watch fulfils the properties of calculator and watch. Formally, a concept $C_{special}$ is a multiple specialization of $C_1, ..., C_n$ iff $x \in C_{special}^{extension} \Rightarrow \forall i \in 1..n : x \in C_i^{extension}$[Ped89].

Generalization can be defined also in terms of intension and extension of a concept [Ped89]: a concept $C_{general}$ is a single generalization of a concept C if all members of $C^{extension}$are members also in $C_{general}^{extension}$. This means that all phenomena belonging to $C^{extension}$ will belong also to $C_{general}^{extension}$. Concept employee is a generalization of concept worker since every worker is an employee. Formally $C_{general}$ is a generalization of concept C iff $x \in C^{extension} \Rightarrow x \in C_{general}^{extension}$. As in the case of specialization there is **multiple generalization**. A concept is a multiple generalization of a set of other concepts if it is a single generalization of every concept in the set. For example the concept of employee is a generalization of worker, manager, security guard, secretary, because all are employees. In formal notation $C_{general}$ is a generalization of $C_1, ..., C_n$ iff $\forall i \in 1..n, x \in C_i^{extension} \Rightarrow x \in C_{general}^{extension}$.

3.1.4 Semantical Elements of Reverse Inheritance

The idea that reverse inheritance should have an appropriate symmetrical semantics in order to produce the same class hierarchy structure, having the behavior as if it was defined by direct inheritance, is proposed in [Sak02]. So, the foster class will include all the features (attributes and methods) that are **common** to the exherited classes. Also it can be specified by the programmer which features should be excluded from exheritance.

In [LHQ94] two rules are set for defining the semantics of reverse inheritance class relationship: one sets the type conformance between subclasses and superclasses and the other defines the class dependency which is oriented from superclass to subclass. The subclasses will conform to the types of the newly designed superclass and the newly created superclass depends upon the subclasses. Of course, the two rules are not sufficient and a set of restrictions are also defined to complete the definition. These will be presented further.

As we already know from ordinary inheritance, subclasses depend and conform to their superclasses. So both **dependency** and **type conformance** have the same direction from subclass to superclass. In the analysis made in [LHQ94], the reverse inheritance concept changes their directions: the type conformance remains in the same direction, but the dependency is now oriented from superclass to subclass.

As presented in the Unified Modeling Language description document [OMG04], which is considered the standard modeling language for the object-oriented development process, the generalization relationship can be applied to several model elements like classes, associations, stereotypes, actors. By definition, the role of generalization is to relate a more **general** element and a more **specific** element, so the instance of the specific classifier is also an instance of the general classifier.

In the work of [Pon02] an analysis is made on relation of generalization with other UML elements and two aspects are emphasized: an incremental one and an overriding one. The former goes along with classes and the latter fits to associations, stereotypes, signals, use cases, actors. The incremental aspect refers to subclasses which have a richer set of messages in their interface than their parents. Overriding happens when two methods are created, one in the superclass and

[1] By phenomena, in this context we refer to objects.

one in the subclass, denoting the same message, but having different parameter and return types or different behavior.

We can conclude that all semantical definitions include either the idea of intension intersection or extension reunion of the generalized concepts.

3.1.5 Reuse of Object vs. Reuse of Class

In this subsection we will refer to different reuse cases for objects and classes.

Class hierarchies and database schema were build independently and so they achieved a degree of inhomogeneity. The challenge is in which manner these classes or database schema can be reused. In order to achieve the goal of reusability, **a single set of messages** should be available.

Some simple solutions seem to solve the uniformity problem. One is to make local changes in the classes [SN88]. The disastrous consequence is a dramatic chain of modifications in the clients, which will trigger a new cycle of software development [Mey97].

Another solution is to create new views for these objects. This will determine an explosion of variants of the original ones which can differ slightly [SN88, Mey97]. So, we created a huge configuration management problem.

Another possible solution is to create a union of the generalized class objects [SN88]. The common messages of the generalized classes can be received uniformly by all subclass instances. This solution involves inconsistencies when a foreign message is sent to an object which cannot execute it, but it works immediately only for the features having the same name.

In [SN88] are discussed the several semantical relationships between classes: identity relationship, role relationship, history relationship, counterpart relationship, category relationship. There are investigated situations when two objects are equivalent: one possibility is for their classes to model the same real-world object or their classes can model real world-objects which have some common properties. The **identity relationship** between two classes stands when the real-world objects modelled by those two classes are identical at all points of time. The **role relationship** occurs when two objects may model the same real world object in different situations or context. For example a person could be at the same time university employee and company employee. This person has two different roles which could be modelled by different classes. These classes are an example of role related classes. The **history relationship** between classes occurs when these classes model a real-world object at two different real-world times. The **counterpart relationship** hold between two non-equivalent objects which represent two different real-world objects. It is necessary for them to have some common properties and to represent alternate situations in the real world. For example two classes modeling air connections and train connections are counterpart related because both have properties like departure city, destination city and fare. The **category relationship** holds between objects which share some common properties. Two classes modeling coal plants and oil plants are category related because both share plant common related properties.

3.1.6 Explicit vs. Implicit Declaration of Common Features

We think that the specification of the excepted features in the definition of the reverse inheritance relationship between two classes has a drawback, affecting clarity. If one wants to develop further a reverse inheritance based class hierarchy, he has to know the list of all the common features from exherited classes. Instead it would be better to have a list of them explicitly declared in the foster class. The explicit list of features in the foster class will be of much more help to the programmer, for example in the definition of a subclass derived directly from the foster class. One more reason to sustain the affirmation is related to the possible adaptations to be declared around common methods having incompatible signatures. So the syntax will be easier. The exheritance concept comports two essential aspects: interface[2] exheritance and implementation exheritance. Each aspect will be detailed in the next chapters.

[2]By interface we denote the set of public features in a class. It is different from the concept of Java interface, which technically is a pure abstract class [AG00].

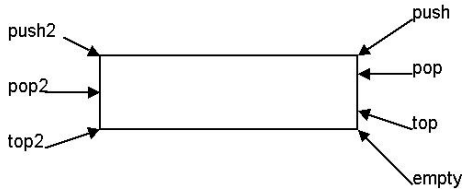


Figure 3.1: Dequeue Example

3.1.7 Allowing Empty Class

In [Sak02] the case of no exherited features is discussed. It can be useful in languages where there is no default superclass for all the user defined classes. This practice fits better to dynamically typed languages. It involves mechanisms of runtime type checking and casting operators. For instance, in Java there is a class *Object* which is the **absolute superclass of all classes**. In the definition of a new class, the relation with class *Object* is not explicit, it is implied automatically by the semantics of the language. In Java we find two main types of classifiers[3]: classes and interfaces. Analyzing the problem, we draw the conclusion that Java presents an asymmetrical semantics regarding interfaces: there is no primordial superinterface. The problem of lacking a super interface in Java could be solved with the help of reverse inheritance. It is the same for Eiffel language, having an absolute super class named "ANY"[4].

C++ is different from this point of view, it has no default superclass. In practice, in some top-down developed class libraries there is defined a default class as root class of all the classes in the library. This solution is non uniform, in such cases the name of the superclass differs from hierarchy to hierarchy. Also different default behaviors are provided to such classes. We draw the conclusion that such solutions are highly parochial. Another possibility is to define such a class using the concept of reverse inheritance, without touching the target classes.

3.1.8 Source Code Availability

One important issue about reverse inheritance is the source code of generalized classes [Sak02]. There are several situations that have to be discussed. The most favorable situation is when source code is available. This gives many choices in the implementation of a such a class relationship. We can imagine an implementation based on the modification of the original source code and on the equivalent source code generation, compilable by the Java/Eiffel/C++ compiler. Another situation is when source code is available but it is read-only. It cannot be modified because of many reasons: copyright policy, increased effort for maintenance. In this case it can be generated equivalent source code using decoupling techniques to protect the original sources. The most problematic case is the one where no sources are available, just the interface and the binaries of classes. In such case byte code modification techniques should be applied.

3.1.9 Single/Multiple Exheritance

Single exheritance is the most simple case of exheritance. It involves only one target class to be exherited. Both interface and implementation can be exherited. In [Ped89] an example of a double ended queue is given to emphasize the semantics of single generalization. It is started from a class named *Dequeue* having a set of methods which operates at one end of the dequeue: *pop*, *push*, *top*; methods for operating at the other end of the dequeue: *pop2*, *push2*, and a separate method *empty*.

From the dequeue a stack is created. Class *Stack* is declared as generalizing *Dequeue* and excluding operations *push2*, *pop2*, *top2* from its interface. From the point of view of interface

[3]We refer to the classifier defined in the sense of UML.

[4]We mention that special class "ANY" has internally private superclasses.

exheritance there are no conflicts. Name conflicts obviously cannot occur, because initially the generalizing class has no features. Some name changes could be necessary to give a more suggestive meaning to methods. For instance if we generalize a class *Collection* from class *Stack* instead of exheriting the *push* method with the original name we should better exherit it with another name, like *add* for example.

3.2 Interface Exheritance

In this section we will discuss about the interface content of a generalizing class. Method implementations defined in subclasses are not taken into account from this point of view.

3.2.1 Concrete vs. Abstract Generalizing Classes

In [Ped89] is emphasized that this aspect of exheritance is the most simple. As mentioned in [Sak02], the integration of interface exheritance in Java can be done with minimum of effort because of the notion of "interface" they introduced in the language. A Java interface consists in a set of abstract methods [AG00]. It can be considered as a pure abstract class. An abstract class in Java may contain abstract methods having no implementation, just signature and also concrete methods with implementation. We note also that interfaces can be created by specialization of several multiple interfaces, they can be implemented by several subclasses and their methods are all public. It is suggested that interfaces could be defined by generalization of classes and other interfaces.

Not all languages possess such an interface concept like Java does, so we have to use the class concept as generalization classifier. For those languages is proposed the idea of generalization into fully abstract classes (e.g. Eiffel, C++) [Sak02].

3.2.2 The Influence of Modifiers on Exherited Features

When exheriting method interfaces one question arises: should we consider just the public ones or should we consider all of them, including non-public ?

In order to answer this question we have to analyze what happens if we exherit from non-public methods. The positive reasons are: they can be better reused from the level of the generalized class and they can be inherited later. Negative reasons are class encapsulation violation which implies visibility modification: clients may be affected, exposed methods can be overridden. So, potential problems may be introduced at the implementation level too[5]. In [Sak02] it is advised not to distinguish between public and non-public in the exheritance of method interfaces.

It is mentioned in [Sak02] that attribute exheritance does not involve big problems. Though, some type and visibility problems may occur. Here we can take two cases: public and non-public attributes. Although in the literature the use of public attributes is not encouraged, still some programming languages allow their use like C++, Java. In Eiffel attributes can be exported but they cannot be modified, only read, because the access to an attribute involves an execution of a query which provides the desired result.

We will analyze from the conceptual point of view each type of modifier encountered in common object languages. The problem of visibility consists in finding the appropriate modifiers for the exherited features in the superclass. The main idea is that we want to preserve or to affect as little as possible the feature's visibility.

The most simple case is when we exherit **public** features because they will be treated as public in the superclass. Any access to the exherited features is freely granted.

If we deal with **package** access type[6] modifiers in the subclasses then it means that features have to be available in their package. We have to discuss several cases:

[5] Implementations issues are not the subject of this analysis.

[6] This is known also as default access type and it is specific to Java language. It has no dedicated keyword, all features having no keyword have by default package access type. C++ and Eiffel do not consider visibility at package, cluster or subsystem level.

Example 23 Examples in Java

```
// Java example
abstract class StaffMember {
  abstract public void print();
}
class SecurityAgent {
  public void print(){ System.out.println("SecurityAgent"); }
 }
class TeachingAssistant extends StaffMember {
  public void print() { System.out.println("TeachingAssistant"); }
 }
class Professor extends StaffMember {
   public void print() { System.out.println("Professor"); }
 }
class Employee exherits StaffMember, SecurityAgent {
   public void print() { System.out.println("Employee"); }
}
```

i) All classes are in the same package: the superclass and all exherited subclasses. In this case the modifier of the exherited features in superclass can remain the **package** access.

ii) The superclass is in a different package than all the subclasses. In other words we can say that the features migrates from one package to another. This case requires to change the visibility of an inherited feature, to be accessible from the origin package. The possibilities are **protected** if accesses to that feature are made from inside the class or **public** if other clients need access. Unfortunately, the two choices violate encapsulation.

iii) The superclass is in the same package as some of the subclasses, but other subclasses are in different packages. This situation is a mix of the cases described above. The visibility modifier in this case should be computed for each feature in subclasses and the **most visible** modifier should be used. The price paid for homogeneity of reverse inheritance is breaking encapsulation.

If exherited features from subclasses are **protected** then it means that they are available to all their subclasses. If the feature declared in the superclass of generalization is **protected**, then no possible clients are affected.

If exherited features in subclasses are **private** then it means that they are accessible only in their original classes. As a consequence their modifier in the superclass of generalization should be **protected**.

3.2.3 Status of Original Methods: Abstract/Concrete

When **abstract methods** are exherited from the generalized classes that means that they will have to be declared abstract also in the foster class. The same should happen if we exherit both abstract methods and concrete methods. So the foster class becomes abstract automatically. This kind of behavior seems to be fair from reverse inheritance conceptual point of view.

If we decide that we have a sufficiently general implementation we can put it in the superclass without affecting the potential abstract subclasses. These ideas are exemplified in two examples: one Java, the other C++ (see examples 23 and 24).

In the two examples class *StaffMember* is abstract: in Java because of the abstract modifier, moreover it has an abstract method named *print()*; in C++ because of the virtual and equals zero *print()=0* method declared. Class *Employee* generalizes two classes in the parallel hierarchies one abstract and one concrete: *StaffMember* and *SecurityAgent*. The *print* method which equips all the classes, could be either abstract or concrete. If we possess a general enough implementation that will fit to all subclasses from now on, then we can put it in the *Employee* class. Conversely, we declare the print method as abstract, obviously leaving it without implementation. In either cases

Example 24 Examples in C++

```
// C++ example
class StaffMember {
  public: virtual void print() = 0;
 };
class SecurityAgent {
  public: void print() { printf("SecurityAgent"); }
 };
class TeachingAssistant {
  public: void print() { printf("TeachingAssistant"); }
 };
class Professor {
  public: void print() { printf("Professor"); }
 };
class Employee exherits StaffMember, SecurityAgent {
  public: virtual void print() { printf("Employee\n"); }
 };
```

subclasses like *TeachingAssistant* or *Professor* can override the print method with appropriate behavior. If we create the same class hierarchy but using inheritance this time, then we will find the same semantical behavior regarding the status of the print method.

In Java we can use also the **final** modifier for the exherited method's status if we want to be more imperative about the implementation put in the superclass. We remind that with ordinary inheritance a method declared as **final** cannot be overridden in the subclasses, otherwise a compiling error is generated.

When **concrete methods** are exherited there is the possibility to exherit just the interface or together interface and implementation. This choice should be available to the programmer who uses the reverse inheritance class relation [Sak02]. If it is chosen not to exherit implementation, then in the foster class just the corresponding abstract method can be specified. The aspects dealing with implementation exheritance will be discussed later, in a specially dedicated section.

3.2.4 Type Conformance Between Superclass/Subclass

Related to interface exheritance issue, in [Ped89] it is demonstrated using an experimental language that from the type conformance point of view, there are **no conflicts** introduced in a class hierarchy having subclasses/superclasses introduced by inheritance/reverse inheritance. The main idea of the demonstration is to prove using formalisms that the feature set of the generalizing class contains at most the intersection of the subclass features sets.

Before proof, generally speaking, some notations are necessary:

$$C^{methods} = \{m_1, \ldots, m_n\}$$

denotes the set of methods of class C.

Class A is defined as generalization of classes $B_1, B_2, \ldots, B_k$ removing methods $r_1, \ldots, r_n$. To prove that $B_i (i \in 1 \ldots k)$ conforms to A, means that class A method set is a subset of those of any instance of class $B_i (i \in 1 \ldots k)$. We use the following formalism:

$$A^{methods} = \bigcap_{i=1}^{k} B_i^{methods} \setminus \{r_1, \ldots, r_n\}$$

So it is demonstrated that A is a superclass of $B_i (i \in 1 \ldots k)$, so the conformance rule is valid.

In the [LHQ94] definition of semantics a type conformance rule is set. The type of subclasses have to conform to the type of superclass. From their point of view the superclass type is a

generalization of the subclasses types. It can imply type intersection or type union, depending on the type definition. In section 3.1.3 we discussed about the intension and the extension of an object. Referring to these two conceptual aspects of an object they consider that if a type is a set of features than the type of the superclass should be their **intersection**. If the type is considered as a set of objects, then the superclass type of the generalizing class will be a least the **union** of the subclass types.

3.2.5 Common Features and Assertions

In this paragraph we discuss ideas from state of the art regarding how common features are defined. An attempt in this direction is made in [LHQ94] and two restrictions are set forth: i) common features are those who have same name, ii) it is possible to define a common signature to which all signatures from the subclasses conform.

We present also some ideas of how preconditions, postconditions and invariants are affected by reverse inheritance. We remind that predicates are the main concepts around which the **Design by Contract** technique was built [Mey97]. The purpose was to offer to the programmer tools to express and validate correctness of a program. The relation between a class and its clients may be viewed as a formal agreement expressing rights and obligations for each of the parties.

A **precondition** states all the predicates that have to check when a routine is called. **Postconditions** are predicates which verify the properties that must hold when a routine returns [Mey97].

According to [LHQ94] the assertions rules defined in [Mey97] and [Mey02] should be reversed. The precondition for a feature in the foster class should imply all the preconditions in the generalized subclasses. The postcondition of a feature in the superclass should be no stronger than any of the corespondent preconditions from exherited classes.

Possibilities of building the precondition and postcondition for the features in the superclass are also mentioned in [LHQ94]. For example the precondition can be obtained by applying the *AND* logical operator against all the preconditions from exherited subclasses. The result will be definitely a stronger precondition. The postcondition can be obtained in the same manner using *OR* logical operator against all the corresponding postconditions in subclasses. This approach assumes that all variables involved in superclass predicates are defined in each subclass. Otherwise, for each subclass missing variable we could choose not to evaluate the respective term of the expression. In other words we could propose to evaluate the predicates only for the classes which have all the variables defined.

As a conclusion regarding assertions in [LHQ94] in the definition of common features, it has to be mentioned that they depend on the possibility to define a precondition other than False, which is no weaker than the precondition of the feature in each class.

3.2.6 Possible Conflicts

Name Conflicts

In [SN88, Ped89, LHQ94, Sak02] name conflicts are discussed. They occur when two methods have the same semantics but have different names. This conflict is named **lost friends** in [Sak02]. This kind of conflict cannot be detected automatically, so this must be set by the programmer. A supporting syntax is suggested to be used in order to indicate which methods should be exherited and which method name to be kept.

We think that it has to be considered not just the name of the method but its entire signature. This involves methods name, return type, parameter name, number and type, whether they are implicit or not, invariants, preconditions, postconditions.

In [SN88] the problem of name conflicts is discussed in object-oriented database schema. The conflict takes place between the local and global interface of an object. Local interfaces refer to the original interface of the object, which the object was designed with, while global interfaces

Example 25 Name Conflicts (1)

```
class OIL_PLANT
 attributes:
  PlantName
  Produced: MWh  {energy produced}
  OilFired: BARRELofOIL
 methods:
  FireOn
  PowerOff
  FillOil
class COAL_PLANT
 attributes:
  PlantName
  Produced: MWh  {energy produced}
  Consumed: TONofCOAL
 methods:
  Start
  PowerOff
  PutCoal
```

Example 26 Name Conflicts (2)

```
class POWER_PLANT
  metaclass: CATEGORY_GENERALIZATION_CLASSES
  generalization_of: OIL_PLANT, COAL_PLANT
  attributes:
    consumed: MJOULE
    corresponding:
      OIL_PLANT.OilFired
      COAL_PLANT.Consumed
  methods:
    PowerOn
    corresponding:
      OIL_PLANT.FireOn
      COAL_Plant.Start
```

denote the common set of messages. The same semantical messages have different names in the two interfaces.

The two classes presented in the example taken from [SN88] model two kinds of plants: oil and coal. Each of them has a name property, produced energy property, starting method, stopping method and charging method. All these features are common to the two classes, some of them have the same name, like *PlantName* or *PowerOff*, and others have different names, although they have the same semantics, like *FireOn* and *Start*. In this case name conflict situation appears.

So, they propose the solution of **object coloring**. It deals with the separation of the **local and global behavior**. To objects is attached a color attribute which will determine which local or global behavior should be followed at runtime. The switch between these two states is achieved by adding an "*as*" message to the object model. This modification is made by affecting the most general class in each subsystem.

In fact the mechanism proposed resembles very much the polymorphism mechanism. The substitution principle and the dynamic linking can be achieved by coloring an object. One can say that the coloring practice is even closer to the type casting facility offered in most object-oriented programming languages.

Example 27 Name Conflicts (3)

```
deferred foster class SHAPE
  adopt
    BOX
      rename
        boundary as perimeter;
    CIRCLE
      rename
        circumference as perimeter;
feature
    perimeter: REAL;
end
```

At runtime a *cp* named object, instance of $COAL\_PLANT$ receiving an "*as*" message like *cp* as $POWER\_PLANT$ will switch to the global behavior of $POWER\_PLANT$. Now if this instance will receive the *PowerOn* message, the instance will choose automatically the *Start* method to be executed.

We think that the choice of referencing different names with a unique global name could solve this kind of conflicts. This idea should be adapted in order to match attribute names and also method signatures.

In [LHQ94] the renaming facility from Eiffel [Mey02] is used in solving name clashes (see example 27).

Semantically equivalent features developed in different classes by different programmers will have different names. In the example of example 27, boundary feature from *BOX* and circumference feature from *CIRCLE* have the same semantical value, and they are mapped to a unique name.

Value Conflicts

Value conflicts are encountered when features with different semantics have identical names. They are referred in [Sak02] as **false friends** conflicts. Implicitly it is suggested that features should not be exherited and such situations should be specified by the programmer. They cannot be automatically detected and a syntax support for conflict declaration is needed. It is suggested not to exherit such features because they are not the same.

There are cases when name conflicts can be detected automatically. Two classes having a same ancestor can have renamed methods using renaming techniques like those in Eiffel. Both kind of conflicts can happen [Sak02]. In the case of lost friends conflict, the features seem to have the same seed but have different names because of renaming. A solution at compiler level is given: they should be organized as the same feature by the compiler.

The Solution of Renaming in the Case of Name and Value Conflicts Renaming is considered to be a solution in the case name and value conflicts. Also there are some negative effects: it influences clarity in the declaration of features, through the inheritance path it can have several names. On the other hand renaming is considered to be a good way to change the linguistic meaning from a too restricted to a more general one. So, the names will be more suggestive in the generalized class.

Scale Conflicts

Another type of conflicts could be considered the scale conflicts. They can appear when numerical values are involved. This happens more in object-oriented systems. The problem is that features representing values do not use the same scale.

Example 28 Scale Conflicts

```
class MJOULE
  metaclass: DATA_TYPE_CONVERSION_CLASSES
  generalization_of: BARRELofOIL, TONofCOAL;
  transformation_methods:
    FromBarrelOfOil {convert from Barrels of Oil to MJoule}
    FromToneOfCoal {convert from Tones of Coal to MJoule}
    ToBarrelOfOil
    ToTonOfCoal
```

Example 29 Parameter Order Conflicts

```
deferred foster class SHAPE
adopt
   BOX
    rename
     zoom(center: POINT,factor: REAL) as scale(factor: REAL, center: POINT);
   CIRCLE
feature
  scale(factor: REAL,center: POINT) is deferred end
end
```

An example presented by us in section 3.2.6, about how this kind of conflict can be eliminated, is discussed in [SN88]. We remind the reader that the common attribute named *OilFired* is expressed in barrels of oil while *Consumed* attribute is expressed in tons of coal. In the generalizing class the desired member should have the same name and the same scale, meaning *MegaJoule*.

In their work [SN88], in order to solve the problem of scale differences, they proposed the concept of **object transformation**. This concept is implemented with the help of conversion classes. These classes contain a set of methods representing the necessary conversion protocol between several scales. The main idea of the solution is to switch between the **local and global representation** of an object. The local representation uses one scale, while the global one uses another. A conversion class that solves the presented problem can be like the one presented in example 28.

In example 28 class *MJOULE* encapsulates all the transformation routines between barrels, tons and MJoules. These transformation methods represent the adaptation behavior from a scale to another. To be more explicit, they adapt values. This technique, as it is presented in the example, can be applied only to attribute instances which are values or maybe to methods which return values. We think that with the help of some modifications this could be also applied to methods not just attributes. In our opinion the place of such an adaptation behavior code should be in the generalizing class, close to the adapted feature, because of clarity reasons.

Parameter Conflicts

Parameter Order In [LHQ94] another kind of conflict is emphasized briefly. In exherited methods the order of parameter may vary. This conflict can be solved by a **mechanism for binding arguments dynamically**. For Eiffel, an appropriate syntax is proposed:

Example 29 specifies the mapping between the method *zoom(center:POINT,factor:REAL)* and method *scale(factor:REAL,center:POINT)*. We notice that besides renaming, parameter order is set also. In order to perform such parameter reorganization, the number of parameters in all the exherited methods and in the superclass must be all equal.

Example 30 Parameter Number Conflict

```
class B
{
 void foo(int x,int y,int z = 0,long g = 10){}
}
class C
{
 void foo(int x, int y, double t = -1, float pi = 3.14){}
}
class A exherits B, C
{
 virtual void foo(int x,y);
}
 B b;
 C c;
 A * pab=&b;
 A * pac=&c;
 pab->foo(1,2); // equivalent with pab->foo(1,2,0);
 pac->foo(2,3); // equivalent with pbc->foo(2,3,-1,3.14);
...
```

Parameter Number The number of parameters is an important criteria in order to match methods when exheriting them. Of course, the number of parameters is desired to be the same in superclass and subclass methods. In languages like C++ which permits the declaration of methods having implicit parameters it is possible to declare a general signature in the superclass and a conforming signature in the subclass, in addition it can have as many implicit parameters as needed.

As a remark, the position of the default parameters is always after the non-implicit ones. Also, the order of the default parameter is important, because you cannot use implicit *pi* parameter unless you specify firstly an actual parameter for *t* in method *foo* of class *C*.

Parameter Type The type of parameters can cause problems in exherited features. This kind of conflict has similarities with the one discussed in section 3.2.6. The problem is how can we unify two parameters having different types. First of all we start our parameter type analysis with a small discussion about the **parameter transmission mechanisms** encountered in Java, C++ and Eiffel object-oriented programming languages. So, in Java we have value transmission mechanism for primitives and object references. The value of the actual parameter is copied into the formal parameter. Changes on primitive typed formal parameters will not affect the actual parameters. When dealing with object references, they cannot be affected, but the referred objects, obviously, can be modified.

In C++ we have types like primitives, objects, pointers and references. The transmission of primitives is the same like in Java. What is interesting in C++ is the implicit call of the copy constructor[7] when transferring object value parameters [Str97]. The situation in Eiffel [Mey02] regarding parameter transmission is the same like in Java. There are transferred primitive values and reference values.

Another issue which have to be discussed before trying to get to parameter unification problem of reverse inheritance, is the **variance** in the analyzed programming languages. There are three possibilities of parameter variance: covariant, nonvariant and contravariant. A programming language is covariant if in the redefinition of a method in the subclass, the types of parameters vary along with the type of the class in which the method is declared in.

[7] The copy constructor can be implicit and then byte copy of the object is performed, or the programmer can override this default behavior by providing an explicit copy constructor.

Example 31 Parameter Type Conflicts (1)

```
class A {}
class B extends A {}
class Parent
{
 void foo(A argument){}
}
class Child extends Parent
{
 void foo(B argument){}
}
```

Example 32 Parameter Type Conflicts (2)

```
class Point2D
{
 void setX(double x){}
}
class Point exherits Point2D
{
 void setX(int x){}
}
 Point p = new Point2D();
 p.setX(3);
```

In example 31 we can discuss covariance issues. Method *foo* in class *Child* is a redefinition of method *foo* in class *Parent* if: i) the language is covariant and B is a subtype of A; ii) the language is contravariant and B is a supertype of A; iii) the language is nonvariant and A and B represent the same class.

One situation is when we deal with primitives types in object-oriented languages. If we deal with compatible types we could perform **type casting** implicitly like in the example 32.

In our experiment (see example 32) we have two methods *setX* with two different argument types: *int* in the superclass and *double* in the subclass. For instance if we replace in C++ or Java a parameter's *int* type with a *long* type, arithmetical computations are not affected. But if bit level operations are executed, the result will not be the same. The conclusion is that primitive type substitution implies potential risk to the affected code, a runtime casting mechanism, which can sometimes affect values (e.g. precision loss) and knowing the compatibility rules between the primitives. The most radical solution that can be applied is not to allow the feature exheritance, unless parameters have the same type.

3.3 Implementation Exheritance

3.3.1 Impact of Polymorphism in the Generalization Source Class

In the state of the art are studied two different situations: when exherited methods in the superclass are virtual or non-virtual. When no virtual methods are present in the foster class, in [Ped89] two unsatisfactory solutions are proposed.

Principal Subclass Implementation Pedersen [Ped89] proposes that one of the generalized classes should be chosen as main subclass. It is also motivated that the choice should be made by the programmer because he knows better the implementations of subclasses and because in some

Example 33 Impact of Polymorphism

```
class Rectangle
{
 double a,b;
 public double area(){return a * b;}
}
class Rhombus
{
 double a,theta;
 public double area(){return a * a * sin(theta);}
}
class Parallelogram exherits Rectangle, Rhombus
{
 double a,b,theta;
 public double area(){return a * b * sin(theta);}
 // the most general method for area computation
}
```

languages interface inheritance means also implementation inheritance.

New Implementation The other proposed possibility to deal with exherited implementations is to provide a new implementation in the foster class. Both solutions proposed by Pedersen [Ped89] are criticized in [Sak02] because the semantics of generalization is broken. Because methods are non-virtual or non-polymorphic, the specialized behavior of all subclasses become unusable. In some special cases such a class construction can be useful. One of them is presented in the next paragraph.

Non-Virtual Methods Can Be Useful Sometimes If exherited methods in superclasses are not virtual there is no problem from technical point of view (one can build such a class hierarchy without compiler errors), but it does not express the desired semantics of generalization. Exceptions may occur in the situations where a more general implementation is available and the specialized one from the subclasses can be overridden without affecting the consistency of the class hierarchy, like in example 33.

Generally speaking a virtual method is a method that has a polymorphic behavior. Depending on the type of the object at runtime, a polymorphic method call may exhibit different behaviors. Here we refer only to the dynamic linking component of polymorphism. No matter if the original method is abstract or concrete, generally, the exherited method specified in the foster class should be made virtual [Sak02]. It is known that in any language virtual methods can be overridden in the subclasses.

Empty Method The first solution discussed in [Ped89] is to equip the exherited method in the foster class with empty body in the case of no common behavior. This method could be easily overridden by specialized subclasses. In languages like Java we could define it concrete with empty body, triggering an exception in case someone calls this method, or abstract with no body, meaning that all the concrete subclasses are obliged to implement it. In Eiffel we can declare it in the superclass as deferred method. In C++ we have two possibilities by declaring it abstract or as pure virtual. A pure virtual method is a method declared with the "virtual modifier" and having also "=0" suffix meaning that it is a pure method.

Main Class Behavior The second proposition and the default one made in [Ped89] is to exherit implementation from the selected main subclass, when all implementations exhibit the same

Example 34 Selective Method Exheritance

```
class Iterator1
{
 int value=0;
 void increment()
  assume value >= 0
  {value = value + 1;}
  guarantee value > 0
}
class Iterator2
{
 int value = 0;
 void increment()
  assume value >= 0
  {value = value + 2;}
  guarantee value > 0
}
```

behavior. In practice it seems a rare case that a set of classes are equipped with exactly the same body.

Some Common Behavior The idea promoted in this case by [Ped89] is a manual selection of the common behavior from subclasses and the definition of a new method in the generalizing class. This is motivated due to the fact that later on this implementation could be inherited in other subclasses of the generalizing class.

Selective Method Exheritance In the case of virtual methods in foster class, [Sak02] proposes an alternative solution to method body exheritance. The idea of exheriting methods from different subclasses is encouraged. The solution given seems more flexible and attractive. We think that some adaptations are still necessary to provide a more advanced degree of class reuse.

In practice it does not seem probable that one subclass can provide all the suitable behavior. Assertions, which deal with predicate abstraction, do not describe completely the behavior of a routine. So in general we can say that even if we find a method in a subclass which has assertions, which conforms against all the assertions from the other subclasses, is not sure that the behavior of that method will be fit to all the subclasses.

The two classes in example 34 have the same precondition and postcondition for method *increment()*, but different behaviors.

Adaptive Approach We think that in many situations an adaptive approach is more suitable. We propose to analyze a mechanism which allows to use the code from subclasses in a more flexible manner. Calls to the original version of the code are possible using the **inferior** calling mechanism, like in Beta programming language.

In the proposed example we can find two classes modeling Alcatel, Inc and Nokia, Inc phones. We need to treat these classes in a uniform manner, so we decided to create a foster class generalizing them. There can be noticed that the two classes have implemented ringing behavior[8]. It is quite natural that the new created class, baptized *GeneralizedPhone*, to exhibit the ringing behavior. So we exherit the signature of method *ring()* from both classes. We suppose that each subclass has specific ringing behavior. We need to reuse this behavior but in a different way. For example we would like to add actions before and after the each ringing operation. This can be

[8] For clarity and simplicity reasons we suppose that these methods have the same signature. In practice it seems quite improbable to be so. The problem of signature adaptation is treated in a different section.

Example 35 Adaptive Approach

```
class AlcatelPhone
{
 void ring()
 {
  // original Alcatel implementation
 }
}
class NokiaPhone
{
 void ring()
 {
  // original Nokia implementation
 }
}
class GeneralizedPhone exherits AlcatelPhone, NokiaPhone
{
 void pre_ring()
 {/* pre ring operations */}
 void post_ring()
 {/* post ring operations */}

 factored void ring()
 {
  pre_ring();
  inferior.ring();
  post_ring();
 }
}
```

Example 36 Type Invariant Assumptions

```
class A
{
 void f()
 {
  if (this instanceof A)
  {
   // do some actions
  }
  else
  {
   // do other actions
  }
 }
}
```

done with the help of a descendant access. An implementation of method *ring()* having a call to the *pre_ring()* operations, a call to the original specific method from the subclass and a call to the *post_ring()* operations can be imagined. The main drawback of this approach is that the semantics of the original classes may be changed by adding the new features.

3.3.2 Adding New Behavior

Referring to the example 35 we showed that new behavior added into the foster class can be useful. The *pre_ring()* and *post_ring()* methods added to the *GeneralizedPhone* foster class were used to create an enhanced ringing method. Other advantages of this feature will be presented in the section dealing with the mixing of inheritance with reverse inheritance. Still the classes are in danger of having the semantics changed. Such a capability would be a valid option in a potential reengineering tool based on the concept of reverse inheritance, but not in the programming language semantics.

3.3.3 Exheriting Dependencies Problem

In [Sak02] the problem of dependency exheritance is mentioned. It is stated that exherited methods dependencies have to be exherited also in order to be usable in the foster class. There are two possible solutions for this problem. One would be to exherit the dependencies as well. In practice this would mean to exherit almost all the features in a class. The other approach is to provide the missing dependencies in the class where the exherited methods were exported. Such an idea is used in the traits mechanism presented in subsection 2.4.

3.3.4 Type Invariant Assumptions

It is noted in [Sak02] that type invariant assumptions, executed in the context of an exherited method in the generalizing class, can be broken. This can happen not only in assertions but in the code of a regular method. Example 36 presents such a situation. The type assumptions, investigated in the condition of the *if* instruction, were designed to work in the context of the original class. The first branch of the *if* instruction is taken. If the code of method f would migrate in a potential generalizing class A, then the f method semantics would change. In this case the second branch of *if* would be taken.

From this example results that a use of the reflection mechanism in the implementation of a method would make exheritance impossible. Related to these problems the solution would be to

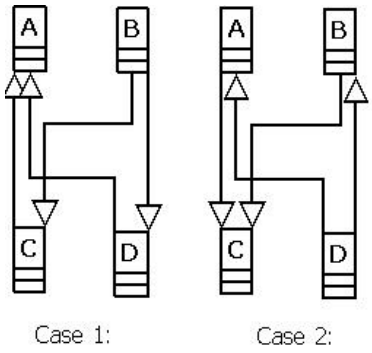


Figure 3.2: Fork-Join Inheritance Example

choose not to exherit the methods having such assumptions. In order to do so, these assumptions have to be detected first, so a dedicated technique would be required.

3.4 Mixing Inheritance With Exheritance

In this section we analyze some interesting combinations of inheritance and reverse inheritance in class hierarchies, looking carefully at the restrictions which have to be considered in order to avoid potential problems.

3.4.1 Fork-Join Inheritance

We begin with the analysis of a conceptual example from the state of the art. A case of fork-join like inheritance scheme is presented in [Sak02]. There are considered two cases: i) class A is defined then classes C and D are defined as inheriting from A. Class B is defined by generalization of classes C and D; ii) classes A and B are defined first, then class C is defined as a generalization of the two. Class D finally inherits multiply from A and B.

The arrows with up direction denotes specialization and conversely, the down directed arrows denote the generalization relationship.

In case 1 features from class A are propagated through inheritance into classes C and D. Then, by reverse inheritance from C and D, they are propagated into class B. Classes A and B may have some common features but at the same time each class may have specific features which were not propagated from one to another. So there is no subtype relation between A and B.

In case 2 the features of class C are propagated via multiple reverse inheritance to classes A and B, then they are inherited directly into class D by normal multiple inheritance. A subset[9] from class C message set are transferred to A and B and from there, the subset is included in the class D message set by the multiple inheritance. Again, there is no subtype relation between C and D.

In [Sak02] two particular sub-cases are analyzed: if there are no features excluded in the generalization and if there are no features added in inheritance. In the first sub-case features in class B will have all the common features of C and D obtained by inheritance from A, but it will also have specific exherited features. If both generalization and specialization have not excluded or added new features it is created an effect of **cloning** class A in class B of case 1 or class C in class D of case 2.

[9] It is a subset because not all the features are exherited, some may be excluded.

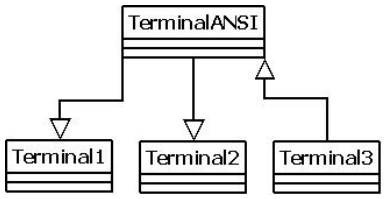


Figure 3.3: Terminal Example

3.4.2 Reusing Common Behavior

Another interesting idea is presented in [Ped89] about how common behavior factored using reverse inheritance, can be reused. For exemplification, a real world case is analyzed dealing with terminals. Given two classes *Terminal1* and *Terminal2*, which model two different terminals, the decision of creating a more general class of the two is taken, in order to group their commonalities for reusing purposes.

In the given context both terminals are ANSI terminals so a generalizing class *TerminalANSI* is created through generalization from *Terminal1* and *Terminal2*. Class *TerminalANSI* will contain all the common interface and implementation from the exherited classes and conform to the standard ANSI specification. Of course there could be features specific to subclasses which are excluded, being not subject for exheritance. A new class baptized *Terminal3*, modeling a new terminal is created as inheriting from foster class *TerminalANSI*. Thus, all the common interface and behavior will be inherited. Of course specific behavior can be added too.

The conclusion that can be drawn from this example is that using the two concepts together we can benefit from already defined classes. An alternative to this approach would be to define class *Terminal* from scratch and to rewrite or copy the standard behavior of ANSI terminal from one of the two classes. This is an error-prone practice and it should be a avoided. There are two reasons for this: it increases the entropy of the system and it is bad for the code management. We discuss what happens when the ANSI standard evolves in a new version is released and software components have to keep in step with the up to date modifications. We analyze the possibility of feature adding in reverse inheritance on this example without loosing generality. The new enhancements required by the standard will be the same for all three classes.

Specialization - The Classic Solution

One possibility is to create three subclasses for each terminal class using direct inheritance and to equip them with the enhanced behavior. The class hierarchy describing this solution is presented in figure 3.4. The three new added classes are: *EnhancedTerminal1*, *EnhancedTerminal2* and *EnhancedTerminal3*. So we created three more classes in the system which will contain the enhancements. There are two possibilities for the enhancements to be contained: they are either copied directly in all the subclasses or they are encapsulated in a new class, and a member of this type will be declared in all the three subclasses. The first possibility involves code duplication while the second one implies the creation of a new class and the usage of composition. The second possibility can be used only if the enhancements are the same for all the three terminals, still the code dealing with the manipulation of the component object is duplicated.

Feature Adding in Foster Class

Another possibility is to put directly the enhancements in the foster class *TerminalANSI* directly and then they will be automatically inherited in all the subclasses. In other words this means the creation of a foster class version. This solution addresses the "fragile base class problem" [MS98]. This type of problem appears when acceptable versions of the base classes are created which damage the extensions. In [MS98] this problem is viewed as a flexibility problem and restrictions

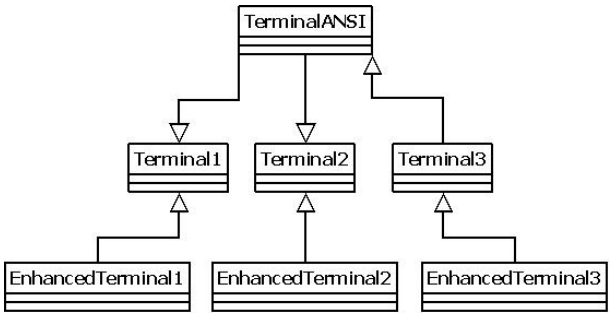


Figure 3.4: Terminal Enhancement (1)

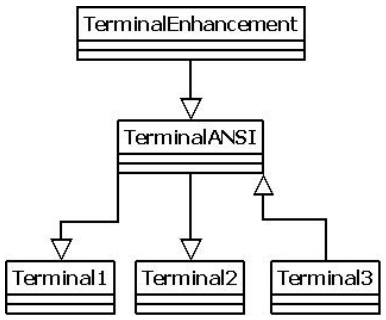


Figure 3.5: Terminal Enhancement (2)

can be set in order to discipline inheritance. On the other hand, a conceptual drawback is noticed because reverse inheritance should not imply feature inheritance but only feature exheritance.

Setting Superclass for Foster Class

An alternative to the previous solution is to create a new foster class having as target the *TerminalANSI* class and containing the enhancing behavior discussed earlier. So, we can leave the base class of the hierarchy untouched, this solution could be used in situations where source code is not available or no class maintenance responsibilities are accepted. The idea of adding features to a base class using reverse inheritance still suffers from semantical breaking and contamination with fragility. In figure 3.5 such a situation is depicted.

3.4.3 Dynamic Binding Problems

It is demonstrated in [Ped89] that multiple generalization conflicts are the same as for multiple inheritance, thus the solution to the second problem could be applied to the first. Conflict resolutions in multiple inheritance were analyzed in section 2.1.1. In [LHQ94] a similar problem of accessing a unified feature, which is multiply inherited, is discussed. In this case the multiple inheritance mechanism for Eiffel is used and combined with renaming.

In example 37 which was presented in a different manner in section 3.2.6 it is now created a class that derives from both *BOX* and *CIRCLE*. The problem is which implementation will be used when having a reference of class *SHAPE* and *perimeter* feature is called. This is an ambiguous situation which has the following alternatives: boundary from class *BOX* or circumference from class *CIRCLE*. A proposal is made in [LHQ94] to take as implementing feature the one present in the first declared subclass. In our case it is *boundary* from class *BOX*, because class *BOX* is the first in the inheriting list of class *CIRCULAR_BOX*. This approach is wrong because the selection

Example 37 Exheritance Dynamic Binding Problem

```
class BOX
feature
  draw is do  end
  height, width, area, boundary: REAL
end
class CIRCLE
feature
  draw is do  end
  radius, circumference: REAL
end
deferred foster class SHAPE
  adopt
    BOX
      rename
        boundary as perimeter
    CIRCLE
      rename
        circumference as perimeter
feature
    perimeter: REAL
end
class CIRCULAR_BOX inherits
 BOX, CIRCLE
end
s: SHAPE
cb: CIRCULAR_BOX
r: REAL
!!cb;
s := cb;
r := s.perimeter;
```

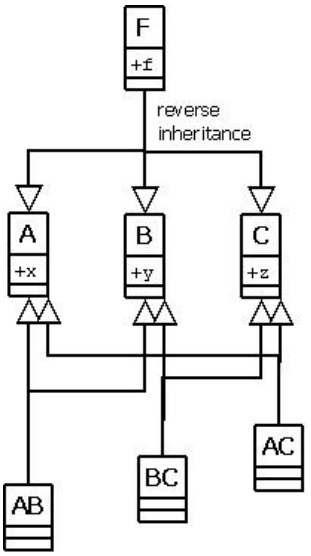


Figure 3.6: Exheritance Dynamic Binding Solution

method does not allow any configuration of selection in more complex class hierarchies. Such an example is given in figure 3.6 where classes A, B, C were created first and have features x, y, z. Then the generalizing class F is created having A, B, C as subclasses. Later on the AB, BC and AC classes are created having the first superclass in the list the class denoted by the first letter in their name. So for class AB the first superclass is A while B is the second. In this configuration if one wants to select feature x for AB, feature y for BC and feature z for AC it is not possible. The first two choices are valid, while the third one is not possible. The order selection solutions does not work in all cases.

The approach of Eiffel in the case of dynamic binding problem apparently could be used. This implies using the **select** keyword for each feature we want to dynamically bind in ambiguous cases. Such an approach has the drawback that the feature corresponding to the selected features in the subclasses, can be non-exherited in the generalizing class.

3.4.4 Architectural Restrictions

From class relations point of view, in any common object-oriented language a class hierarchy based on inheritance cannot be cyclic. So in reverse inheritance based hierarchies the same rule is applied. This rule is applicable also to hierarchies containing both inheritance and reverse inheritance. Any further architectural restrictions can be set only in the concrete context of a programming language. The philosophy of the language will determine: if repeated reverse inheritance is possible or not and what happens in the presence of ordinary inheritance and reverse inheritance having the same target class.

3.5 Summary

This chapter contains all general ideas about exheritance and its semantical elements. First the motivation is presented, the exheritance concept being found in object-oriented databases and in class modeling of the object-oriented paradigm. Formal definitions of single/multiple specialization/generalization, based on the notion of the intension and extension of a concept, are given. Some exheritance basic semantical elements are discussed: common features, type conformance, generalization and specialization. The reverse inheritance concept can be located at the intersection of object-oriented programming and object-oriented databases. Each domain has its own

interest of reuse: class reuse, respectively object reuse. Several contexts are presented in which reverse inheritance is the main means in achieving reuse. The issue of allowing the source class of exheritance as empty is analysed in the context of several programming languages. Then the problem of having available the source code of the reverse inheritance affected classes is discussed. This issue belongs more to the implementation part where such decisions have to be discussed. The decision to include the discussion in this chapter is to suggest the several implementation semantics possibilities.

Next, we focus on the interface content of a superclass in the reverse inheritance class relationship. The first analyzed issue is the abstract/concrete status of the exheritance source class. Then the influence of modifiers is discussed. In this sense the protection mechanism is considered as subject for the analysis. Abstract/concrete status of method is an important point of discussions. When discussing about the interface of the generalization class, type conformance has to be demonstrated using formalisms. Common feature and assertions problematics are discussed in the context of Eiffel language. An extensive section is dedicated to conflicts caused by inhomogeneities of common features. They are classified as name conflicts, value conflicts, scale conflicts and parameter conflicts. In this sense several possible adaptations techniques are presented.

Further on, implementation exheritance problems were discussed. The most severe problem is the one of polymorphism impact on the generalizing class. When there is no polymorphism implementation exheritance is problematic. In the case of polymorphic methods there is no problem since they can be very easily overridden in the subclass. The adding of new behavior in the generalizing class is studied in the context of exheritance. This capability is a disabled option since the semantics of reverse inheritance does not include feature inheritance and the semantics of exherited classes is changed, which is a severe aspect. Then dependencies problems are analysed when implementations are exherited and two solutions are discussed. A special aspect related to type verifications is discussed in the context of implementation exheritance.

Finally, we analysed one combination of ordinary and reverse inheritance. The fork-join inheritance shows the benefits of using together the two class relationships. Next, was presented an evolution problem, which was solved using reverse inheritance as the basic means in several solutions: using specialization, adding features in the foster class and setting superclass for the foster class. Dynamic binding problems were tackled presenting two unsatisfactory solutions. At the end of the chapter several architectural restrictions are discussed.

From the study presented in this part results clearly the fact that a general semantics for reverse inheritance to fit in all object-oriented programming languages it is not possible. Even its counterpart, ordinary inheritance has several implementations in each programming language. This is because every language has its own particularities and a general compromise cannot be found.

In conformance to the ideas studied in this report we can draw conclusions related to the most suitable programming languages to implement the reverse inheritance concept. If we decide to implement it in Java we have to take into consideration the fact that there is no multiple inheritance between classes in consequence we cannot design a reverse inheritance class relationship. Because there is multiple inheritance between interfaces we could introduce the concept of reverse inheritance between interfaces. This decision is based on the fact that it is not a good thing to allow the creation of class hierarchies with the help of reverse inheritance which cannot be obtained using ordinary inheritance. In order to avoid semantical inconsistencies it is better to keep the symmetry of the language.

C++ programming language has the advantage of having multiple inheritance, thus favoring reverse inheritance between classes. There are no adaptation mechanisms for the features, adding them with the new concept would break the philosophy of the language. There is a great difference between class features: attributes and methods from the client point of view. This aspect can introduce potential problems at the implementation level.

In Eiffel the implementation of such a concept like reverse inheritance would fit better because of several reasons. One main reason is the fact that the language supports multiple inheritance. Another big advantage is the philosophy of the language which includes feature adaptations. Reverse inheritance needs such adaptation mechanisms. Another argument in favor of Eiffel is the

uniqueness of the feature names. This is due to the fact that no overloading is possible. From the client class point of view there is no difference between a method or an attribute query. In other words a feature can be implemented by computation or by storage in a free way. A possible problem related to the other programming languages is the existence of assertions which can easily prevent potential features from being exherited.

In order to experiment the reverse inheritance concept it would be appropriate to define a semantics for a certain programming language. As presented earlier the Eiffel programming language would fit better to this research. In this direction, the definition of a reverse inheritance semantics would imply setting rules and syntax constructions, giving examples and explanations. The semantics has to contain all the interactions and side effects of the reverse inheritance concept and the rest of the language mechanisms.

The implementation of a prototype will allow using the reverse inheritance class relationship between already existing classes, thus building new class hierarchies. In fact this prototype will implement the semantics of reverse inheritance. One way of doing this is to transform class hierarchies using reverse inheritance in an IDE (Integrated Development Environment). From this point of view there are several possibilities for obtaining the executable system: either by writing a compiler capable of generating binary code for the reverse inheritance class relationship, or by writing a translator to convert the source code using reverse inheritance extension to equivalent pure source code.

Part II

The Design of an Exheritance Relationship

In this part we propose a dual class relationship of reverse inheritance, in order to achieve the goal of class reusability and behavior enhancement. The idea of dual relationship comes from the fact that we use two conceptual links between classes: reverse inheritance (conforming and non-conforming) and a feature importing, like-type class relationship. Further on we will explain in detail each class relationship semantics.

The approach of this work is built on [LHQ94], which we consider the most advanced approach from the state of the art, in this direction [**C**CL$^+$05d]. We keep the same restrictions of not affecting the behavior of exherited classes when exheriting from them. The choice of Eiffel was taken because its philosophy is closer to the concepts required by reverse inheritance e.g. the renaming technique, the presence of conforming and non-conforming inheritance, the lack of overloading for features. There are some attempts to implement reverse inheritance for Java programming language [**C**Pc05, CR**C**06b] and even to adjust its semantics in order to solve a restricted set of problems [**C**RC$^+$06a], but the resulted semantics is a great deviation from the philosophy of Java.

A secondary objective of this part is to prove that the integration of the reverse inheritance class relationship in Eiffel comes naturally, will not complicate the language semantics and will not break any already existing language rules. Reverse inheritance is also known in literature as upward inheritance [SN88], exheritance [Sak02], generalization [Ped89, OMG04].

Chapter 4

Creating a Class by Reverse Inheritance

4.1 Reverse Inheritance: Definition and Notations

In this chapter we intend to define the semantics of conforming reverse inheritance. In order to do this, we will rely on the ordinary inheritance definition. Inheritance allows the definition of new classes by adding or adapting features in subclasses without changing their semantics. Along with inheritance, the definition of new types is supported, as specializations of the already existing ones [Mey02]. In Eiffel there are two types of inheritance: conforming and non-conforming. **Conforming inheritance** offers feature inheritance and subtype conformance between the subclass and superclass. **Non-conforming inheritance**[1] does not offer type conformance as conforming inheritance does, but only inheritance of features. So, it is more useful when data and code are needed to be imported into a class without making it a subtype of the superclass.

In order to follow the philosophy of the language, we think that reverse inheritance should behave in the same way. In this chapter we address conforming reverse inheritance because it deals with all complex situations, but in section 4.4 we will point out the characteristics which are specific to non-conforming reverse inheritance.

Reverse inheritance class relationship in general, conforming or non-conforming, has a target class and one or more source[2] classes. They will be referred further on also as the superclass and respectively subclasses or exheritant class and respectively exherited classes.

When defining default behaviors of reverse inheritance semantics they will be defined as alternatives. On the other hand they should not be declared explicitly because in Eiffel most defaults are unnamed, for instance: there are no keywords for the alternatives of **deferred** or **frozen**.

Together with the notion of reverse inheritance we will use also an alternate name: **exheritance**, denoting the same concept. The features which are the subject to reverse inheritance will be called **factored**, **reverse inherited** or **exherited**. In order to distinguish between pure Eiffel and Eiffel with reverse inheritance we will name the extended language **RIEiffel**.

In the next subsections we will set the main principles of reverse inheritance class relationship and we will propose a notation.

[1] We can think also about removing features in non-conforming inheritance since the subclass will not conform to the superclass.

[2] By source classes we mean the classes which exist initially, and by target classes we mean the classes which are created afterwards.

4.1.1 Definitions

We want to build a class relationship which is completely interchangeable[3] with its symmetrical counterpart, namely the ordinary inheritance. This refers to both conforming and non-conforming cases.

From this issue derives the fact that reverse inheritance is not an absolute necessity. Then the question arises whether this class relationship is good anyway. Adaptations which are used for merging features are good arguments for sustaining such a class relationship. Some rules will give an intuitive definition of reverse inheritance.

Firstly, since we are designing an extension to an existing language, it is important that classes and programs which do not use the extension will not be affected.

Rule 1: ***Genuine Extension*** Eiffel classes and programs that do not exploit reverse inheritance must not need any modifications, and their semantics must not change.

As the target class is created the last, it should not affect the rest of the hierarchy from the behavioral point of view. Using reverse inheritance we should not achieve structures which are not possible to create with ordinary inheritance.

Secondly, it is very important that after a class has been defined by reverse inheritance, it can be used just as any ordinary class. Otherwise, foster classes would be less useful and the additional language complexity caused by reverse inheritance would not pay off.

Rule 2: ***Full Class Status*** After a foster class had been defined, it must be usable in all respects as if it were an ordinary class. In particular, a foster class can be used as a parent in ordinary inheritance and as an heir in further reverse inheritance.

Thirdly, in ordinary inheritance the semantics of a given class is not affected if a new class is defined as its direct or indirect subclass (descendant in Eiffel), or if some existing descendants are modified. In contrast, any modification to a superclass (ancestor in Eiffel) affects all its subclasses, and can even make some existing descendants illegal unless their redefinitions are changed also. Reverse inheritance is designed to be a mirror image of ordinary inheritance, in this respect the dependencies between classes will be the opposite of what they are in ordinary inheritance.

Rule 3: ***Invariant Class Structure and Behaviour*** Introducing an ancestor C to one or several classes $C_1, \ldots, C_n$ using reverse inheritance must not modify the structure and behaviour of $C_1, \ldots, C_n$, nor induce new inheritance relationships between existing classes.

The paper [Sak02] suggested that it could be possible to define also new inheritance relationships between existing classes (if they are feasible). However, that is more confusing than useful, at least when extending an existing language. Since we are designing an extension to an existing language, it is important that classes and programs which do not use the extension will not be affected.

Fourthly, the reverse inheritance relationship is intended to be symmetric with ordinary inheritance. This means that it should be as completely interchangeable with ordinary inheritance as possible. In the new version of Eiffel [Int06] this would imply also that conforming and non-conforming reverse inheritance relationships must be distinguished.

Rule 4: ***Equivalence with Ordinary Inheritance*** Declaring a reverse inheritance relationship from class A to class B should be equivalent to declaring an ordinary inheritance relationship from class B to class A. Of course, this does not mean that the syntactic definitions of the two classes would be the same in both cases. As a consequence of this rule, it would be good if all adaptation capabilities provided for reverse inheritance had their counterparts in pure Eiffel language. However, we actually wish to have some adaptations that cannot be exactly

[3] By interchangeable we admit in this context that some modifications have to be made to feature clauses in order to obtain the same behavior from the class hierarchy.

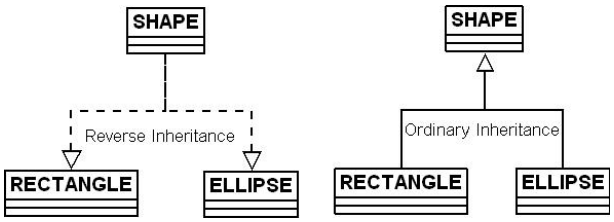


Figure 4.1: Reverse Inheritance

translated to ordinary inheritance (see section 5.2). On the other hand, we did not consider it worthwhile to implement all possible complications of Eiffel ordinary inheritance also in reverse inheritance; rule 7 is an example of that.

Fifthly, we want reverse inheritance to leave the existing inheritance hierarchy as intact as possible.

Rule 5: ***Minimal Change of Inheritance Hierarchy*** Introducing a foster class must neither delete direct inheritance relationships (parent-heir relationships) nor create any inheritance relationships (ancestor-descendant relationships) between previously existing classes. Note that, taking rule 4 into account, reverse inheritance may well create new inheritance paths between existing classes, but only for existing ancestor-descendant pairs (section 6.1). The paper [Sak02] suggested that it could be possible to define also new parent-heir relationships, and even equivalence relationships, between existing classes (if they are feasible). However, we decided not to include that possibility in our Eiffel extension, to keep things simpler.

Sixthly, we need to define which features are candidates to be exherited in reverse inheritance. The following rule is essentially a consequence of the previous rules and the adaptation possibilities of ordinary inheritance in Eiffel extended for reverse inheritance (as just mentioned).

Rule 6: ***Exheritable Features*** The features $f_1, \ldots, f_n$ of the respective, different classes $C_1, \ldots, C_n$ are exheritable together to a feature, in a common foster class if there exists a common signature to which the signatures of all of them conform, possibly after some adaptations. Each of the features $f_1, \ldots, f_n$ can be either immediate or inherited. In pure Eiffel these features could be similarly factored out to a common parent, but any extended adaptations (see above) would require new or modified methods in the heir classes. Some common special cases are simpler than the general case. In single reverse inheritance, all features are trivially exheritable. In multiple reverse inheritance, all f_i may already have the same signature, or one of them may have a signature to which all others can be made to conform. We will explain the possible adaptations in section 5.2.

Lastly, we want to avoid the complexity of allowing one feature in a foster class to correspond to several features in the same exherited class, although this would be a direct equivalent of repeated inheritance with renaming.

Rule 7: ***No Repeated Exheritance*** Two different features of the same class must not be exherited to the same feature in a foster class. The definition of the semantics of reverse inheritance in the following sections, on both the conceptual level and the concrete language level, relies on the above six rules.

4.1.2 Text and Graphical Syntax

In figure 4.1 we present a class diagram which shows how to represent the reverse inheritance class relation in parallel with the ordinary inheritance. The UML representation selected for reverse inheritance is the dashed line having a downward pointing triangle arrowhead. The intention of

Example 38 Reverse Inheritance Example

```
class RECTANGLE
end
class ELLIPSE
end
foster class SHAPE
 exherit
 RECTANGLE
 ELLIPSE
 all
end
```

Example 39 Ordinary Inheritance Equivalent Example

```
class SHAPE
end
class RECTANGLE
 inherit SHAPE
end
class ELLIPSE
 inherit SHAPE
end
```

this two class diagrams is also to show that reverse inheritance is the symmetrical relationship of ordinary inheritance, the behavior of the two class diagrams is intended to be equivalent. This class relationship can be expressed also using a syntax extension like in the following code:

In example 38 the classes *RECTANGLE* and *ELLIPSE* already exist, and the superclass *SHAPE* is created afterwards. One can notice that the keyword **exherit** was used in order to reflect the reverse inheritance relationship between the classes. On the other hand class *SHAPE* is a special kind of class, the source of reverse inheritance, and the **foster** keyword is used to mark it with respect to normal Eiffel classes. The **foster** keyword was taken from [LHQ94] paper and it is very suggestive for the adoptive status of the class. A foster class can be effective or deferred, may have subclasses by ordinary inheritance or even superclasses by reverse inheritance. The code from example 38 is semantically equivalent to the situation in which class *SHAPE* is created first, and then subclasses *RECTANGLE* and *ELLIPSE,* like in example 39.

Reverse inheritance[4] allows the programmer to create a superclass out of one or several subclasses and to select the **common features** to be factored out into that superclass.

In order to specify the common features we have several choices: either we specify them explicitly, either we consider them implicitly selected. With features, some actions are possible like: **rename**, **undefine**, **adapt**, **moveup**, **select**, **export**, **redefine**. These facts are illustrated in the rules of example 40.

In the grammar from example 40 we provide the definition of the foster class which is the source of reverse inheritance class relationship. We can notice that it starts with the keyword **foster** which denotes the special type of the defined class for better readability. In this rule *EXHERITANT_CLASS_NAME* is the name of the foster class.

Common features from subclasses can be exherited (or factored). Exheritable features in the subclasses are those features which have either the same signature or those for which the signature may be adapted using the clauses presented in chapter 5. In the selection clause, the **exherit** keyword is then used in different combinations to select the exherited features:

[4]When we do not mention conforming or non-conforming it means that we discuss about reverse inheritance in the general sense.

Example 40 Syntax for Exheriting Features

```
foster_class_definition::=
  foster class EXHERITANT_CLASS_NAME
  ...
  exherit heir_list
  exherited_feature_list
  [foster_adaptation]
  ...
end
heir_list::= heir*
heir::=
  EXHERITED_CLASS_NAME
    [rename renaming_list]
    [redefine feature_list]
    [undefine feature_list]
    [adapt feature_list]
    [moveup feature_list]
    [select selection_list]
  end
renaming_list::=
  rename identifier as new_identifier (, identifier as new_identifier)*
selection_list::=
  select feature_name in class_name (, feature_name in class_name)*
exherited_feature_list::=
  exherit (all | nothing | only feature_list| except feature_list)
feature_list::= feature_name (, feature_name)*
foster_adaptation::=
    [export export_list]
```

- using the **all** keyword denotes that all possible features from exherited classes will be selected. If there are no common features in the subclasses the resulting foster class will be empty. The effective list of selected features is not explicitly listed, being inferred by the compiler, but it could be highlighted by the programming environment.
- using the **nothing** keyword denotes that no features from exherited classes will be selected. This keyword can be useful for the creation of a new type.
- using the **only** keyword and a list of features explicitly selected from the subclasses. If there are explicitly selected features which do not exist in all foster classes, the compiler will consider it an error. This declaration alternative has the advantage of having an explicit list of selected features, thus increasing clarity of foster class code for the programmer.
- using the **except** keyword and a list of features explicitly excluded from the selection. This choice works better in cases where multiple features are exherited and just a few exceptions must be stated. If non-eligible features are excluded the compiler will produce a warning. As in the first declaration alternative, the programming environment could highlight the effective list of the exherited features.

Rule ***Exheritable Features***. A feature f contained in the subclasses C_1.. C_n is exheritable if one of the following assumptions is satisfied:

- The signatures of f in all C_i ($1 \leq i \leq n$) are identical;
- The signatures of f in all C_i ($1 \leq i \leq n$) may be adapted in order to conform:
  - either to the signature of f in one C_j ($1 \leq j \leq n$) ;
  - or to another signature to which all the signature of f in C_i ($1 \leq i \leq n$) may conform, possibly after some adaptations.

Next, the foster class grammar contains exheritance clauses, which must be specified for each exherited class. One exheritance clause specifies the condition for the feature exheritance from one subclass and it has the following structure:

- *EXHERITED_CLASS* is the name of the exherited class;
- The rule variant **rename** *renaming_list* refers to the ordinary renaming mechanism of Eiffel. In the context of foster class semantically equivalent features must have the same name.
- The **redefine** clause has the same semantics as in ordinary inheritance of Eiffel and it is used for signature redefinitions or for implementation redefinitions.
- The **undefine** clause will have the same semantics as in the context of ordinary inheritance, meaning that all exherited features will be deferred in the foster class. This is the implicit behavior for both attributes and methods. The use of **undefined** keyword is not necessary since it is the default behavior for all exherited features.
- The keyword **adapt** is used to list the features which need adaptations. The adaptations will be provided in the implementation of the feature. It will be used for all adaptations that cannot be performed by the **redefine** and **undefine** clauses. These issues are developed in chapter 5.
- The **moveup** clause allows to specify the features from a subclass which come with their implementation in the superclass. In other words, the keyword **moveup** is used for implementation exheritance (or concrete version exheritance).
- The **select** clause will be used to mark a feature to be used in special dynamic binding situations of repeated inheritance. The semantics of select will be discussed in section 6.2.

Figure 4.2: Dequeue Example

After the specification of exheritance branches with their adaptation clauses there are some other adaptations which belong to the foster class and not to the exheritance branches. They are placed after the exheritance branches section of the foster class because, common features will be known to the foster class by their final names:

- The **export** clause specifies lists of features and lists of client classes where the features are exported to. The semantics is the same as in ordinary inheritance.

The compatible combinations of these clauses will be studied in chapter 6, as well as their impact on dynamic binding.

Finally, the foster class contains feature declarations using the regular Eiffel syntax. Some features can be adapted and their body will contain special syntactical elements which will be presented in chapter 5.

4.2 Single/Multiple Reverse Inheritance

Eiffel supports multiple inheritance, so it seems quite natural that we introduce multiple reverse inheritance. If we deal with only one subclass, then we have single reverse inheritance, while when having multiple subclasses we deal with multiple reverse inheritance. Single exheritance seems to be useful especially when we already have a specialized class and we want to reuse only a part of it, by creating a more general class. In figure 4.2 and 4.3 we will analyze the case of the *Dequeue* example taken from [Ped89].

4.2.1 Single Reverse Inheritance

The example proposed in figure 4.2, shows a class *DEQUEUE* which has two sets of features for the operations related to each end of the dequeue: *push*, *pop*, *top* for one end and *push2*, *pop2*, *top2* for the other end. There is a global method *empty* which conceptually belongs to the dequeue, and equally to both ends. In figure 4.3, a new superclass *STACK* is created by reverse inheritance, exheriting only the operations dedicated to one end of *DEQUEUE* like *push*, *pop*, *top* and *empty*. This example shows a possible use case where single reverse inheritance can be useful. Class *STACK* can be defined using the syntax extension like in example 41.

Due to the flexible syntax we can have two ways for defining the foster class. One way is to implicitly exherit everything except a certain list of features. The other way is to explicitly list the features which are exherited. In both cases it is necessary to specify whether the implementation is exherited or not along with the signature of exherited features. We may even create an empty superclass using the reverse inheritance but in practice such a class does not seem to be very useful.

When using single exheritance, since there is only one subclass, all features of the subclass may be exherited, even with their implementation. In such a case no signature conflict may arise, since the exherited feature in the subclass will have the same signature, even the same implementation, if needed.

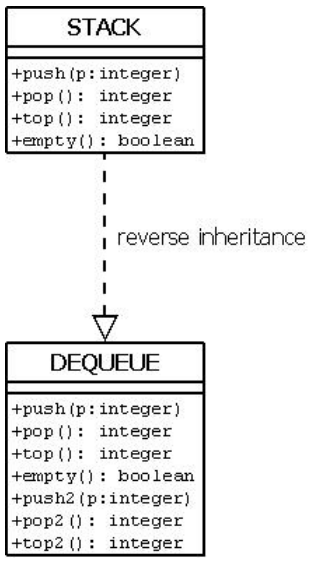


Figure 4.3: Dequeue Class Diagram

Example 41 Dequeue Class

```
class DEQUEUE
feature
  push(p:INTEGER) is do ... end
  pop: INTEGER is do ... end
  top: INTEGER is do ... end
  push2 (p:INTEGER) is do ... end
  pop2: INTEGER is do ... end
  top2: INTEGER is do ... end
  empty: BOOLEAN is do ... end
end
foster class STACK -- variant 1
exherit
  DEQUEUE
   moveup
     push, pop, top, empty
  end
  except push2, pop2, top2
end
foster class STACK -- variant 2
exherit
 DEQUEUE
  moveup
     push, pop, top, empty
 end
 only push, pop, top, empty
end
```

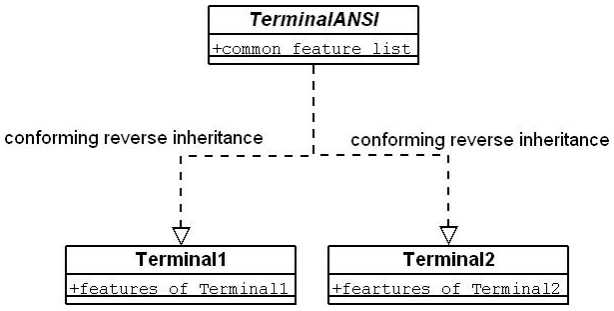


Figure 4.4: Multiple Reverse Inheritance

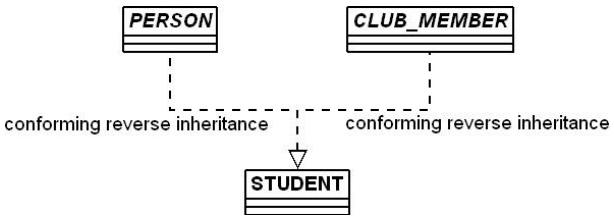


Figure 4.5: Two Independent Reverse Inheritance Relationships

4.2.2 Multiple Reverse Inheritance

Multiple reverse inheritance is a special case of reverse inheritance where multiple source classes are involved. Such a class hierarchy is equivalent to several ordinary inheritance relationships having as superclass the foster class. We will rely on the exheritance clauses in order to resolve possible conflicts or to perform the necessary adaptations[5].

In figure 4.4 we intent to show how such a target class can be designed starting from two concrete classes using multiple reverse inheritance. We propose an example based on terminals adapted from [Ped89], in which starting from two terminal class implementations we decide to design an abstract superclass to abstract the behavior of an ANSI terminal. The newly created abstract class, *TerminalANSI*, will contain common feature signatures declaring the behavior of the ANSI standard terminal.

4.2.3 Several Independent Reverse Inheritance Relationships

The cases in which several superclasses exherit from a class, like in figure 4.5, are not multiple reverse inheritance, it is just the fact that several reverse inheritance relationships which happen to have the same target class. In figure 4.5, *STUDENT* is a subclass for both *PERSON* and *CLUB_MEMBER*. This kind of architectural decision can be taken when the two different superclasses are needed for a certain class hierarchy. This can be useful when we want to partition a class for a better reuse or when different points of view on the same type are needed. The two superclasses will exherit features independently from the common subclass. From the type point of view, the two superclasses are supertypes for the subclass. The class hierarchy is equivalent to a retroactive multiple inheritance structure. The common subclass will conform to each superclass created by reverse inheritance. An ambiguity related to feature exheritance may arise if some features from a subclass are multiply exherited into two or several superclasses. As we specified

[5]This will be studied in detail in chapters 5 and 6.

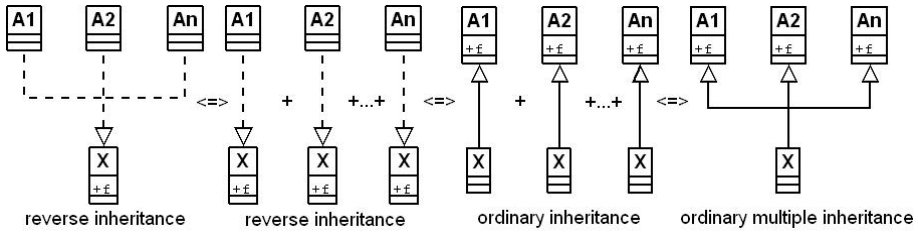


Figure 4.6: Several Independent Reverse Inheritance Relationships

above, the two exheritance class relations are independent, so the same feature can be exherited independently into several superclasses.

In order to be consistent with the semantics of ordinary inheritance we show that such a class hierarchy has an equivalent based on ordinary inheritance. In order to do this, we will analyze a more general case of several independent class relationships like those in figure 4.6.

In the general case of figure 4.6 we start from the initial situation in which several classes A_1, A_2, ... ,A_n have the same subclass X. This class hierarchy can be decomposed in multiple reverse inheritance relationships between A_i and X where i=1..n. These reverse inheritance class relationships can be transformed into ordinary inheritance equivalent relationships. All these combined will form a configuration of an equivalent multiple inheritance relationship. So, we proved that a configuration of several independent reverse inheritance relationships having the same source class is equivalent with a multiple ordinary inheritance.

4.3 Feature Factorization

The need to exherit common features is present in both types of reverse inheritance (conforming and non-conforming). By common features we mean the features which have the same semantics in the context of the given class hierarchy [**C**CL$^+$04c, **C**CL04a, **C**CL04b]. Common features having the same signature can be automatically exherited, while features having different signatures have to be adapted using a special syntax extension. Such situations which need adaptations are discussed in chapter 5.

4.3.1 Implicit Rules Regarding Feature Exheritance

In this subsection we will show how to declare the exherited features, when these features are exherited implicitly or explicitly and what is the nature of the features in the foster class.

Implicit Rules Regarding Attribute Exheritance

When in several subclasses we have attributes with the same signature, or attributes whose signature may be adapted in order to conform to a common signature in the superclass and, if the attribute is marked as exherited implicitly or explicitly, then the declaration of a deferred feature with the same name is automatically inserted. This means that implicitly the feature will be deferred.

In example 42, attribute a has the same signature in all subclasses A and B. In class C it will be implicitly exherited as deferred feature having the same common type T.

Rule ***Attribute Exheritance - The Default***. When an attribute is exherited, from several subclasses it is deferred implicitly in the foster class.

If we want to create a concrete feature from the attributes of the subclasses, then we have to move an instance from one subclass or to redefine the exherited attribute. In example 43, we have

Example 42 Implicit Rules for Attribute Exheritance (1)

```
class A
feature
 a: T
end
class B
feature
 a: T
end
deferred foster class C
 exherit
 A
 B
 all
feature
  -- a: T is deferred end
  -- is implicit
end
```

the same class configuration as in example 42, but the feature a is redefined in the foster class. Redefinition in Eiffel serves two purposes: one is related to the attachment of an implementation to a deferred feature and the other is for covariant signature redefinition. In our case by redefinition we aim its first purpose.

In example 44 we present the other possibility of exheriting the concrete version of an attribute, by "moving up" one concrete version from the exherited classes into the foster class.

On the other hand, the rules of Eiffel do not allow to undefine an attribute neither to redefine an attribute as a method. As a consequence in reverse inheritance we can exherit a concrete attribute only if it is an attribute in all exherited classes.

Rule ***Attribute Exheritance***. When an attribute is exherited from several subclasses but should be effective in the superclass it has to be redefined in the foster class or moved up on one exheritance branch. If we also want to adapt its signature it is necessary to provide a conforming redefinition declaration in the foster class.

Rule ***Attribute Exheritance.*** An attribute can be exherited as concrete in the foster class if it is a concrete attribute in all exherited classes.

Implicit Rules Regarding Method Exheritance

When there are methods which are exherited, we have to consider the signature and the body. Because it is supposed that exherited classes are developed in different contexts, it is very likely that methods will not have the same body. This is the reason why we decided that it is better to exherit by default only the signature of the method. To do this we can proceed like in example 45. Since method m has the same signature in both subclasses and it is marked as exherited in both of them, it is implicitly exherited as deferred like it is shown in the last two commented lines.

Rule ***Method Exheritance - The Default***. When exheriting a common method from subclasses implicitly the signature is exherited meaning that the corresponding feature in the superclass is implicitly deferred.

To select the implementation from one of the subclasses for a given exherited feature[6] we have to use a **moveup** clause on one exheritance branch. The implicit behavior of exheriting common

[6]The conditions in which an implementation can be moved into the foster class are discussed in detail in chapter 6.

Example 43 Implicit Rules for Attribute Exheritance (2)

```
class A
feature
  a: T
end
class B
feature
  a: T
end
foster class C
 exherit
 A
 B
 all
 redefine a
feature
  a: T
  -- due to the redefine clause
end
```

Example 44 Implicit Rules for Attribute Exheritance (3)

```
class A
feature
  a: T
end
class B
feature
  a: T
end
foster class C
 exherit
 A
  moveup a
 end
 B
 all
feature
  -- a: T
  -- due to the moveup clause
end
```

Example 45 Implicit Rules for Method Exheritance (1)

```
class A
feature
  m(p: T1): T2 is do ... end
end
class B
feature
  m(p: T1): T2 is do ... end
end
deferred foster class C exherit
 A
 B
 all
feature
  -- m(p: T1): T2 is deferred end
  -- is implicit
end
```

features as deferred will cause the undefinition of all the exherited features except the one being moved up in the foster class. This is illustrated in example 46. The implementation of m from class A has been selected for method m in foster class C. This was done by using in the exheritance branch corresponding to class A in foster class C the **moveup** clause for the m method and the undefine clauses in the rest of the exheritance branches works implicitly. In this case only one exheritance branch is left to be undefined implicitly, the one corresponding to class B.

In the case of multiple inheritance [Mey02] a conflict arises when two or more features with the same name and different implementations are inherited from several superclasses. One solution is to undefine all features from superclasses, thus leading to a deferred feature in the subclasses. Another possibility is to undefine all features except the one which will provide the implementation in the subclass. To provide a new implementation in the subclass will require to redefine all the inherited features from the subclasses.

Rule ***Method Implementation Exheritance*** **- The Default.** When a method is exherited with its implementation from one subclass, in other words when it is moved up, all the corresponding methods from the other subclasses are undefined by default.

4.3.2 Allowing Implicit and Explicit Common Feature Selection

In this subsection we discuss about the benefits and drawbacks of explicitly or implicitly declaring the common features. It must be noted that the selection of exherited features is made globally at the foster class level and not on each exheritance branch.

Implicit All Common Feature Selection

The **exherit ... all** keyword combination will denote that all exheritable features are selected for factorization. In example 47 foster class *SHAPE* will exherit features *area* and *color* from *RECTANGLE* and *ELLIPSE* classes. Features *boundary* from class *RECTANGLE* and *circumference* from class *ELLIPSE* do not belong to the set of exheritable features since they have different names.

This selection choice is good in the case of multiple feature selection without excluding some features. In the case of single exheritance the keyword combination will exherit all features from the class since all are exheritable.

Example 46 Implicit Rules for Method Exheritance (3)

```
class A
feature
  m(p: T1):T2 is do ... end
end
class B
feature
  m(p: T1):T2 is do ... end
end
foster class C
 exherit
 A
  moveup m
 end
 B
  -- undefine m
  -- is implicit
 all
feature
  -- m(p:T1):T2 (with the body of m from class A)
end
```

Example 47 Implicit All Common Feature Selection

```
class RECTANGLE
feature
  area:REAL
  color:INTEGER
  boundary:REAL
end
class ELLIPSE
feature
  area:REAL
  color:INTEGER
  circumference:REAL
end
foster class SHAPE
 exherit
 RECTANGLE
 ELLIPSE
 all
end
```

Example 48 Explicit Common Feature Selection

```
class RECTANGLE
feature
  area:REAL
  color:INTEGER -- foreground color
end
class ELLIPSE
feature
  area:REAL
  color:INTEGER -- background color
end
foster class SHAPE
 exherit
 RECTANGLE
 ELLIPSE
 only area
end
```

Explicit Common Feature Selection

We state in example 48, the syntax used for the explicit declaration of a common feature subset. Class *SHAPE* exherits *RECTANGLE* and *ELLIPSE* having a common attribute named *area*, this attribute being explicitly selected for exheritance. According to the rules related to attribute exheritance, the exherited attribute in the foster class will be a concrete feature, an attribute having the same common type, like in the subclasses.

In this context the common features can be explicitly selected to be factored in the foster class using the **exherit ... only** keyword combination. One of the advantages is that the explicit list of features will increase the foster class code clarity. On the other hand, if a lot of candidate features must be exherited the other options have to be considered since the list of features will grow, and the exclusion of non-exherited features is simpler. All the features listed for exheritance must be eligible candidates for exheritance otherwise the code is semantically incorrect. In such case the compiler will generate an error.

Implicit Common Feature Selection

In the other context where the exheritance is made implicit there should exist a possibility for the programmer to avoid the exheritance of some features using the **exherit ... except** keyword combination. This can be useful when features with the same signature have different semantics. In this case the compiler will infer automatically the actual list of exherited features. This actual list being implicit will affect code readability. This problem can be solved by the programming environment by highlighting the actually exherited features.

In any case the features which do not have the same signature (or an adapted/redefined one) will not be exherited automatically. In case the exclusion list contains features which are not valid candidates for exheritance, the compiler will issue a warning.

In example 49 we consider that common features are automatically exherited, it would be the case for *area* attribute of both *RECTANGLE* and *ELLIPSE* classes. Since *color* attribute has different semantics in the two classes, it should not be exherited. Attribute *boundary* of class *RECTANGLE* will not be exherited because it appears only in class *RECTANGLE*. It is the same case for the attribute *circumference* which is only declared in class *ELLIPSE*. If two features have different names but represent the same feature (apparently *boundary* and *circumference* do represent the same behavior let's say *perimeter*) then they have to be mentioned explicitly in an appropriate renaming clause.

Example 49 Implicit Common Feature Selection

```
class RECTANGLE
feature
  area:REAL
  color:INTEGER -- it is the foreground color
  boundary:REAL
end
class ELLIPSE
 feature
  area:REAL
  color:INTEGER -- it is the background color
  circumference: REAL
end
foster class SHAPE
 exherit
 RECTANGLE
 ELLIPSE
 except color
end
```

No Feature Selection

In some cases the creation of a new type is necessary and no common features need to be exherited. Such a selection can be achieved using the **only** keyword, an empty feature list and the **except** keyword and the list of all features.

In example 50 the use of **nothing** keyword is more intuitive, for creating a new type *SHAPE*.

4.3.3 Influence of the Nature of Common Features

In this subsection is analyzed the nature of common features in the process of exheritance. Common features in the exherited classes, regarding their nature, can be: attributes, methods or deferred features.

Factoring Features Represented by Attributes

If we deal with common attributes in the subclasses they can be exherited as deferred (implicitly) or concrete (by moving up or by redefinition) features in the superclass. As it is mentioned in [Sak02] only some type and name conflicts may occur. Types will be discussed later, in section 5.2, while name conflicts are analyzed in section 5.1.3. A basic situation of exheriting attributes having the same signatures is presented in example 51.

The two original existing classes *RECTANGLE* and *ELLIPSE* have one *perimeter* feature each, with the same name. The case where features have different names is discussed in section 5.1.3. About constant attributes, they cannot be exherited as effective by moving one of them up even if it happens to have the same type and the same values. This behavior is imposed because constant features cannot be redefined in the context of inheritance.

Rule ***Attribute Exheritance 1.*** Common attributes having the same type in the exherited classes are exherited as a deferred feature in the foster class having the same common type.

Rule ***Attribute Exheritance 2.*** If attributes in exherited classes do not have the same type but a supertype for all the types in subclasses exists, that supertype will be used as type for the implicitly exherited deferred feature in the foster class. In this case redefinition statements must be used.

Example 50 No Feature Selection

```
class RECTANGLE
feature
  area:REAL
  color:INTEGER
  boundary:REAL
end
class ELLIPSE
 feature
  area:REAL
  color:INTEGER
  circumference: REAL
end
foster class SHAPE
 exherit
 RECTANGLE
 ELLIPSE
 nothing
end
```

Example 51 Factoring Features Represented By Attributes

```
class RECTANGLE
feature
 perimeter: REAL
end
class ELLIPSE
feature
 perimeter: REAL
end
deferred foster class SHAPE
 exherit
 RECTANGLE
 ELLIPSE
 all
feature
  -- perimeter:REAL is deferred end
  -- this feature declaration is implicit
end
```

Example 52 Factoring Features Represented by Attributes and Methods

```
class RECTANGLE
feature
 area:REAL
end
class ELLIPSE
feature
 radiusA: REAL
 radiusB: REAL
 area is do Result:=3.1416 * radiusA * radiusB end
end
deferred foster class SHAPE
 exherit
 RECTANGLE
 ELLIPSE
 all
feature
  -- area: REAL is deferred end
  -- this feature declaration is implicit
end
```

Rule ***Attribute Exheritance 3.*** If a common supertype for the exherited attributes does not exists in the exherited classes then redefinitions must be applied using a new common supertype.

Factoring Features Represented by Attributes and Methods

Attributes and methods having the same signature[7] are factored implicitly as a deferred feature in the superclass. It is only the case of methods having the same return type and no arguments. According to the philosophy of Eiffel there should be no difference between the implementation of a feature by memory or by computation [Mey97]. This case is illustrated in example 52. The two original existing classes *RECTANGLE* and *ELLIPSE* have the same feature *area* implemented respectively by an attribute and by a method. Both represent the same feature in the given context, so they are factored out implicitly as a deferred feature in the superclass.

Rule ***Attribute and Method Exheritance.*** In case of attributes and methods having no arguments but return types, thus making their signature equivalent, they are exherited in the foster class as a deferred feature. If the types of the exherited features are not the same, redefinitions are allowed in order to achieve a common conforming signature.

Factoring Features Represented by Effective and Deferred Methods

Another case that we consider is the one where all subclasses have a method with the same name and a deferred or effective status. If one or more methods from subclasses are deferred and no suitable implementation can be found among them, then the resulting method in the superclass will be deferred, only the signature which is common, will be factored. In example 53 we illustrate the capabilities of reverse inheritance to factor both deferred and effective features through the case of three classes *RECTANGLE*, *POLYGON* and *ELLIPSE* which initially exist. Only *RECTANGLE* and *ELLIPSE* have implemented a method *draw* (class *POLYGON* declared *draw* as a deferred feature). Obviously, the two implementations of method *draw* seem to be different, so a deferred

[7] It is probable that features with the same semantics have different signatures. Some adaptations can be performed during exheritance to get the same signature for the features. These adaptations will be presented in chapter 5.

Example 53 Factoring Features Represented by Effective and Deferred Methods

```
class RECTANGLE
feature
 draw is do -- rectangle implementation end
end
class ELLIPSE
feature
 draw is do -- ellipse implementation end
end
deferred class POLYGON
feature
 draw is deferred end
end
deferred foster class SHAPE
 exherit
 RECTANGLE
 ELLIPSE
 POLYGON
 all
feature
 -- draw is deferred end -- is implicit
end
...
RECTANGLE r
ELLIPSE e
SHAPE s
create r
create e
s=r
s.draw -- version of RECTANGLE
s=e
s.draw -- version of ELLIPSE
...
```

feature should be chosen in the superclass. Further, the polymorphic behavior of all *SHAPE* instances is illustrated, so the *draw* method can be called on any of them.

Rule ***Effective and Deferred Method Exheritance.*** The effective and deferred features from the subclasses are exherited implicitly as a deferred feature in the foster class.

4.3.4 Factoring Implementation

Another case is where both common signature and the implementation are factored [Ped89]. If we decide to exherit implementation, then exheritance is possible only when all methods that are called and all attributes that are accessed, are exherited as well. Implementation exheritance is made by selecting the feature having the implementation using the **moveup** keyword. The programming environment tool could offer some help to the programmer regarding dependent features: each time an implementation is chosen to be factored, the programmer can be informed automatically about the dependencies. The most simple case is the one in which the methods happen to have the same code, such cases are rare though. The most typical situation is to exherit deferred features in the foster class.

Example 54 illustrates such an implementation exheritance situation.

Example 54 Factoring Implementation

```
class RECTANGLE
feature
 perimeter:REAL is do ... end
 halfperimeter is do perimeter/2 end
end
class ELLIPSE
feature
 perimeter:REAL is do ... end
 halfperimeter is deferred end
end
foster class SHAPE
 exherit
 RECTANGLE
  moveup
    perimeter,
    halfperimeter
 end
 ELLIPSE
 all
 feature
  -- perimeter:REAL;
  -- halfperimeter:REAL is
  --  (RECTANGLE implementation)
  -- end
end
```

In example 54 the two original classes have both a method *halfperimeter*. It is implemented in *RECTANGLE* but deferred in *ELLIPSE*. So the decision what was taken is to exherit the body of the *RECTANGLE* implementation into class *SHAPE*. This is possible only if all references within this method are also exherited. In our case the only feature which needs to be exherited is *perimeter*. Any potential subclass of *SHAPE* will benefit from the exherited behavior with the condition of providing an implementation for feature *perimeter*. Of course, for *ELLIPSE* the feature *halfperimeter* remains deferred.

In the general case implementation exheritance induces several problems. The first problem deals with the dependency of exherited features. The dependency analysis process should not be recursive at compile time. The selection of features which have to be exherited can be made when analyzing each feature in the compilation process. Another issue related to this subject is whether to import automatically by the compiler the dependencies or to let them be selected by the programmer. Dependencies must be either exherited as effective from one of the exherited classes or provided in the foster class by redefinition. It would seem natural to implicitly exherit dependencies as deferred if possible. If dependencies problem cannot be solved by another implementation exheritance or redefinition then implementation exheritance is not allowed. Implementing a dependency which is not present in all the exherited classes would change the behavior of those classes and we do not allow such thing.

Another problem can arise when we decide to exherit several implementations from different subclasses. From the technical point of view there is no problem. In practice, groups of methods which belong tightly together may have been designed differently in those classes. The chance that they can be reused in the foster class are very small.

Type Safety When exheriting the implementation a special attention must be given to the type of *current* calls, argument type and return type of the features we exherit. In covariant

Example 55 Unsafe Type Moveup Example

class FC

```
 exherit
  EC1
  EC2
   moveup f --feature f will call a.m which does not exists in T
  end
 all
 redefine a
feature a:T
end
class EC1
feature
 a:T1
 f is do end
end
class EC2
feature
 a:T2
 f is do a.m end
end
class T end
class T1 inherit T end
class T2 inherit T
feature m is do end
end
```

redefinitions CAT[8] calls may arise, like in example 55.

Feature f is exherited together with its implementation containing a call to method m of an also exherited feature a of type *T2*. Feature a is redefined covariantly in the foster class being of type T. But type T does not support method m, so the $a.m$ call becomes invalid. Such cases can be detected statically at precompile time and implementation exheritance in such cases fails.

4.4 Type Conformance

In this section we address the impact of reverse inheritance on existing regular type conformance of Eiffel. Eiffel entities have different kinds of types which include reference type or an expanded type [Mey02]. Implicitly a class is considered to be a reference type.

Type conformance definition can be found in [Ped89] where it is stated that a type T **conforms** to a type S if instances of type T can be used as if they were instances of type S. We can also say that instances of type T conform to S.

As ordinary inheritance has type conformance between the subclass and superclass, with reverse inheritance we keep the same restriction, but this time all subclasses must be conform to the foster class. As Eiffel is a covariant language, a redefined feature can have a signature in the subclass which can have covariant types. In the signature of a feature, types can interfere as parameter types and return types. Reverse inheritance keeps the same rules regarding covariance. All the types used in the foster class can be supertypes of all the corresponding types in the subclasses.

[8] CAT means Changing Availability of Type.

Example 56 Conforming Reverse Inheritance

```
class RECTANGLE
...
end
class ELLIPSE
...
end
foster class SHAPE
 exherit
 RECTANGLE
 ELLIPSE
 all
end
...
RECTANGLE r
ELLIPSE e
SHAPE s
create r
create e
s=r
s=e
...
```

4.4.1 Conforming Reverse Inheritance

In particular conforming reverse inheritance ensures type conformance between the subclass types and superclass types.

Example 56 shows that instances of the subclasses conform to the type of the superclass. This fact is expressed by the possibility of referencing the subclass typed objects using superclass typed references. The syntax and the examples presented until now addressed conforming reverse inheritance.

Rule ***Type Conformance***. Conforming reverse inheritance ensures that all subclasses conform to the foster class.

4.4.2 Non-conforming Reverse Inheritance

In order to keep the symmetry with ordinary inheritance the reverse inheritance class relationship will have to provide also the semantics for the non-conforming reverse inheritance.

It is known that non-conforming ordinary inheritance in Eiffel is used in the context of feature reuse without keeping conformance between subclass and superclass. This class relationship is known also as "facility inheritance" or "implementation inheritance". In [Mey97] the notion of class is defined as both a software module and a type; this class relationship exploits the module property from the class definition.

With non-conforming reverse inheritance common features are exherited into the target class but without their types to be supertypes[9] of the subclasses. Regarding the semantics of this class relationship we have to state that all rules from the conforming reverse inheritance apply, except those related to type conformance. Conforming and non-conforming reverse inheritance can be combined freely together, without any side effects. Such combination can be used when we want to organize a set of classes, but we want to restrict type conformance only to a set of them. This could be considered as a benefit only from the design point of view. We are using the same syntactic elements as Eiffel for specifying non-conforming inheritance:

[9] In fact the superclass is a subtype, but the language rules disables the conforming behavior.

Example 57 Non-conforming Reverse Inheritance

```
foster class SHAPE
 exherit
 {NONE} RECTANGLE
 {NONE} TRIANGLE
 {NONE} ELLIPSE
 all
end
```

Example 58 Non-conforming Reverse Inheritance (2)

```
foster class SHAPE
 exherit
 RECTANGLE
 TRIANGLE
 {NONE} ELLIPSE
 all
end
```

The syntactical extension proposed implies the usage of the **NONE** keyword in front of every subclass which is non-conform to the declared one. Like its conformance pair, this class relationship can be single and multiple.

Rule ***Non Conforming Reverse Inheritance***. Non-conforming reverse inheritance obeys to all the rules related to conforming reverse inheritance except the rule regarding type conformance between the subclasses and the foster class.

In Eiffel conversions work between non-conforming classes. In the case of non-conforming reverse inheritance we preserve the same behavior, classes related with non-conforming reverse inheritance can be the subject for conversions.

Reverse inheritance branches conforming and non-conforming can be combined easily like in example 58 where we decided that the *ELLIPSE* shapes should not conform to class *SHAPE*, so we specified reverse inheritance as non-conform. All other involved shapes will conform to the foster class.

Rule ***Reverse Inheritance Combinations***. Conforming and non-conforming reverse inheritance can be used together within the same foster class.

4.4.3 Genericity and the Foster Class

In this subsection we will discuss about the relation between genericity and the foster class in the context of type conformance. In ordinary inheritance any generic class *C[U]* conforms to *C[T]* if U conforms to T. Class U may conform to T because of ordinary inheritance (U is a subclass and T is a superclass in the same inheritance hierarchy), but also because of reverse inheritance (T is foster class and U is a subclass in the same inheritance hierarchy).

In example 59 class *SHAPE* is the foster class for *RECTANGLE* and *ELLIPSE*. So any instance of *RECTANGLE* or *ELLIPSE* can be referred using a reference of type *SHAPE*. Combining this type conformance idea in the context of genericity, we created a generic class *LIST[G]* which was instantiated three times with *RECTANGLE*, *ELLIPSE*, *SHAPE* generic parameters. The instances *lr* and *le* can be referred also with *ls* reference, denoting that classes *LIST[RECTANGLE]* and *LIST[ELLIPSE]* conform to *LIST[SHAPE]*.

Rule ***Genericity and the Foster Class.*** Two instantiated, generic classes using the same base class will conform if their generic parameters conform through reverse inheritance.

Example 59 Genericity and the Foster Class

```
class RECTANGLE
 ...
end
class ELLIPSE
 ...
end
foster class SHAPE
 exherit
 RECTANGLE
 ELLIPSE
 all
end
class LIST[G]
 ...
end
...
RECTANGLE r;
ELLIPSE e;
SHAPE s;
create r;
create e;
s=r;
s=e;
...
LIST[RECTANGLE] lr;
LIST[ELLIPSE] le;
LIST[SHAPE] ls;
create lr;
create le;
ls=lr;
ls=le;
...
```

Example 60 Argument, Result Type and the Foster Class

```
class RECTANGLE
end
class ELLIPSE
end
class SHAPE
 exherit
 RECTANGLE
 ELLIPSE
 all
end
class A
 feature f(x,y,z:SHAPE):SHAPE is
  ...
 end
end
class B inherit
 A
  redefine f
 end
 feature f(x,y,z:ELLIPSE):ELLIPSE is
  ...
 end
end
```

4.4.4 Argument, Result Type and the Foster Class

In this subsection we discuss the interaction between feature redeclaration and reverse inheritance. To be more exact the point of interest is argument and result type redefinition in the context of feature inheritance. Redeclaration means **redefinition** or **effecting**. Redefinition may change an inherited feature's implementation, signature or specification [Mey02]. **Effecting** means providing a concrete implementation for a feature originally declared as deferred in the super class. By **argument** we refer to procedure/function parameters and by **result** we refer to function return values or attribute types[10]. The signature of a feature consists of name, parameter number and type, return type. In the redeclaration of a feature, the original arguments and result types can be replaced in the subclass with conforming ones. We intend to show that even types related through reverse inheritance can be used in feature redefinition in a very natural way, as if they would be designed using ordinary inheritance.

In example 60 we consider that class *SHAPE* exherits classes *RECTANGLE* and *ELLIPSE*. Superclass A declares feature f having arguments and result of type *SHAPE*. Class B, subclass of A redefines feature f having arguments and result type *ELLIPSE*. The redefinition involves covariant arguments and result which have been achieved due to reverse inheritance. Class *ELLIPSE* is a subclass of class *SHAPE* linked by reverse inheritance class relationship.

Rule ***Argument, Result Type and the Foster Class.*** In the context of feature redefinition, covariant arguments or result types can be linked by ordinary or reverse inheritance.

4.4.5 Expanded vs. Non-expanded Foster Classes

Following the definition of what expanded and non-expanded classes mean in the context of Eiffel language, the main idea is that the expanded / non-expanded status of a class is not transmitted

[10] It is known that in Eiffel, a feature of the superclass can be redefined as an attribute in the subclass, but the reverse is not allowed.

Example 61 Expanded vs. Non-expanded Foster Classes

```
class A
end
class B
end
expanded foster class C
 exherit
 A
 B
 all
end
```

through inheritance. The implicit status of a class is non-expanded, if the **expanded** keyword is not specified. The non-expanded class object declarations will represent object references, while the expanded ones will represent object values.

In example 61 any instance of class C will be an object value and not a reference to object.

Rule ***Implicit Status of Foster Class.*** The foster class implicitly is considered as non-expanded.

Rule ***The Status of the Descendants.*** The status of the descendant classes in a reverse inheritance class relationship is not affected by the status of the foster class.

As a consequence, it can be stated that there is no restriction about the use of the **expanded** keyword in the foster class.

4.5 Type Exheritance

In this section we will show how the type system of Eiffel can be exherited. In the analysed examples we exherit features having several kinds of return types. The same examples are valid in the case of formal argument types. The types taken into account are class types with all their forms, separate and expanded types which are very similar from the exheritance point of view, like types with all its forms and finally bit types.

4.5.1 Exheriting Class Types

Class types seem to be the most complex types of Eiffel because they can refer class declarations, formal generics and can have recursive actual generics. Class types referring class declarations can have actual generics while class types referring formal generics cannot have. The actual generic types can be any type of the Eiffel type system. Each one of these situations is analysed in the next subsections.

Exheriting Class Types Referring Classes

In example 62 we have a classic situation in which a feature f is using a class type referring a class declaration T. As expected the feature will be exherited implicitly as deferred using the same type T in the foster class FC.

Rule ***Exheriting Class Types Referring Classes.*** Features having class types referring class declarations in their signatures can be exherited if the correspondent class types are identical in all exherited classes.

Example 62 Exheriting Class Types Referring Class Declarations

```
class EC1
 feature f:T is do ... end
end
class EC2
 feature f:T is do ... end
end
class T end
foster class FC
exherit
 EC1
 EC2
 all
feature
 -- f:T is deferred end
end
```

Example 63 Exheriting Class Types Referring Formal Generics

```
class EC1[H]
 feature f:H is do ... end
end
class EC2[I]
 feature f:I is do ... end
end
foster class FC[G]
exherit
 EC1[G]
 EC2[G]
 all
feature
 -- f:G is deferred end
end
```

Example 64 Exheriting Class Types Referring Class Declarations and Having Actual Generics

```
class EC1
feature
 f:T[T1,T2,T3] is do ... end
end
class EC2
feature
 f:T[T1,T2,T3] is do ... end
end
class T[G1,G2,G3] ... end
class T1 ... end
class T2 ... end
class T3 ... end
foster class FC
feature
 -- f:T[T1,T2,T3] is deferred end
end
```

Exheriting Class Types Referring Formal Generics

In example 63 we deal with class types referring formal generics. In such cases because of genericity problems discussed in section 5.3 the class configurations are limited. The example taken is a valid one and the exherited feature in the foster class will have the type G. Features f from *EC1* and *EC2* are of type H, respectively I, which are instantiated by G from the foster class. So the final types of features f are both G, that is why the feature is exheritable.

Rule ***Exheriting Class Types Referring Formal Generics.*** Features having class types referring formal generics in their signatures can be exherited if the correspondent class types are instantiated with the same type in all exherited classes.

Exheriting Class Types Referring Class Declarations and Having Actual Generics

In example 64 a composed type example is taken. Class T takes three actual generic parameters. In each exherited classes, type T is equipped with the same types *T1*, *T2*, *T3* in the same order. Because class types may have actual generics which can be any type of Eiffel we can consider it as a type composing mechanism. In order to exherit such a type it is necessary that the types represented by the actual generics to be exheritable upon the rules of this chapter. The types can be composed on multiple levels, thus the process of exheritance may become recursive.

4.5.2 Exheriting Expanded and Separate Types

In this section we will present expanded and separate type adaptations together since they must obey the same rules. The expanded keyword attached to a class type creates a new type and the behavior of the original class instances is changed. All the instances will be objects and not references to objects. This is the default behavior of objects in C++. Separate types are used for the thread mechanism of Eiffel. The interaction between exheritance and concurrent programming of Eiffel will not be discussed because it is not our research goal, but still some type rules will be issued in the context of reverse inheritance.

In example 65 we have two exherited classes *EC1* and *EC2*. The *f1* features are using expanded types in their signatures and the exherited feature in the foster class has the same expanded types. The *f2* features from the exherited classes use separate type and is natural that the correspondent type in the superclass to be the same separate type.

Example 65 Expanded and Separate Type Exheritance

```
class EC1
feature
 f1(a:expanded T):expanded T is do end
 f2(b:separate T):separate T is do end
end
class EC2
feature
 f1(x:expanded T):expanded T
 f2(y:separate T):separate T
end
class T end
foster class FC
 exherit
  EC1
  EC2
 all
feature
-- f1(a:expanded T):expanded T is deferred end
-- f2(x:separate T):separate T is deferred end
end
```

Rule ***Exheriting Expanded Types.*** Features having expanded types can be exherited if the correspondent expanded types are equal in all exherited classes.

Rule ***Exheriting Separate Types.*** Features having separate types can be exherited if the correspondent separate types are equal in all exherited classes.

4.5.3 Exheriting Like Types

Anchored types (or like types) were introduced in Eiffel in order to avoid the covariant redeclaration of the inherited features. In this section we analyze how features which are anchored can be exherited. Therefore we consider the following cases regarding anchored features: features which are anchored to other features, features which are anchored to **current** and features which are anchored to arguments. When a feature in a class is anchored to **current** it means that it is anchored to the local class type.

Exheriting Features Anchored to "Current"

In the example 66 we consider the case of features anchored to the special anchor *current*. Such an anchor refers to the local class in which it is written. For example feature f of class B is of type B, while feature f of class C is of type C.

Exheriting such a feature like f in class A means that it will be of type A. This behavior obeys the conformance rules of reverse inheritance. Type A is a supertype of types B and C.

Rule ***Exheritance for Anchored Feature***. The features anchored to **current** in the subclasses can be exherited keeping the same anchored type.

Exheriting Features Anchored to Other Features

In this case we deal with features anchored to other features, like those from the next example:

In example 66, the anchors are attributes. We can set the following rule:

Example 66 Exheriting Anchored Features (1)

```
class B
feature
   f: like current
end
class C
feature
   f: like current
end
class A exherit
 B
 C
 all
end
```

Example 67 Exheriting Anchored Features (2)

```
class B
feature
  b: T1
  f: like b
end
class C
feature
  c: T2
  f: like c
end
foster class A exherit
 B
 C
 all
end
```

Example 68 Exheriting Anchored Features (3)

```
class B
feature
  f(p: T1; p2: like p): like p is do ... end
end
class C
feature
  f(p: T2; p2: like p): like p is do ... end
end
class A exherit
 B
  redefine att
 end
 C
  redefine att
 end
 all
 feature
  f(p: T; p2: like p): like p is do ... end
end
```

Rule ***Exheritance for Anchored Feature***. Let us assume that b is defined in class B (resp. c is defined in class C) and that it is of type T1 (resp. T2)

- if *T1* and *T2* are equal, the feature f in both subclasses has the same signature and is automatically exherited;
- if *T1* is the supertype of *T2* (resp. *T2* is the supertype of *T1*), in class A feature f can be exherited with type *T1* (resp. *T2*).
- if *T1* and *T2* have some common supertype T then feature f can be exherited with type T.
- if *T1* and *T2* are not related by any relations, feature f can be exherited only with type *ANY* (this kind of exheritance will not help too much in practice).

Exheriting Features Having Arguments Anchored to Other Arguments

In this case we consider not just method arguments but also return types. In example 68, we analyze a feature having an argument and the return type. From it we can draw the conclusion that the second parameter and the return type will always follow the type of the first argument. Feature f can be exherited taking into account the rules defined for type adaptations in section 5.2. In the exheritance of feature f the only thing that counts is the relation between types *T1* and *T2*.

Rule ***Exheritance for Anchored Feature***. The features anchored to other arguments in the subclasses can be exherited respecting the signature adaptation rules for the types the features are anchored to.

Exheriting Features Having Arguments Anchored to Features

In this section we present the case of exheriting features which have arguments anchored to features from the class. In ordinary inheritance when such a situation arises both the referred feature and the feature using anchors are inherited in the subclasses automatically. This way the feature using anchors will have always available the referred feature. In reverse inheritance this dependence is

Example 69 Exheriting Anchored Features (4)

```
class B
feature
  att: T1
  f(p: like att): like att is do ... end
end
class C
feature
  att: T2
  f(p: like att): like att is do ... end
end
class A exherit
 B
  redefine att
 end
 C
  redefine att
 end
 all
 feature
  att: T
  -- f(p: like att): like att is deferred end -- is implicit
end
```

not assured implicitly because someone might want to exherit the feature using anchors, but not the referenced feature. In this situation reverse inheritance in invalid, so it must be restricted by rules. One possibility is to use the anchored feature target type, which might be a recursive process.

In example 69 we present a simple example of a feature f having anchored types for the argument and for the return type. Both anchors refer to attribute *att* in both subclasses B and C. Feature f is exheritable but it will be valid in the foster class only when feature *att* will be exherited too, since feature f uses *att*.

Rule ***Exheritance for Anchored Feature***. The features anchored to other features in subclasses can be exherited if the other features are also exherited.

4.5.4 Exheriting Bit Types

In this section we will present the exheritance of features having bit types in their signature. In Eiffel the bit types may refer an integer manifest constant or an integer constant feature. In example 70 we will show how the bit type can be exherited from the exherited classes in all the possible combinations.

The *f1* features from the exherited classes have the bit type expressed using the same value integer manifest constant and they are exherited using the same type reference.

In exheriting the bit types referring exheritable constant features we have two choices:

- either we exherit the type and the referred feature also and we set the link between the type and the exherited feature at foster class level;
- or we exherit the type and we create a new manifest constant having the same values as the ones referred in the exherited classes.

The first choice would seem more natural to perform but it has the drawback that the language does not allow constant features to be redefined. Still what we could do is to move up automatically

Example 70 Exheriting Bit Types

```
class EC1
feature
 b:INTEGER is 7
 f1(x:bit 7):bit 7 is do end
 f2(x:bit b):bit b is do end
 f3(x:bit 7):bit 7 is do end
end
class EC2
feature
 b:INTEGER is 7
 f1(x:bit 7):bit 7 is do end
 f2(x:bit b):bit b is do end
 f3(x:bit b):bit b is do end
end
foster class FC
 exherit
  EC1
  EC2
  all
feature
 -- f1(x:bit 7):bit 7 is deferred end -- is implicit
 -- f2(x:bit 7):bit 7 is deferred end -- is implicit
 -- f3(x:bit 7):bit 7 is deferred end -- is implicit
end
```

the feature from the exherited classes, but this would be in contradiction with the principle of exheriting implicitly all features as deferred. The second solution is more feasible from the technical point of view, but does not keep the same philosophy of the exherited class.

In our example *f2* features refer a constant feature *b* which has the same value 7, being present in each exherited class. In this case we exherit a feature whose types are linked to a manifest constant. In case of *f3* features we mixed the two natures of the types and the implicit result is that the types used in the superclass are linked to manifest constants. This behavior is normal since not all features are linked to some exheritable constant feature.

Rule ***Exheriting Bit Types*** In order to exherit bit types, they must refer the same value and they will be exherited in the foster class as referring an integer manifest constant.

4.5.5 Exheriting Various Types

In example 71 we mixed several types: class types referring formal generics and like types. These types are special because they do not point directly to the target type. In order to perform exheritance on such types we must consider their target types. If they are identical then exheritance is possible. Feature *f1* is of type *T* in both exherited classes, it is normally exherited and the common type is *T*. Feature *f2* is like *f1* in one exherited class and of type *T* in the other exherited class. Since the two types refer the same target type *T*, exheritance is possible and the exherited type will be obviously *T*. Feature *f3* is a bit more complex. The base class *A* is the same in both exherited classes while the actual argument is in the situation of feature *f2*. Because we showed that *f2* is exheritable, also *f3* is exheritable (same base generic class and same target type of actual generics). Feature *f4* adds the expanded keyword to the class type having an actual generic.

Example 71 Exheriting Various Types

```
class EC1
feature
 f1:T
 f2:like f1
 f3:A[like f1]
 f4:expanded A[like f1]
end
class EC2
 f1:T
 f2:T
 f3:A[T]
 f4:expanded A[T]
end
class T ... end
class A[G] ... end
foster class FC
exherit
 EC1
 EC2
 all
feature
 -- f1:T
 -- f2:T
 -- f3:A[T]
 -- f4:expanded A[T]
end
```

4.6 Behavior in the New Created Class

In order to preserve the behavior of the subclasses and to keep the symmetry between the two complementary class inheritance relationships it is not allowed to add new behavior in the superclass. Otherwise, if some new behavior was specified in the superclass, it would be inherited by all the subclasses, thus changing the semantics of the original ones.

In some special conditions, the target class of reverse inheritance can have superclasses. As we mentioned earlier the behavior of the source class must not be modified, thus the content of the superclass must be restricted. It can contain only a subset of the exherited features preserving their signatures and some method implementations can be provided within this class.

Another interesting case could be when a foster class has two superclasses, meaning that it is the target of a multiple inheritance class relationship. In this case the same rules of preserving the behavior of exherited classes apply as in single inheritance. The features in both superclasses have to be included in the set of the exherited features.

As a conclusion we can state that any hierarchical structure is allowed to be on the top of one foster class as long as the behavior of the subclasses is not affected by the inheritance of new features.

Rule ***New Features in Foster Class*** Features to a foster class can be added only if they will not change the original behavior of the subclasses. Behavior can be attached only to a feature in the foster class (directly or by normal inheritance, single or multiple) if that feature can be exherited.

This rule will set the base for determining when a foster class can have superclasses:

Rule ***Foster Superclass*** A foster class can have superclasses only if the inherited features match the exherited features.

4.7 Use of Exheritance Clauses for Factoring Features

In this section we will discuss the impact of the exheritance clauses like **redefine, undefine** and **moveup** in the context of exherited features. The **redefine** and **undefine** clauses are already part of the Eiffel language, but their semantics has to be clarified in the context of reverse inheritance.

Redefinition of a feature in a subclass consists in changing the signature, the specification or the implementation. Of course, in ordinary inheritance the redefined signature must conform to the original one. With reverse inheritance since we redefine the feature in the foster class, all the feature signatures from subclasses must conform to the redefined one in the superclass. By conformance we mean the characteristic of a type to be reused instead of another [Mey02]. The redefinition of an attribute from the superclass as a method in the subclass is not possible. From this restriction we infer that a set of features from the exherited classes can be exherited as a concrete attribute if all features in the set are concrete attributes.

Undefinition of a feature in the context of ordinary inheritance means that the undefined feature becomes deferred in the subclass. This adaptation can be applied to methods, but not to attributes. In the context of reverse inheritance it is natural to exherit a feature no matter if is a method or an attribute as a deferred feature as long as the signatures are compatible. Still if there is a deferred feature in one of the exherited classes, the corresponding feature in the foster class cannot be attribute.

The **moveup** clause is newly added to the language and its semantics is related to the selection of implementation from the subclasses. If in the subclasses there is an implementation for a feature which is suitable in the foster class, the **moveup** keyword should be used on the exheritance branch corresponding to the subclass that has the desired implementation. Of course, there should be only one implementation selected for a feature. It is acceptable to use **moveup** on multiple branches if the implementations in the corresponding subclasses are the same for a specific feature.

The **adapt** clause is used for performing special kind of adaptations that cannot be performed with **undefine** and **redefine** clauses. Because exherited features came from different subclasses, which may belong to different class hierarchies, some special adaptations seem to be necessary. In chapter 5 are presented details on these issues.

To have a first idea of possible combinations for the exheritance clauses we make the following remarks:

- The **rename** clause can be combined freely with the clauses **undefine** (implicit behavior), **redefine**, **adapt** and **moveup**. When combining renaming with other exheritance clauses, renaming has to be performed first, and only then the other desired clauses are used. Using the renaming clause, the exherited feature will acquire a new name and the new name will be used in the next desired exheritance clauses.

- The **adapt** clause, being used only for the adaptations that cannot be performed with **undefine** and **redefine** clauses, it cannot be used in combination with the two keywords. It is not possible to redefine and adapt a feature at the same time since the clauses refer to a disjunctive set of adaptations: the former - to classic Eiffel adaptations and the latter - to special adaptations. It does not make sense to undefine a feature and to adapt it at the same time, since there will be no implementation available. The combination of **adapt** and **moveup** is not permitted since it affects the clarity of code.

- The order used in the exheritance branch for **rename**, **undefine** (which is implicit), **redefine** and **select** is based on the one used for the clauses existing already in Eiffel.

  - After a **renaming** clause the new name of the feature has to be used in the potential undefinition, redefinition, adaptation, moving or selection. Like in ordinary Eiffel, renaming deals with the name of the feature and not with the feature itself. The rest of the operations redefine, adapt, undefine, select, moveup refer to modifications related to signature, specification, implementation.
  - **Undefine** is the default implicit behavior and is placed after renaming and before selection clauses, but is not allowed together with adapt since the two are not compatible.
  - **Adapt** is a special kind of redefinition, so both can be treated using the same priority order: after implicit undefinition and before selection. An adapted feature is prohibited to be moved up.
  - When **moving up** a feature implementation from the subclass in the foster class the eventual renaming operation should be considered only. Since the implementation is exherited, it cannot be undefined. The redefinition of a moved feature is necessary if its signature is changed in the foster class.

4.8 Summary

In this chapter we presented basic concepts about how a foster class can be created using reverse inheritance. It was pointed out that the most simple class configuration when using reverse inheritance is the single reverse inheritance case. Also we have to notice that class configurations which look like multiple inheritance in context of ordinary inheritance, are just some independent reverse inheritance class relationships. What is important in such class configurations is not so much the shape of the diagram, but the order of class creation, namely, the time stamp of each class.

The implicit and natural semantics of factoring features was presented. When attributes and methods have the same signatures in subclasses they can be automatically exherited. Implicitly, the attributes are exherited without their implementation, meaning that they are deferred in the foster class. For methods we use the same rule, the signature is exherited only, being deferred in the foster class. For practical reasons in the semantics of reverse inheritance is allowed to choose

between the implicit and explicit selection of the exherited features. There are four options: to implicitly factor all possible features, having the possibility of explicitly excluding some common features, to let the programmer specify explicitly the features needed in the foster class and to factor no features at all.

The nature of the features is taken also into account. We analyzed cases where attributes and methods with the same signature were present. In that case they will be factored as an abstract feature in the foster class. The same rule is set if methods are factored from the subclasses and some are concrete and some deferred. When implementation is decided to be factored the problem of dependencies arises.

For symmetry reasons and for keeping the philosophy of the Eiffel language consistent, reverse inheritance is designed as a dual class relationship, having two forms: a conforming one and a non-conforming one. The syntax used at this point is the same as the one used for ordinary inheritance. Another supposition for keeping the semantics of the language intact is to not allow the behavior in the foster class influence the behavior of the subclasses.

Type exheritance refers to the rules showing how types can be exherited. Types represented by generic classes instantiated with actuals behave like composed types because the type exheritance rules must be applied recursively. Expanded and separate types depend on class types so the rules for class types have to be applied along with the appearance of **expanded** respectively **separate** keywords. The like types may be exherited with their link to the anchor if the anchor is exheritable too, if not, the type in the foster class is the type of the anchor. Bit types are very special, they cannot be compared with any other type. If their size expressed by a manifest constant or a constant feature is equal then the bit type may be exherited. Generic types may be instantiated with any type from the Eiffel type system, like types seem to point to any other types, in the same manner, so before exheriting them, they must be evaluated.

Exheritance clauses used in factoring features rise several cases, which were semantically analysed and rules were stated about them. It was taken into account the undefine, moveup, redefine exheritance clauses and the nature of the features attributes and methods. Some combinations of exheritance clauses are invalid, some other combinations are valid under certain conditions and some cases are always perfectly valid. The analysis was made on two exherited classes, but the same reasoning can be applied when working with multiple ones.

Chapter 5

Adaptation of Exherited Features

[LHQ94] presents a set of adaptations which are valid in the definition of the reverse inheritance semantics. However, we think that there are some points where the adaptation mechanism proposed by [LHQ94] can be extended. Adaptations can be seen as local transformations applied to candidate features from subclasses in order to make them conform to a common signature. These feature adaptations are performed before their factorization, such that a feature with different signatures satisfies the constraints for being exherited.

In this chapter we will show what kind of adaptations can be performed and what are their restrictions. Each adaptation provides a mapping between the original signature of a feature and the common signature located in the foster class.

5.1 Adaptations for Ordinary Inheritance Applied to Reverse Inheritance

5.1.1 Feature Redefinition

The semantics of feature redefinition in reverse inheritance is intended to be kept the same as in ordinary inheritance. Feature redefinition in ordinary inheritance implies changing either **signature**, **specification** or **implementation**. In the context of reverse inheritance, redefinition can be used to adapt the signatures of certain features to one common signature. There are some limitations of the changes that can be performed.

Signatures from subclasses will have to conform to the signature of the superclass no matter if they are linked by an ordinary or reverse inheritance class relationships. Specification redefinition will obey (in particular) the rules presented in section 5.4. Implementation redefinition will take into account the rules in subsection 4.3.4 and section 4.6, because an implementation can be rewritten or exherited. If the implementation is removed, meaning that the feature is deferred, the undefine clause is used implicitly.

An example of such a feature redefinition is provided in example 72. Feature f from class A must be redefined because the signature and the behavior in class C is changed. Feature f from class B has only a new implementation preserving the original signature.

In the context of ordinary inheritance there are restrictions about redefining an attribute from the ancestor as a method in the heir. This is due to the fact that a method in the heir containing an assignment to that attribute could be inherited, so the redefinition of the attribute into a method would invalidate the inherited assignment. For this reason, in reverse inheritance a feature from the exherited class can always be redefined as a method in the foster class, but it can be redefined as an attribute only if it is an attribute in all exherited classes.

Rule ***Feature Redefinition***. When a feature f is exherited and when a change of signature, specification (assertions) or implementation is necessary, then feature f has to be redefined and must satisfy the following conditions:

Example 72 Feature Redefinition

```
class T
end
class T1 inherit T
end
class A
feature
  f(x: T1) is do end -- implementation of class A
end
class B
feature
  f(x:T) is do end -- implementation of class B
end
class C exherit
 A
  redefine f
 end
 B
  redefine f
 end
 all
feature
  f(x: T) is do end -- implementation of foster class C
end
```

- all the signatures of f specified in the subclasses must conform to the new signature of f in the foster class;
- specification adaptation must agree with the rules defined in section 5.4;
- implementation adaptation will conform to the rules related to implementation exheritance (subsection 4.3.4) and to the rules related to behavior in the newly created class (section 4.6);
- an exherited feature can be redefined as an attribute if it is an attribute in all exherited classes.

As an observation, it can be noticed that a supertype for a set of classes, needed in a covariant redeclaration, always can be obtained by reverse inheritance.

5.1.2 Feature Undefinition

In ordinary inheritance if an effective feature is undefined then it becomes deferred in the subclass (to be noted that the current rules of Eiffel do not allow the undefinition of attributes). With reverse inheritance an exherited feature is undefined implicitly in all exheritance branches, this means that it will be deferred in the superclass. Of course, this works for any kind of features: attribute and method with the constraint that signatures are covariant or at least adaptable. Some of these aspects were also discussed in section 4.3.1.

Rule ***Deferred Feature in Foster Class***. An exherited feature is implicitly deferred in the foster class and is undefined in all exheritance branches. In this case the foster class becomes deferred.

Example 73 Feature Renaming

```
class BOX
feature
 boundary: REAL
end
class CIRCLE
feature
 circumference: REAL
end
deferred foster class SHAPE exherit
  BOX
   rename
    boundary as perimeter
  end
  CIRCLE
   rename
    circumference as perimeter
  end
  all
feature
  perimeter: REAL
end
```

Rule ***Orthogonality According to Features***. Attributes and methods (**once**[1] or not) are undefined when they are exherited.

5.1.3 Feature Renaming

Because quite often classes are developed independently, in most cases it happens that common features have different names so that a name adaptation is required. In example 73 taken from [LHQ94] we present such a name conflict situation and we propose a syntax extension for solving it.

The classes *BOX* and *CIRCLE* both have attributes which refer to the perimeter of the shape, but using different names *boundary* and *circumference*. Because we want to factorize those features in class *SHAPE* we had to rename the two features using a common name such as *perimeter*.

It may also occur that two features from different subclasses have the same name but represent different features. This is called the case of "false friends", as presented in [Sak02], and it is not discussed in [LHQ94]. Obviously renaming is used to stop the ambiguity and there is no possibility for an automated approach. The syntax used for renaming is the same like the one proposed for ordinary inheritance in Eiffel.

Rule ***Renaming Feature***. Renaming should be used when several semantically equivalent features do not have the same name or when features with different semantics have the same name.

5.1.4 Conclusions

After all these examples we can draw the conclusions regarding the semantics of the exheritance clauses in two contexts: ordinary inheritance and reverse inheritance. The conclusions are centralized in table 5.1.

[1] In Eiffel once methods are executed when they first called and the result is stored. All other calls to once methods will return the stored result, being no longer executed.

Inheritance/ Exheritance Clauses	In context of ordinary inheritance	In context of reverse inheritance
rename	to change the name of the feature	to change the name of the feature
undefine	to make a feature deferred (it can not be used for attributes, but only for methods)	to make a feature deferred (it is used implicitly for both attributes and methods)
redefine	to change signature (in a covariant way), specification or implementation of a method	to change signature (in a covariant way), specification or implementation of a method
adapt	no semantics attached	to change the signature (in other ways than redefine does, and will be studied thoroughly in chapter 5)
select	is used for multiply inherited features with a common seed when a certain feature needs to be selected in a polymorphic call	has a special semantics dedicated to solve dynamic binding problems and it will be presented in detail is section 6.2
moveup	no semantics attached	used to select an implementation of a method in the foster class from one of the exherited classes (of course if the dependencies allow this, see chapter 6)

Table 5.1: Semantics of Inheritance and Exheritance Clauses

The renaming mechanism has the same semantics in the two class relationships. The undefinition mechanism has the same semantics but in ordinary inheritance must be explicitly invoked, while in reverse inheritance is applied implicitly to set the status for the exherited features. The redefinition mechanism has the same known semantics for both ordinary and reverse inheritance. The adaptation mechanism is valid only in the context of reverse inheritance and it is meant for adaptations which are not achievable by redefinition. The feature selection mechanism to solve dynamic binding conflicts has the same semantics in both class relationships, but different syntax, since the feature selection decision is taken late, after the creation of the root class. The moveup mechanism enables the explicit selection of the implementation from one subclass for an exherited feature. The counterpart mechanism in ordinary inheritance is the inheritance of features from the superclass to the subclasses which is implicit. So the moveup mechanism in the context of ordinary inheritance has no semantics attached.

5.2 Special Signature and Value Adaptations

In order to increase the expressiveness of the reverse inheritance relationship, and implicitly class reusability, it may be interesting to provide a mechanism allowing to customize the signature of the feature when exheriting. The following rule provides a general framework for such adaptation.

Rule ***Special Feature Adaptation***. The **adapt** clause allows to specify the name of the method whose signature should be adapted in order to make possible the factorization of the corresponding method in the various subclasses. All adaptations are put after the keyword **adapted** which is located just after a possible method precondition. Each statement declared in this area allows to specify the adaptation to perform depending on the subclass which is considered. The name of the class is put between braces, following the same syntax as the one used for specifying the class corresponding to the **precursor**. If no adaptation is specified for a subclass of the foster class, then no adaptation will be performed except

Example 74 Adaptation Grammar Rules

```
Adapted_opt: /* empty */
        | E_ADAPTED Adapted_list E_END
Adapted_list: Adapted_item
        | Adapted_list Adapted_item
        | Adapted_list ';' Adapted_item
Adapted_item: '{' Class_type_list '}' Attribute_adaptation
        | '{' Class_type_list '}' Routine_adaptation
Class_type_list: Class_type
        | Class_type_list ',' Class_type
Attribute_adaptation:
        Adapted_type E_IS '(' Expression ')' Adapted_result
-- Expression is used in assignments to the attribute, and may contain
-- 'Precursor'. Adapted_result is used for reading the attribute, and may
-- contain 'Result'.
Adapted_type: Type
        | E_LIKE E_PRECURSOR
Adapted_result: ':' Expression
-- May contain 'Result'.
Routine_adaptation:
        Adapted_formals Adapted_type_mark_opt E_IS
        Adapted_actuals Adapted_result_opt
        | Adapted_type E_IS Adapted_result
Adapted_formals: '(' Entity_declaration_list ')'
        | '(' E_LIKE E_PRECURSOR ')'
Adapted_type_mark_opt: Type_mark_opt
        | ':' E_LIKE E_PRECURSOR
Adapted_actuals: '(' Actual_list ')'
        | '(' E_PRECURSOR ')'
-- The expressions in Actual_list may contain names of formal arguments
-- of the foster class routine.
Adapted_result_opt: /* empty */
        | Adapted_result
```

if one adaptation had been defined for one of its ancestors, if any. The **adapted** keyword allows to specify the adaptation to perform depending on the considered subclass. Possible adaptations are scale adaptation, modification of the parameter order, number, or type, modification of the type of feature (attribute type or function return type).

The adaptations applied to attributes are valid only if they are expanded and their instances are treated as values. For non-expanded classes things become more complicated and are not discussed here.

The grammar rules are listed in example 74.

5.2.1 Scale Adaptation

The idea of providing scale adaptation can be found in [SN88]. This mechanism is used to facilitate the conversion between value scales (they use conversion methods between the scales which are encapsulated in a special meta-class). In our approach we will use the conversion formulas as an adaptation technique because it seems to fit better in the programming language philosophy.

In example 75 we illustrate a possible use of the scale adaptation mechanism. We start from two classes *RECTANGLE* and *ELLIPSE* which have two methods returning the area of each

Example 75 Scale Adaptation (1)

```
class RECTANGLE
feature
 getSurface: INTEGER is
 do
  -- rectangle specific implementation in cm^2
 end
end
class ELLIPSE
feature
 get_Surface: INTEGER is
 do
  -- ellipse specific implementation in m^2
 end
end
foster class SHAPE exherit
 RECTANGLE
  rename
   getSurface as getArea
 end
 ELLIPSE
  rename
   get_Surface as getArea
  adapt
   getArea
 end
 all
feature
 getArea: INTEGER is
  adapted
   {ELLIPSE} : like precursor is : Result * 10000
 end
end
```

shape, but using different scales for that. The method in class *RECTANGLE* returns a value in cm^2 while the corresponding one in class *ELLIPSE* returns a value in m^2. Following the same homogeneity principle announced in section 72 we definitely need a conversion between the two scales. So we decided to extend the syntax of Eiffel to be able to specify the desired transformation. Although in the example we can remark that the two techniques of renaming and adapting are orthogonal since they do not affect each other.

The like precursor used in the adaptation denotes the new type returned by the method. The expression after the **is** keyword represents the adapted returned value of that method. Internally the method from the exherited class uses its own representation and it must be adapted when it is returned.

In the case of adapting an attribute, conversions should be provided in both ways. This is necessary when setting the value of an attribute. In our approach we consider that a natural way to do this is to adapt the assigner method of the attribute. This is illustrated in example 76. In order to exherit the *surface* attribute we need to also exherit its assigner method *putSurface*. In this example we can see that both features (attribute and its assigner) are associated to a piece of code which achieves the value conversion. When the attribute *surface* is evaluated using a reference of type *SHAPE* on an instance of *ELLIPSE*, then its implementation is transformed to have the same representation as in the superclass. When an external object assigns a value to this attribute through the assigner method attached to a *SHAPE* reference a conversion is also necessary. Class *ELLIPSE* works using its own representation and when its instances interact through *SHAPE* references the conversion code is put to work. When an attribute has no assigner method then no scale adaptation can be performed on it.

Rule ***Function Scale Adaptation***. Scale adaptation can be applied to methods returning values and it implies providing a conversion from the returned type scale of the feature in the subclass to the scale of the corresponding feature in the superclass.

Rule ***Attribute Scale Adaptation***. Scale adaptation can be applied to attributes and it implies providing conversions in both ways: in one way like in the rule above and in the other way by adapting the assigner methods of the attribute.

5.2.2 Parameter Order Adaptation

The issue of parameter order adaptation is discussed in [LHQ94]. They proposed a syntax extension to solve this problem[2](see example 77):

The proposed syntax extension relies on the *adapt* clause similar to scale adaptation (see section 5.2.1). It provides a new parameter mapping related to the original one. The relation between the parameters remains the same that is to say, one to one. No parameter is omitted or duplicated. The syntax extension allows to specify that when a call to the method *scale* is made (through a reference of type *SHAPE*) on an instance of class *BOX*, it is equivalent to performing a call to the method *zoom* with the parameters in the reverse order.

Example 78 (derived from example 77) illustrates a situation where the adaptation of the parameter order is useful. When the *scale* method is called then everything works as if it was the *zoom* method which is called relying on both adaptation mechanism and dynamic binding semantics[3].

5.2.3 Parameter Number Adaptation

A similar though more complicated situation arises when the number of parameters is not the same. In such a situation there are several possibilities:

[2]We slightly adapted the syntax to fit better to our approach. The changes made are only at the syntactical level and the semantics is preserved.

[3]Issues dealing with the dynamic binding will be addressed with much more details in chapter 6.

Example 76 Scale Adaptation (2)

```
class RECTANGLE
    -- Rectangle specific implementation in cm^2
feature
 surface: INTEGER assign putSurface

 putSurface(p: INTEGER) is do surface := p end
end
class ELLIPSE
    -- Ellipse specific implementation in m^2
feature
 area: INTEGER assign putArea
 putArea(a: INTEGER) is do area := a end
end
foster class SHAPE exherit
 RECTANGLE
 ELLIPSE
  rename
   area as surface
   putArea as putSurface
  adapt
   putSurface
 end
 all
feature
surface:INTEGER is
  adapted
   {ELLIPSE} like precursor is (precursor / 10000) : Result * 10000
  end
putSurface(s:INTEGER) is
  adapted
   {ELLIPSE} (a:INTEGER) is : (s / 10000)
  deferred
  end
end
```

Example 77 Parameter Position Adaptation

```
foster class SHAPE exherit
 BOX
  rename
   zoom as scale
  adapt
   scale
 end
 all
feature
 scale(factor:REAL;center:POINT) is
  adapted
  {BOX} (center:POINT;factor:REAL) is (center,factor)
 end
end
```

Example 78 Using the Adaptation

```
point : POINT
factor: REAL
s: SHAPE
b: BOX
create b
s := b
s.scale(factor,point)
-- equivalent call: b.zoom(point,factor)
```

- To omit or ignore some parameters - this mapping can be used when in the context of a given class, the parameter is not needed because of possible lack of semantics.
- To freeze some parameters to constant values - this technique goes in the direction of languages which support function overloading like C++ does. The restriction imposed by C++ is to locate those default parameters at the end of the declaration list. In our case the proposed syntax bypasses this restriction and allows omitting any parameters independently of their relative position in the list.
- To replicate some parameters - this practice can be used when the method in the subclass has a more general behavior and in order to obtain a particular behavior some parameters can be duplicated. It is intended that the behavior of the superclass will be reused in the subclass in a particular context.
- To rely on the description of basic computations in order to create new parameters. This means to write an expression which yields a result which will be used as a parameter later.

In example 79 all the cases are included and a syntax is proposed to describe them.

In example 79 we have designed a superclass X by reverse inheritance, which has a method m with two parameters. We intended to show that it is possible to use something else than a one to one parameter mapping. The computations involved into parameter adaptations should be basic and may involve only the method parameter and attribute or function of the foster class.

The mechanism presented does not interfere with the other adaptation clauses presented in previous sections, they are orthogonal so they can be freely combined. For instance, it is possible to perform a scale adaptation in the same **adapted** construct, thus making the adaptation expressions more complex. Moreover the feature (m in our example) could have a redefined signature, a new specification, a new written body or an empty body. In this case, the construct **adapted** is placed at the head of method declaration (after possible preconditions), before **deferred** keyword or respectively **do** keyword.

Rule ***Parameter Number Adaptation*** Parameter number adaptations are necessary when exherited methods have a different number of parameters than the method in the foster class. In the adaptation clauses of the foster class method there can be written equivalent calls to subclass methods in which some parameters are omitted, frozen, replicated or computed. The syntax used for this adaptation is the one presented at the beginning of this section (5.2).

5.3 Generic Type Adaptation

In Eiffel, the genericity can be constrained or unconstrained. Unconstrained genericity implies that the generic parameter of a generic class can represent any arbitrary type. If the type is constrained to a specific one then it is possible to do specific operations with it in the class because it conforms to a known interface. From this dual point of view we start the genericity

Example 79 Parameter Number Adaptation

```
class A
feature
  m(p1:INTEGER) is do  end
end
class B
feature
 m(p1, p2, p3: INTEGER) is do  end
end
class C
feature
 m(p1, p2, p3:INTEGER) is do  end
end
class D
feature
 m(p1, p2, p3: INTEGER) is do  end
end
foster class X exherit
 A
  adapt m
 end
 B
  adapt m
 end
 C
  adapt m
 end
 D
  adapt m
 end
 all
feature
 m(q1, q2: INTEGER) is
  adapted
   {A} (p1:INTEGER) is (q1)
   {B} (p1,p2,p3:INTEGER) is (q1, q2, 0)
   {C} (p1,p2,p3:INTEGER) is (q1, q2, q1)
   {D} (p1,p2,p3:INTEGER) is (q1, q2, q1 + q2)
  do
  ... -- possible implementation
 end
end
```

Example 80 Unconstrained Genericity (1)

```
class A[G1]
feature
  e: INTEGER
  f: G1
end
class B[G2]
feature
  e: INTEGER
  f: G2
end
foster class C exherit
 A
 B
 all
feature
  -- e: INTEGER is deferred end --is implicit
end
```

impact analysis on reverse inheritance. We have to specify also that a class can depend on more than one generic parameters, actually it is possible for a class to have a list of formal generic parameters. Our attention is focused on features having generic types potentially to be exherited and also on the relationship between foster class and exherited class. It is known that a generic class must generate[4] a concrete class through the process of formal generics instantiation. In inheritance the subclass refers an already instantiated generic superclass. Next, we analyse if the same thing may happen in the context of reverse inheritance: if the foster class will instantiate the generic exherited classes in the process of exheritance.

5.3.1 Unconstrained Genericity

Let us have two classes A and B which exist initially and a superclass C created using a reverse inheritance class relationship with A and B. We take the case of two classes specifying one generic parameter, which seems to be general enough. At first glance we can identify three main cases which are addressed in the next three subsections.

Non-generic Foster Class and Generic Subclasses

Class C has no generic parameters, while classes *A[G1]* and *B[G2]* have. In this case, only the features which do not involve generic types are subject to exheritance.

In example 80, feature e from A and B subclasses will be automatically exherited while f features cannot be exherited since they are of different generic types. The equivalent class configuration built with ordinary inheritance will have generic subclasses which inherit from a non-generic superclass.

If subclasses were exherited as instantiated with a concrete type *DOUBLE* for example, then feature f would seem exheritable since it is of type DOUBLE in both subclasses through class instantiation. On the other hand the type of the exherited feature in the superclass would not be available in the equivalent class hierarchy, so the exheritance of generic features must be forbidden in this case.

One important aspect worth to discuss here is how the exheritance is performed. As it is presented in example 80 we exherited directly the generic subclasses, without any class generation.

[4] In Eiffel, the term of class generation (not instantiation) is used in the context of generic classes, in order to avoid the confusion with the concept of class instantiation which generates objects.

Example 81 Unconstrained Genericity (2)

```
class A[G1]
feature
  f: G1
  m(p: G1) is do ... end
end
class B[G2]
feature
  f: G2
  m(p: G2) is do ... end
end
foster class C[G3] exherit
 A[G3]
 B[G3]
 all
feature
  -- f: G3 is deferred end -- is implicit
  -- m(p: G3) is deferred end -- is implicit
end
```

We could also exherit features from two generated classes, but the generation types would be lost in the equivalent ordinary inheritance class hierarchy. Another possibility is to extend the syntax and to exherit the subclasses instantiated with class *NONE* argument: *foster class C exherit A[NONE] B[NONE]*. Thus, there is no concrete instantiation type[5] but it is shown that the referred class is generic. The advantages of such a syntax extension are more visible in the next section. In conclusion, we can declare that this case is valid and rules are issued to handle such a case.

Rule ***Exheriting Generic Classes with a Non-generic Foster Class***. When subclasses are generic and unconstrained and the foster class is not generic, then it is possible to exherit the generic classes, not the generated ones and to factor only features which are non-generic.

Generic Foster Class and Generic Subclasses

Classes *A[G1]*, *B[G2]*, *C[G3]* are all generic classes. In this case the class hierarchy based on reverse inheritance looks like in example 81. Like in example 80, it is possible to exherit non-generic features as far as they satisfy the signature exheritance requirements mentioned in earlier sections. Because there is no constraint on the generic arguments it is possible to exherit features whose signatures are composed out of formal generic types. In order to obtain a consistent equivalent ordinary inheritance class hierarchy, each formal generic of the subclasses must be instantiated with the same formal generic from the foster class. Thus, in the equivalent ordinary inheritance class hierarchy the subclasses will instantiate the superclass with the correspondent formal generics. So, each formal argument from the foster class will have a correspondent in each subclass.

If the foster class has more formal generics than the subclasses then it would be impossible to build an equivalent class hierarchy because in ordinary inheritance some superclass formal arguments will have no correspondents in the subclasses. Also, if the foster class exherits generated subclasses then the generation types will be lost and no equivalent hierarchy can be generated because the types for the superclass generation are inexistent. If the foster class has less formal arguments than the subclasses then it is possible to instantiate some subclass formal arguments with type *NONE* as in the previous case. In this situation the instantiation with *NONE* is more useful since it increases the possibility of exheriting subclasses with different number of formal

[5]Class *NONE* is a conceptual class in Eiffel that inherits all other classes of the universe and it is used for export restriction, definition of non-conforming inheritance.

Example 82 Unconstrained Genericity (3)

```
class A
feature
  f: T1
  m(p: T1) is do ... end
end
class B
feature
  f: T2
  m(p: T2) is do ... end
end
foster class C[G] exherit
 A
  adapt f,m
 end
 B
  adapt f,m
 end
 all
feature
  f: G
  m(p: G) is adapted end
end
```

generics. Such a syntax can be viewed as a deviation from the symmetry of the language, but it will increase the reuse power of reverse inheritance. From this point of view we can consider that example 80 is a sub-case of example 81.

The order of generic parameters in exheritance or inheritance is not important, the only requirement is the formal arguments correspondence.

Rule ***Exheriting Generic Subclasses with a Generic Foster Class***. When subclasses are generic and unconstrained and the foster is also generic and unconstrained, then it is possible to exherit normally non-generic features and also generic features only if for each formal generic of the foster class there is a correspondent formal generic in each subclass.

Generic Foster Class and Non-generic Subclasses

Superclass *C[G]* has generic parameters while subclasses *A* and *B* do not have.

In some cases like in example 82, it may be interesting to build a foster class which is not only more abstract but also can implement, using genericity, a part of the behavior independently from the data involved. This case is particularly useful for data structures like those from example 83). In our example we created the class *C[G]* with a formal generic parameter *G* which will represent the concrete types *T1* and *T2* from classes *A* and *B*. In this way it is possible to create at least generic signatures at the level of the superclass. In some cases it could be useful to take an implementation from one subclass and to generalize it in the superclass using the generic types. Of course, this is possible only when the implementation of the exherited method is prepared to allow a concrete type substitution with a generic one. Reverse inheritance presents class parameterization capabilities but in general situations some other preparations are still required [CRM99].

As a conclusion in this case, multiple reverse inheritance requires the generic parameter *G* to be a supertype of *T1* and *T2*. This requirement could be better handled in the case of constrained genericity. It seems that this case is more appropriate to single reverse inheritance situations.

From the class relationship point of view we can say that instantiation between foster class and exherited classes is not necessary since exherited classes are not generic, while in equivalent

Example 83 Unconstrained Genericity (4)

```
class PERSON_COLLECTION
feature
  add(e: PERSON) is do ... end
end
foster class COLLECTION[G] exherit
 PERSON_COLLECTION
  adapt add
 end
 all
feature
 add(e: G) is adapted ... end  -- the parameterized implementation of add
end
```

ordinary inheritance class relationship such an instantiation is very necessary since the foster class must be instantiated in order to be inherited. Thus, this case is invalid.

Rule ***Exheriting Non-generic Subclasses with a Generic Foster Class.*** When subclasses are non-generic but the foster class is generic and unconstrained, then such a class combination is not valid since instantiation information in inheritance does not exist and cannot be inferred.

5.3.2 Constrained Genericity

In the case of constrained genericity the generic parameter conforms to a type. So, an object can receive a restricted set of messages through the interface of that type. We take the same structural example similar to unconstrained genericity. Classes $A[G1 \rightarrow T1]$, $B[G2 \rightarrow T2]$ are created first, they have generic formal parameters *G1*, *G2* and those parameters will conform to types *T1*, respectively *T2*. Then class $C[G3 \rightarrow T3]$ is created by reverse inheritance with A and B. We consider various cases for the parameters and types.

Non-generic Foster Class and Generic Subclasses

In example 84 if C has non-generic features, then the non-generic features can be normally exherited. Depending on the relations between *T1* and *T2* it may be possible to exherit features involving generic types. If $T1 = T2$ then the type used in the feature exheritance can be $T = T1 = T2$. Otherwise, another type T can be used in the superclass C if *T1* and *T2* conform to T. A special case is when *T1* conforms to *T2* or vice-versa.

Class C is not generic and formal generic parameters *G1* and *G2* conform to type T. So, in superclass C attribute a and the signature of method m are equipped with the corresponding type T. If such a type as T does not exist, then exheritance of the features having generic parameters is not possible. Anyway such a type can always be created through reverse inheritance. Thus, supertypes must exist for all generic parameters in the feature signature.

In the exheritance branches the generic classes must be used and not some generated ones. In the equivalent class relationship any class instantiation type information is lost. The other option would be to use the syntax based on the type *NONE* as in subsection 5.3.1.

Rule ***Exheriting Generic Subclasses with a Non-generic Foster Class.*** If the subclasses are generic having constraints and the foster class is non-generic then exheritance of non-generic features is possible under normal conditions and the exheritance of generic features can be made by redefinition if common supertypes exist for all the corresponding constrained types.

Example 84 Constrained Genericity (1)

```
class A[G1 -> T1]
feature
  a: G1
  m(p: G1) is do ... end
end
class B[G2 -> T2]
feature
  a: G2
  m(p: G2) is do ... end
end
foster class C exherit
 A
  redefine a,m
 end
 B
  redefine a,m
 end
 all
feature
  a: T
  m(p: T)is do ... end
end
class T1 inherit T end
class T2 inherit T end
```

Generic Foster Class and Generic Subclasses

We analyse the case of all classes having a generic parameter, like $A[G1 \rightarrow T1]$, $B[G2 \rightarrow T2]$, $C[G3 \rightarrow T3]$. If *T1* and *T2* conform to *T3*, then factorization of generic features is possible.

In example 85, generic class C exherits generated classes A and B with type *G3* conforming to *T3*. Type *T3* is a supertype of the *T1* and *T2* which are types expressing the constraints for the formal generic of subclasses A and B.

In ordinary inheritance the instantiation types for the superclass must conform to the constraints of that class, but in reverse inheritance the instantiation type for the subclasses generally will not conform to the constraints. In the equivalent class hierarchy based on ordinary inheritance the formal generic *G1* conforms to *T1* and can be used in the instantiation of the superclass C because *T1* conforms to *T3*. Conversely, in the context of exheritance, the formal generic *G3* which conforms to *T3* will not guarantee that conforms also to *T1*, so exheritance instantiated with generics is invalid in general. The only situation when such exheritance is possible is when T1=T2=T3. Attribute a and method m use a generic parameter *G3* in the superclass which conforms to *T3*.

If *G3* is unconstrained then we can say that it is constrained to *ANY*, so we are still in this case. If the formal generics from the exherited classes *G1* and *G2* are unconstrained or constrained to *ANY* then we are still in the current case.

In conclusion we may say that such a class configuration validity is very restricted.

Rule ***Exheriting Generic Subclasses with a Generic Foster Class***. If the subclasses and the foster class are both generic having constraints then exheritance of non-generic features is possible under normal conditions and the exheritance of generic features can be made by redefinition if all involved types are the same.

Example 85 Constrained Genericity (2)

```
class A[G1 -> T1]
feature
   a: G1
   m(p: G1) is do ... end
end
class B[G2 -> T2]
feature
   a: G2
   m(p: G2) is do ... end
end
foster class C[G3 -> T3] exherit
 A
  redefine m
 end
 B
  redefine m
 end
 all
feature
  a: T3
  m(p:T3) is do ... end --to write a new code
end
class T1 inherit T3 end
class T2 inherit T3 end
```

Generic Foster Class and Non-generic Subclasses

The last case in which class $C[G3 \rightarrow T3]$ is generic, is the same as the case of unconstrained genericity. The class configuration is not valid since the instantiation information does not exist and cannot be inferred.

Rule ***Exheriting Non-generic Subclasses with a Generic Foster Class***. If subclasses are non-generic but the foster class is generic and constrained, then such a class combination is not valid since instantiation information in inheritance does not exist and cannot be inferred.

5.4 Redefining Preconditions and Postconditions

The assertion mechanism of Eiffel helps checking software text correctness meaning whether the implementation of a feature conforms to its specification. The basic element of this mechanism is the assertion. Assertions represent boolean expressions which use Eiffel logical operators (coming from class *BOOLEAN*) and the features declared within classes. They are used to express abstract properties of classes. By assertions we mean preconditions, postconditions and invariants. Class invariants behave like postconditions so for them we will apply similar rules to postconditions. Let us investigate how assertions are affected by single and multiple inheritance.

First, in the case of ordinary inheritance, any assertion implicitly includes the corresponding assertion in the parent. On the other hand, an inherited feature may change the precondition of the parent by weakening it (by writing an alternative precondition) and may change the postcondition of the parent by strengthening it (by writing an extra postcondition). The reason for which preconditions are kept or weakened is for any older clients of the superclass to be able to use the subclass. In the case of postconditions, since subclasses are specializations of the superclasses they should have new constraints related to the newly added features.

Example 86 Constrained Genericity (3)

```
class A
feature
  a: T1
  m(p: T1) is do ... end
end
class B
feature
  a: T2
  m(p: T2) is do ... end
end
foster class C[G3->T3] exherit
 A
  adapt a,m
 end
 B
  adapt a,m
 end
 all
feature
  a: G
  m(p: G) is adapted end
end
class T1 inherit T3 end
class T2 inherit T3 end
```

If we consider that *pre1, pre2, ..., pren* are the preconditions and *post1, post2, ..., postn* are the postconditions in the precursor then a redeclared routine will have the following equivalent assertions[6]:

```
alternative_precondition or else pre1 or else pre2 ... or else pren
extra_postcondition and then post1 and then post2 ... and then postn
```

When an inherited feature has not declared any additional preconditions or postconditions then the *alternative_precondition* is equivalent to *false*, since *false* is neutral to the *OR* logical operator and *extra_postcondition* is equivalent to *true*, since *true* is neutral to the *AND* logical operator.

To exherit features means also to determine which will be their new preconditions and postconditions in the superclass, should we exherit a part or the whole preconditions and postconditions that come along with the features. In this section we will see how this task can be accomplished and in which cases it can be automated.

5.4.1 Eliminating Non-Exherited Variables

In [LHQ94], a rule is proposed for handling preconditions. It defines the precondition of a feature in the superclass as the *AND*-ing of all the corresponding preconditions from subclasses. The same is proposed for postconditions but using the logical *OR* operator to compose them. So the foster class will have a stronger precondition and a weaker postcondition. We rely on this approach and we will augment it later on.

Since *true* is the weakest assertion it can be used as postcondition in the superclass. It is not equivalent to use *false* on the corresponding position for precondition, although it is the strongest assertion. Any call to such a feature will be rejected, considering that the precondition failed. So in conclusion the foster class precondition, respectively postcondition, will be the following:

[6] We mention that when dealing with single inheritance n equals to 1.

```
pre1 AND pre2 AND ... AND pren
post1 OR post2 OR ... OR postn
```

In some cases it is impossible to define a valid precondition for the foster class. This happens when at least two preconditions are in contradiction and the foster class precondition will always fail. Such an example is straightforward if *pre1* is $a<5$ and *pre2* is $a>5$. In this cases exheritance is forbidden for conforming reverse inheritance and allowed only for non-conforming reverse inheritance.

The [LHQ94] paper mentions also that some assertions may contain features which are not exherited. One radical solution is to invite the programmer to write from scratch all the assertions. The second solution would be to prompt the programmer to exherit the depending features. Of course, this solution is more or less applicable depending on the semantical restrictions. Another possible solution is to adapt the assertions by disabling those logical subexpressions which contain non-exherited features. In our approach the idea is to automatically modify the boolean expression in such manner that it will affect as little as possible the evaluation. For example, let us consider the features *a*, *b*, *c*. The following transformations may appear if *c* is not exherited or in the last case, if *e1* and/or *e2* contain features which are not exherited:

```
01 (a<b) and (b<c) is transformed in : (a<b)
02 (a<b) or (b<c) is transformed in : (a<b)
03 (a<b) xor (b<c) is transformed in : (a<b)
04 not (b<c) is transformed in : void
05 e1 implies e2 is transformed in : void
```

On line 01 the second operand of the expression *and* could not be evaluated, since *c* is missing in the new context. It is the same for the logical expressions defined on lines 02 and 03. On lines 04 and 05 the two logical expressions are replaced with *void*, this means that those logical expressions are actually ignored. In order to perform such transformations it is necessary to analyze all logical operators from Eiffel and to determine their neutral elements. We consider that *E* is a logical expression.

```
E and true = true and E = E
E or false = false or E = E
E xor false = false xor E = E
```

The operators *not* and *implies* cannot be handled in the same manner as the operators *and*, *or*, *xor*, because the operator *implies* is not reflexive and the operator *not* is an unary operator. Further on we will see what can we do with these operators. Taking into account the priority of Eiffel operators we can define some rules in order to reduce the boolean expressions containing features which are not exherited. A special **reduce** operator will be defined for this purpose (see figure 5.1).

In the extreme case when all features from one assertion are not exherited, then in the superclass that assertion will be ignored.

The order of evaluation of the assertions in our approach is not important since all assertions from different classes will have to be independent. As syntactical formalisms we propose the ones which are defined in example 87.

In example 87 the precondition of feature *f* in class *C* is composed with the precondition of class *A* and precondition of class *B* using the *AND* operator for composing them. The same is done for postcondition, but using the *OR* operator. The built-in expressions **precondition{***classname***}**, **postcondition{***classname***}** are used (without code duplication) to denote the precondition and postcondition of the current feature from the class specified between the brackets. If there are non-exherited features in one of the assertions the **reduce** operator will be applied first, implicitly on the respective assertion.

The **require stronger** and **ensure weaker** keywords are used in order to make the user aware that he is responsible for writing a stronger precondition and a weaker postcondition in the

```
reduce(E)
 = E       -- E contains just exherited features
 = void    -- E cannot be decomposed further on and contains non-exherited features
reduce(E1 and E2)
 = E1 and E2  -- E1 and E2 contain just exherited features
 = reduce(E1) -- reduce(E2) is void
 = reduce(E2) -- reduce(E1) is void
 = void       -- reduce(E1) is void and reduce(E2) is void
reduce(E1 or E2)
 = E1 or E2   -- E1 and E2 contain just exherited features
 = reduce(E1) -- reduce(E2) is void
 = reduce(E2) -- reduce(E1) is void
 = void       -- reduce(E1) is void and reduce(E2) is void
reduce(E1 xor E2)
 = E1 xor E2  -- E1 and E2 contain just exherited features
 = reduce(E1) -- reduce(E2) is void
 = reduce(E2) -- reduce(E1) is void
 = void       -- reduce(E1) is void and reduce(E2) is void
reduce(E1 implies E2)
 = E1 implies E2 -- E1 and E2 contain just exherited features
 = void          -- reduce(E1) is void or reduce(E2) is void
reduce(not E)
 = not E         -- E contains just exherited features
 = not reduce(E) -- E can be decomposed further on
 = void          -- E cannot be decomposed further on and contains non-exherited features
```

Figure 5.1: Exheritance and Assertion Redefinition

Example 87 Exheritance and Assertions: The Syntax

```
class C exherit
  A
   redefine f
  end
  B
   redefine f
  end
  all
feature
  f(x:INTEGER) is
    require stronger
       precondition{A} and precondition{B}
    do
        ...
    ensure weaker
        postcondition{A} or postcondition{B}
    end
```

Example 88 Exheriting the "only" Clause

```
class A
 feature
  a,b,c:INTEGER;
  f is
   require ...
   do ...
   ensure
    only a,b,c
  end
end
class B
feature
  b,c,d:INTEGER;
  f is
   require ...
   do ...
   ensure
    only b,c,d
  end
end
class C exherit
 A
 B
 all
end
```

foster class. Checking if the precondition in the foster class is stronger and the postcondition is weaker is difficult to detect at compile time, because there are multiple preconditions and multiple variables involved. The idea used here is the same like the one used in ordinary inheritance with the keywords **require else** and **ensure then**. In such cases the user has to be aware that for a subclass feature, the precondition, respectively the postcondition from the superclass is evaluated first and only then, the locally defined assertions.

In Eiffel, the mechanism based on the keyword **old** allows to refer to the values of variables before the method execution. Since it belongs to the code execution part, it will not be affected directly by the semantics of reverse inheritance.

The clause **only** is used for declaring the variables whose values can be changed during the execution of a routine; it can be affected by reverse inheritance. This happens when some variables from the list are not exherited in the foster class. In the superclass, the variables which are not exherited cannot belong to the list. The list of the **only** clause can have at most the common features which are exherited.

If the exherited method is redefined in the foster class, the **only** clause can be modified freely according to the set of exherited features. If the implicit behavior of reverse inheritance for this aspect is chosen (see example 88) then just the exherited features from each subclass **only** list are kept. In example 88 features f, b and c will be exherited and in the foster class feature f will have the following postcondition: from the **only** list of feature f in class A feature a will be removed since it is not exherited, the same thing happens to feature d.

Method body assertions like **check**, **loop** variants are not affected directly by the reverse inheritance class relationship. As they are part of the body of a method they can be exherited taking into account the rules related to body exheritance.

Assertions tags in the subclasses, if any, may be kept in the superclass unless some name conflict arises in which case renaming has to be performed by the programmer.

5.4.2 Combined Precondition and Combined Postcondition

A different approach on assertions has its origin in section 8.10.5 of[Int06]. The combined precondition and postcondition of a feature in a subclass having multiple superclasses are defined as follows:

```
pre1 or ... or pren or else pre
(old pre1 implies post1)
 and ... and
(old pren implies postn) and then post
```

The *pre1,...,pren* are the preconditions and the *post1,...,postn* are the postconditions from the corresponding features from the superclasses. In the new standard of Eiffel [Int06] there is a new object test in the form of *{x:T} exp*, where *exp* is an expression, T is a type and the whole construct is a boolean expression evaluating whether *exp* is of type T and attaching x reference to it, in the scope of the test object.

The critical point in a foster class is the fact that the preconditions and postconditions are applicable only to the objects that are instances (direct or indirect) of the corresponding heir classes. Assuming that *prei* and *posti* are the precondition and postcondition of a feature f present in the heir class *Ci (i = 1,...,n)*, and *pre'* and *post'* those declared in the foster class C, as an effective precondition in class C we propose:

```
if ({x1:C1} Current or else ... or else {xn:Cn} Current)
then
({x1:C1} Current implies pre1)
 and ... and
({xn:Cn} Current implies pren) and then pre'
else
 pre''
```

For the effective postcondition, similarly we can have:

```
if ({x1:C1} Current or else ... or else {xn:Cn} Current)
then
({x1:C1} Current implies post1)
 and ... and
({xn:Cn} Current implies postn) or else post'
else
 post''
```

These expressions must indeed be the strongest possible precondition and the weakest possible postcondition. Testing the type of the instance will actually determine exactly which conditions will be checked from the effective precondition and postcondition of the foster class. For an object of type *Ci*, the test object will return true only in *{xi:Ci} Current implies prei*, respectively in *{xi:Ci} Current implies posti*, while in all the other subexpressions it will return false. The *pre'* and *post'* are used to strengthen respectively to weaken the combined precondition, respectively the combined postcondition. If an object is an instance of the foster class (but not of exherited classes) then *pre"* and *post"* assertions are used.

In [LHQ94] the requirements are too strict, and therefore, for instance, exheritance would often be considered impossible, especially if some heir class has elaborated preconditions. In our earlier solution (see subsection 5.4.1), the requirements have been relaxed too much, and therefore the desired conformance between a foster class and the exherited classes would often not be achieved.

5.5 Summary

Regarding the classical adaptation mechanisms from ordinary inheritance, it seems to occur no problem when applying them to reverse inheritance. Feature redefinition mainly has to satisfy signature and implementation exheritance restrictions. Feature undefinition in the context of reverse inheritance was made implicit for both attributes and methods.

Scale adaptations allow specifying mathematical formulas around the exherited features. This mechanism is quite simple and suits only to some simple situations, but the idea can be improved in order to be able to perform more general adaptations.

Parameter order adaptation involves only a translation scheme for the parameters. The proposed syntax is quite simple and sufficient to express the semantics of the adaptation. Parameter number adaptation works only in several restricted cases when it is possible to unify two simple signatures for not so complex features. In one of the sub-cases some mathematical formulas (idea taken from scale adaptations) were used. From this point we can generalize and to allow not just mathematical operations but any expression built with the language constructs. The two adaptations related to scale and parameter number use the same syntax built around **adapt** and **adapted** keywords.

When generic classes are the target of reverse inheritance there were analyzed several cases of unconstrained and constrained genericity, combined with cases of generic/non-generic foster class/subclass. The case of constraint/unconstraint genericity was taken into account. The anaysis was performed taking into account that unconstraint genericity can be viewed as constraint genericity to class *ANY*.

Finally, the adaptation of assertions was studied. Feature exheritance is successful if it is possible to define a precondition other than *false*, which is stronger than each precondition in the subclasses for the corresponding feature. In the worst case, for postconditions and invariants, the *true* postcondition can be used . Regarding the features which are not exherited and are parts of the assertions an algorithm was presented for eliminating those features from the logical expression. The algorithm was based on the neutral values for each logical operator. Because deciding at compile time whether the precondition for the foster class is stronger than the preconditions in the subclasses, the **require stronger** and **ensure weaker** keywords are proposed for use. A different solution which requires information about the type of the exherited class instance manipulated through the common interface of the foster class is the combined precondition and postcondition. Using type information, the combined assertions corresponding to other subclasses are invalidated, enabling only the evaluation of the original subclass assertion and of the additional logical expressions written in the foster class.

Chapter 6

Coupling Exheritance with Inheritance

In this chapter we will discuss aspects related to the integration of reverse inheritance in complex class hierarchies built by ordinary inheritance. We will analyse the problems that may arise considering several types of class configurations. We are interested in:

i) how the exherited features are inherited in regular classes built by ordinary inheritance;

ii) what is the origin of the exherited features when the exherited class has ancestors;

iii) what happens with both inherited and exherited features from an ancestor-descendant class pair;

iv) what kind of parent classes a foster class may have;

v) how can a foster class exherit from another foster class;

vi) how can a foster class inherit from another foster class.

An important part of this chapter is dedicated to the dynamic binding aspects and to the feature selection issues. We will discuss also about the constraints that must be set on the exherited features and classes regarding aspects like: exheritance of export clauses, exheritance of creator, once, frozen, obsolete features, impact of aliases and precursor on the exheritance semantics.

6.1 Combining Reverse with Ordinary Inheritance

Let us have a look at the possible combination of reverse and ordinary inheritance (see table 6.1) in a complex class hierarchy:

Ordinary classes are not allowed to exherit anything. Ordinary classes are allowed to inherit ordinary and foster classes. Foster classes may inherit from classes and even from foster classes with the condition of not inheriting new features. Foster classes may exherit from any classes ordinary or foster.

Actions	**Class(es)**	**Foster class(es)**
class inherits	allowed	allowed
class exherits	not allowed	not allowed
foster class inherits	allowed but constrained	allowed but constrained
foster class exherits	allowed	allowed

Table 6.1: Possible Combinations of Ordinary Inheritance and Reverse Inheritance

Example 89 Inheritance from Foster Class

```
class RECTANGLE
feature
 perimeter:REAL is do ... end
 halfperimeter:REAL is
 do
  perimeter/2;
 end
end
class ELLIPSE
feature
 perimeter:REAL is do ... end
 halfperimeter is
 do
  -- ellipse implementation
 end
end
foster class SHAPE exherit
 RECTANGLE
  moveup halfperimeter
 end
 ELLIPSE
 all
feature
end
class TRIANGLE inherit SHAPE
end
```

6.1.1 Inheriting From a Foster Class in an Ordinary Class

First, we will discuss about the inheritance of exherited features in new descendants. It is a natural decision to keep the principle of inheriting in new descendants any feature of the class in the case of foster parent classes. In order to be more explicit we will start from the example given in subsection 4.3.4 and we will extend it with a new descendant class.

In example 89, we created a new class *TRIANGLE* which is derived from foster class *SHAPE*. It will benefit from all the features of class *SHAPE*. Since class *SHAPE* is the target of the reverse inheritance class relationship, it will obviously have just exherited features. The features from class *SHAPE* are both exherited from *RECTANGLE* and *ELLIPSE* on one hand, and inherited in *TRIANGLE* on the other hand. In this particular case we exherited and inherited *perimeter* method signature and *halfperimeter* method implementation. As it is designed last, the foster class may have only new descendant classes.

Rule ***Inheriting from a Foster Class.*** Features exherited by a foster class will be inherited by any descendant of the foster class, applying the classical rules of Eiffel.

6.1.2 Exheriting from a Descendant

In the example described in figure 6.1, we take into consideration a foster class FC which exherits from an exherited class EC and which has a parent class PC. We are interested in particular about the exheritance of the features inherited from PC in EC and not the ones immediately defined in EC since they were already analysed in chapters 4 and 5.

The case is general since we can consider that class PC is a flattened version of itself and its parents. Still, such a class construction is equivalent to a multiple inheritance construction where

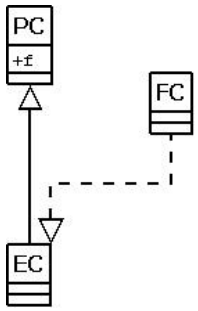


Figure 6.1: Exheriting from a Descendant

class *EC* is obtained by multiple inheritance out of classes *PC* and *FC*. Because we do not want to affect class *EC* by the features of class *FC* factored by reverse inheritance, we have to disallow replication of feature *f* in class EC. If we analyze the nature of inherited features that might be exherited we draw the conclusion, that they can be deferred and effective methods and attributes as well in both *PC* and *EC* classes.

Getting the Implementation for Source Features from the Exherited Class To get a single implementation for feature *f* in the exherited class *EC* all potential implementations coming from class *FC* must be undefined. If feature *f* in *FC* is deferred then no conflicts may arise.

In figure 6.2 we presented all possible cases for feature f in classes *FC* and *EC*. In class *EC* feature *f* can be deferred, effective or inherited, while in class *FC* it can be deferred, effective or moved up.

In case *1a* both features are deferred, so no special treatment is necessary. In case *1b* the feature in *FC* is effective, while the one in *EC* is deferred, so undefinition must be operated. Case *1c* is invalid since no implementation exheritance is possible from a deferred feature.

In case *2a* the deferred version from *FC* is effected in *EC*, so no adaptations are necessary. In case *2b* two effective versions are present, so a redefine clause is necessary. In case *2c* the implementation from *EC* is exherited in foster class *FC* and a redefine keyword would make the hierarchy consistent.

In case *3a* the deferred version from *FC* is in no conflict with the inherited version from *PC*. In case *3b* the effective version from *FC* would conflict with the inherited version from *PC*, so the version from *FC* will be undefined. In case *3c* the exherited version from *EC* is preferred to be undefined not to conflict with the inherited version from *PC*.

6.1.3 Inheriting from an Ancestor and Exheriting from a Descendant

In figure 6.3 we captured a different class configuration where reverse inheritance is used. According to the definition rules of reverse inheritance we are not allowed to create new inheritance paths between unrelated existing classes nor deleting existing paths. This means that reverse inheritance can be used only between classes which are in an ancestor-descendant relationship.

In the class hierarchy there are two classes *PC* (parent class) and *EC* (exherited class) in ancestor-descendant relationship. Between these classes there may be any number of intermediate classes on the ordinary inheritance line. Foster class *FC* inherits from class *PC* directly or indirectly and exherits directly from class *EC*. The selected case is considered general because even in the class configurations presented so far the foster class still inherits from class *ANY*. In this context we have features inherited from class *PC* and also features exherited from class *EC*. In such a class configuration any other parents of *FC* which are not already ancestors of *EC* will be linked to *EC* by reverse inheritance, even if they were originally unrelated. Such parents may be newly created classes, but not already existing ones as it will be explained in subsection 6.1.4.

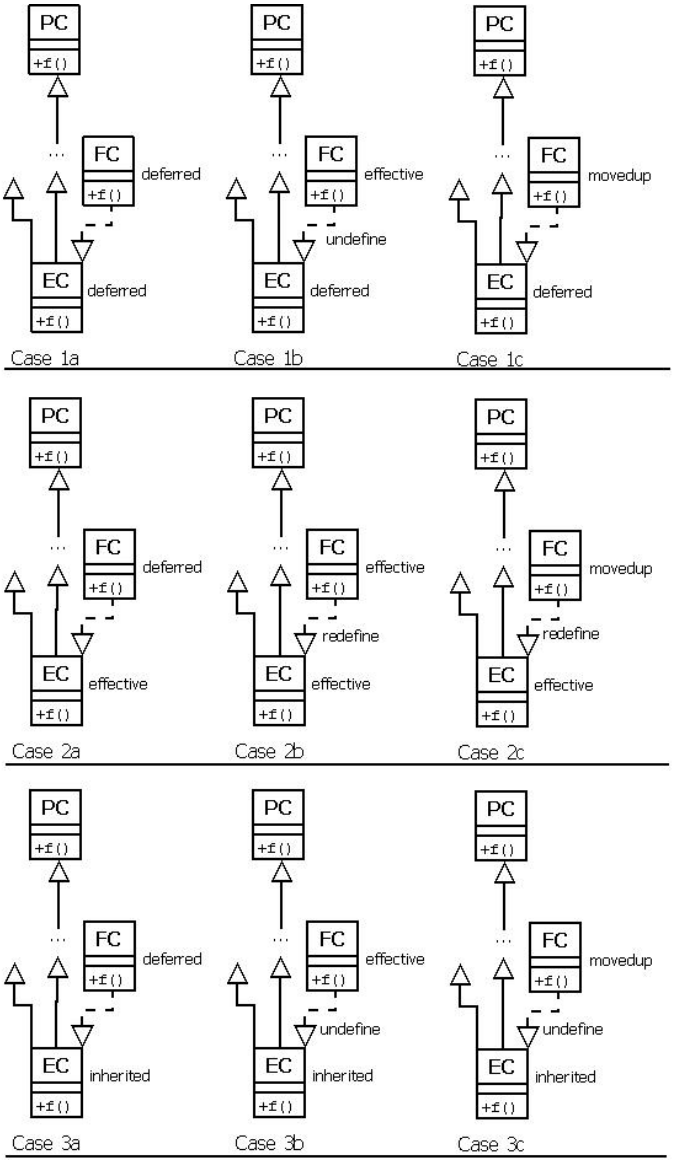


Figure 6.2: Getting the Implementation for Source Features from the Exherited Class

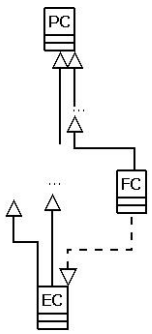


Figure 6.3: Inheriting from an Ancestor and Exheriting from a Descendant

All the features from class PC are inherited in class EC through the ordinary inheritance path. The features from class EC can be divided into three categories:

- inherited from the parent class PC;
- inherited from other ancestors;
- immediate or locally defined.

The set of features in the foster class FC is composed of several subsets:

- the whole set of features inherited from parent class PC and exherited from exherited class EC. This set of features is called **amphibious features** since they are both inherited and exherited. The exheritance of amphibious features is implicit and it cannot be prohibited using except clauses.
- a subset of the features exherited from EC, other than the amphibious ones, as deferred, moved up or redefined.

In the case of multiple exheritance all the exheritance rules presented in the previous chapters hold. In the case of redefined amphibious feature in the foster class FC with redefined signature the type covariance rule must hold. This means that the corresponding signature types from foster class FC are subtypes of the types from parent class PC and also types from the feature signature of exherited class EC are subtypes of the types from the signature within FC.

Getting the Implementation for Amphibious Features Since amphibious features are both inherited and exherited they can get their implementation from three sources: parent class PC by inheritance, exherited class EC by moving up or immediate as locally defined in FC. In the context of multiple inheritance the feature implementation selection is made by **undefine** / **redefine** keywords combination and in reverse inheritance we preserve the same philosophy. Using the same keywords, for each exherited feature at most one implementation must be set. The implementation selection is made through the proper combination of clauses on the inheritance and exheritance branches. In the exherited class EC there must be set only one implementation for feature f also.

In figure 6.4 we present all the possible class configurations with the proper combination of inheritance and exheritance clauses. In PC the feature may be deferred or effective, in EC it may be deferred, effective or inherited while the feature from FC may be deferred, effective, inherited or moved up. Combining these feature statuses we obtain all possible class configurations.

In case *1a* all feature versions are deferred, so there is no keyword action to be taken. In case *1b* the feature within FC is deferred so both effective versions from PC and EC must be undefined. For ordinary inheritance undefinition must be set explicitly, while for reverse inheritance it works implicitly. Cases *1c* and *1d* are invalid since the implementation inheritance or exheritance is not possible while the candidate feature is deferred.

In case *2a* feature f from FC is deferred as the version from PC and the effective version from EC is exherited implicitly as deferred. In such case no feature modifiers are necessary. In case *2b* feature f from FC if effective, effecting the deferred version from PC and redefining the effective version from EC. Case *2c* is invalid since implementation inheritance is not possible when the ancestor feature is deferred. In case *2d* the implementation of feature f is moved up from the EC and effects the deferred implementation from PC. Obviously that the **moveup** keyword is used on the exheritance branch, while on the inheritance branch no keyword is needed.

Cases *3a*, *3b*, *3c* and *3d* are invalid since implementation inheritance is not possible from a deferred source feature.

In figure 6.5 we continue the presentation of the valid cases.

In case *4a* feature f is deferred while its version from PC is effective so an undefine clause is necessary. In case *4b* the two effective versions from PC and FC must be separated by a redefine

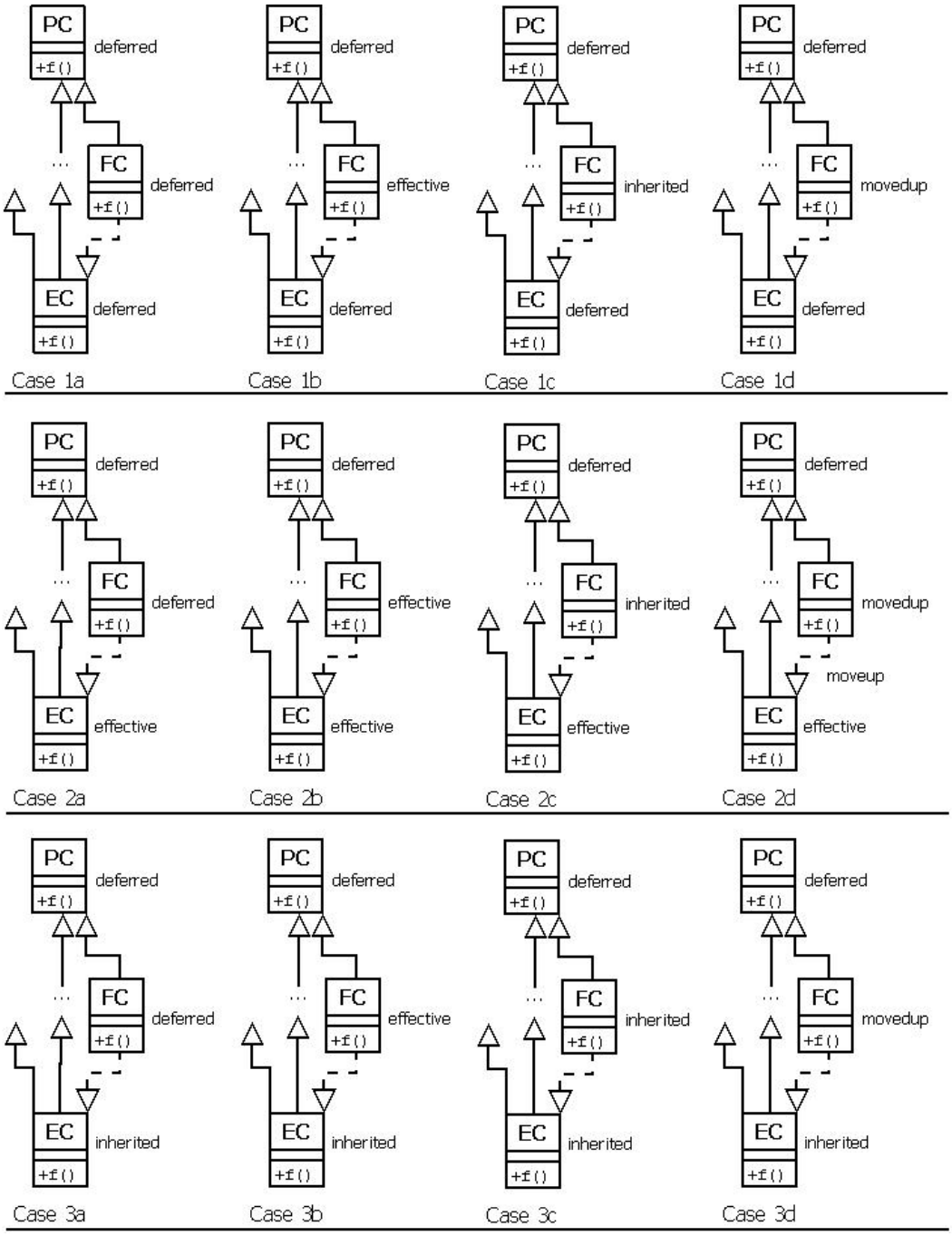


Figure 6.4: Getting the Implementation in Amphibious Features (1)

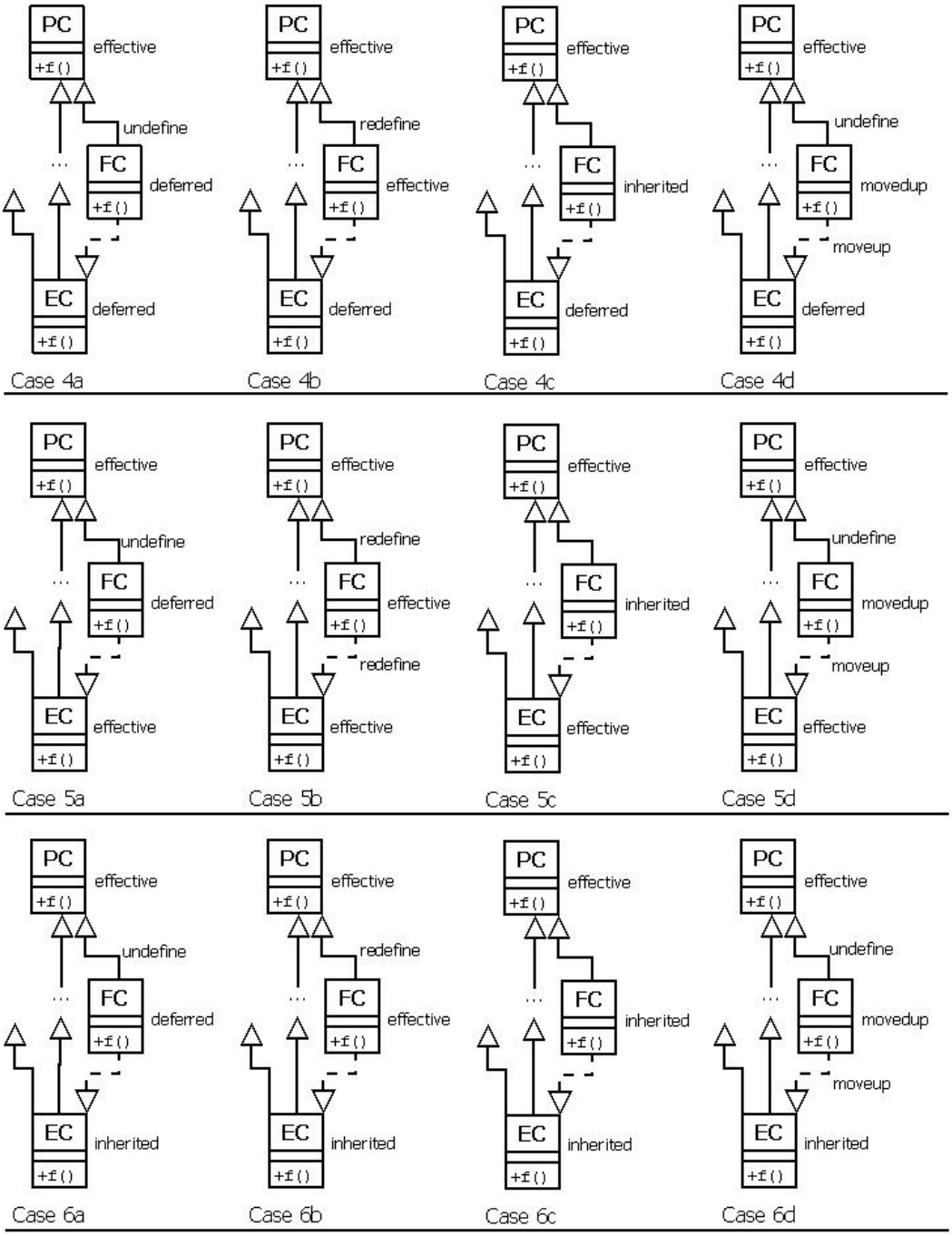


Figure 6.5: Getting the Implementation in Amphibious Features (2)

Figure 6.6: Restricted Inheritance in Foster Classes

keyword. The deferred version from *EC* is effected in *FC* so on the exheritance branch no feature clause is necessary. In case *4c* the effective version from *PC* is inherited in *FC* and the deferred version from *EC* is effected in *FC*. In such case no feature clauses are necessary. Case *4d* is invalid since implementation exheritance is not possible when the exherited feature is deferred.

In case *5a* the deferred version in *FC* is obtained from two effective versions, so the inheritance branch requires the undefine keyword, while on exheritance the undefinition is implicit. In case *5b* we have three effective versions which must be separated by two redefine keywords. In case *5c* the inherited version of the feature in *FC* requires no special keyword handling. In case *5d* the moved up version from *EC* requires the use of the moveup keyword and the undefinition of the version coming from *PC*.

In case *6a* the deferred version from *FC* needs undefine on the inheritance branch while on the exheritance branch this is implicit. In case *6b* the effective version from *FC* is redefined on the inheritance branch and undefined implicitly on the exheritance branch. The undefine is important since after transformations it will become explicit in the context of ordinary inheritance. In case *6c* there are two inheritance paths and the original one must be favored in *EC* by undefining the feature version coming from *FC*. In *FC* the feature version from *PC* is inherited, while the exherited version from *EC* is deferred. In case *6d* the inherited version from *EC* is moved up in *FC* so the inheritance of the version from *PC* must be prohibited by an undefine clause.

6.1.4 Restricted Inheritance in a Foster Class

A foster class *FC* which declares a reverse inheritance relationship to a class *EC* may also inherit from a parent class *PC*, if *PC* contains at most the same features as *FC*. This constraint is necessary in order not to change the behavior of FC descendants. It is interesting to get this opportunity when we want to keep and reuse some features that are already defined (see figure 6.6):

Because of the constraint described above it is necessary to add some constraints on the clause that may be set when using ordinary inheritance within a foster class. In particular this is the case when features are renamed.

Rule ***Inheriting in a Foster Class***. In a foster class *FC* if one method *f* belongs to an ancestor *PC* of *FC*, then the feature must be exherited also and both of them must have the same name. If *f* is renamed when inheriting from *PC*, then it must be also renamed when exherited, obviously with the same name.

6.1.5 Exheriting from a Foster Class

In this subsection we discuss the idea that a foster class could have another foster class on its top. From this point of view, comparing to ordinary inheritance, it is possible to create subclasses

Example 90 Exheriting From a Foster Class (1)

```
class RECTANGLE
 feature draw is do ... end
 ...
end
class ELLIPSE
 feature draw is do ... end
 ...
end
foster class SHAPE
 exherit
  RECTANGLE
  ELLIPSE
  all
 end
end
foster class GRAPHICAL_OBJECT
 exherit
  SHAPE
  all
end
```

to already existing subclasses, thus making the class hierarchy to evolve downward. Reverse inheritance facilitates the upward evolution of class hierarchies like in example 90.

The only delicate issue that can be noted in this situation is the visibility of the implicitly exherited features in cascade. In example 90 the exheritance of the *draw* feature in foster class *SHAPE* is implicit, so the feature is not explicitly listed in the foster class. When the exherited features are not listed explicitly, in a long chain of foster classes it would be very difficult to see which features succeeded to be exherited along the exheritance chain.

On the other hand, the reverse inheritance relationship is meant to be used at redesign time when several classes are available. In this context the creation of a foster class for an already existing foster class implies another process of redesign. In the very same redesign process, ordinary inheritance may be used, like in example 91.

The main idea in example 91 is to favor the use of ordinary inheritance instead of reverse inheritance in the same redesign stage. Reverse inheritance is designed for homogenizing classes having different authors and which originally belonged to independent design processes.

6.1.6 Exheriting from a Hierarchy

In figure 6.7 we have the case of two classes *PC* and *EC* linked by ordinary inheritance. In parallel we create another class hierarchy of two foster classes *FC2* and *FC1* linked by ordinary inheritance and each class exheriting its counterpart: *FC2* exherits *PC* and *FC1* exherits *EC*. The set of features exherited from *PC* in *FC2* is a subset of features exherited from *EC* in *FC1* because the set of features from *PC* is a subset features of *EC*.

6.2 Considering the Time Stamp When Defining a Class

In figure 6.8 we study the different orders in which the class hierarchy is built. In this case class A is the highest ancestor, classes B and C are intermediate and class D is the lowest descendant.

Example 91 Exheriting From a Foster Class (2)

```
class RECTANGLE
 ...
end
class ELLIPSE
 ...
end
foster class SHAPE
 inherit
  GRAPHICAL_OBJECT
 exherit
  RECTANGLE
  ELLIPSE
  all
end
class GRAPHICAL_OBJECT
 ...
end
```

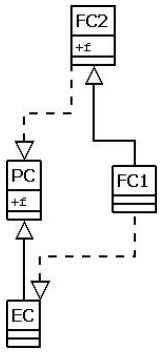


Figure 6.7: Exheriting from a Hierarchy

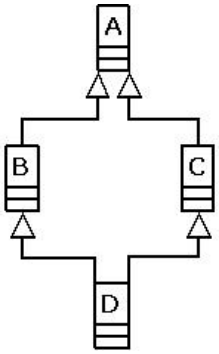


Figure 6.8: Fork-Join Inheritance Example

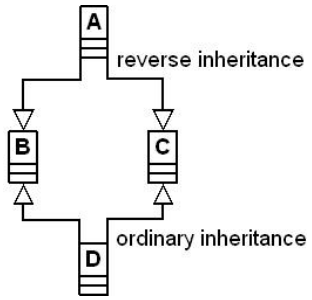


Figure 6.9: Sharing Features (case 1)

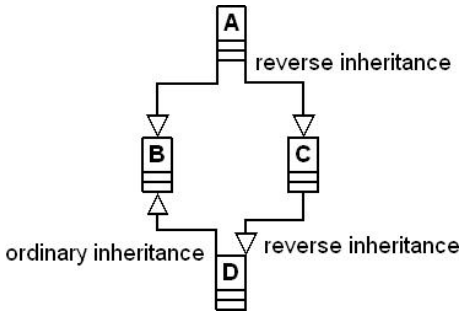


Figure 6.10: Sharing Features (case 2)

An even simpler class constellation than the diamond, but still with fork-join inheritance, is a triangle. Let us drop class C and instead have also a direct inheritance link between A and D. In this situation, it is clearly possible to share a feature f in all possible definition orders (ordinary inheritance and reverse inheritance) of the classes. For a replicated feature, a similar extended clause **select** as above is needed if D is not defined last.

6.2.1 Sharing Features

The different class construction scenarios reported to the temporal coordinate will be marked using "$+$" symbol for those classes constructed by ordinary inheritance and "$*$" for those constructed with reverse inheritance. In Eiffel an attribute declared both in classes B and C cannot be unified in class D unless they have a common seed (except if the attributes are redefined in all inheritance branches). This happens when B and C have a common ancestor. This restriction really looks like an unnecessary non-orthogonality in Eiffel. Some different orders can be imagined in which the sharing of features from B and C is prevented by the rules of Eiffel [1]:

- Case *1*: ***BC+D*A***. This means that classes B and C are created first, then class D is defined by multiple ordinary inheritance from B and C. Finally, class A is built by multiple reverse inheritance from B and C (see figure 6.9).

- Case *2*: ***B+D*C*A***. This means that class B is created first, then D by inheriting from B, then C is built by reverse inheritance from D. Finally, class A is designed using multiple reverse inheritance from B and C (see figure 6.10).

[1] We remind the reader that adding a class to a hierarchy by reverse inheritance will not affect the behavior of the rest of the hierarchy.

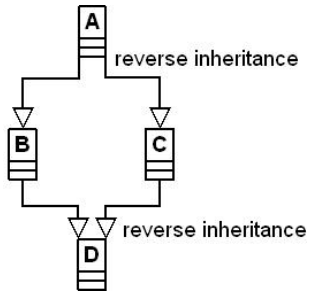


Figure 6.11: Sharing Features (case 3)

- Case *3*: ***D*B*C*A***. This means that class D exist first, then classes B and C are built from D by reverse inheritance and that class A is also built through reverse inheritance from B and C (see figure 6.11).

The cases in which we swap B and C are equivalent so that only one order needs to be treated. We can say that the cases leading implicitly to feature sharing corresponds to the cases where A is defined last. It is interesting to note that if a language contains the capability to define reverse inheritance relationships, then it is not necessary to allow the unification of two features without a common seed. A common seed can be always provided by reverse inheritance.

6.2.2 Replicating Features

The more complicated alternative is a feature f that should be replicated, so that there are two occurrences of f in an object of type D, one corresponding to B and the other to C. One of these two should be statically selected to act as f when an object of type D is accessed through a variable of type A. For each definition order below, the orders where B and C are swapped are equivalent, of course.

Let f_{final} be the final name of the occurrence of f that should be selected in class D. Note that in all definition orders in which not both B and C are defined before D, there must be defined also some other feature in D that can be exherited as *f*.

- Case *1*: ***A+B+C+D***. This means, only ordinary inheritance is used, f must be renamed and/or redefined in B and/or C and/or D, but there are no problems. As we know, the existing Eiffel syntax is simply "**select** f_{final}". It must be put in the right inheritance branch to select f_{final}.

- Case *2*: ***B*A+C+D***, ***BC*A+D***. The handling of these two situations does not differ essentially from *case 1*, because D is defined last, so that it allows to select the right version f_{final}. The clauses rename, undefine and redefine has to be used in such a way that the behavior is the same as in *case 1* (see figures 6.4 and 6.5) .

- Case *3*: ***BC+D*A***, ***B+D*C*A***, ***D*B*C*A***. As it has been mentioned above, these situations lead implicitly to the sharing of features. To achieve replication of *f*, it is necessary to use clauses **redefine** and **rename** in one or several locations. In those cases (contrary to the situations encountered with ordinary inheritance) the clause **select** should appear in the definition of class A, because the diamond emerges there. A possible syntax is "**select** f_{final} in D " (see example 92).

- Case *4*: This represents all the other situations where class B or C is defined last and thus creates the diamond. Therefore, its definition should contain the clause "**select** f_{final} in D " at the right location.

Example 92 Selection of Replicated Features From a Foster Class

```
class B
feature
   f... -- Only name of feature is provided because
        -- it may apply to attribute, procedure, function, etc.
end
class C
feature
   f... -- Only name of feature is provided because
        -- it may apply to attribute, procedure, function, etc.
end
class D inherit
 B
   rename
     f as f_b
 end
 C
   rename
     f as f_c
 end
end
class A exherit
 B
 C
   select f in D
 end
 all
end
```

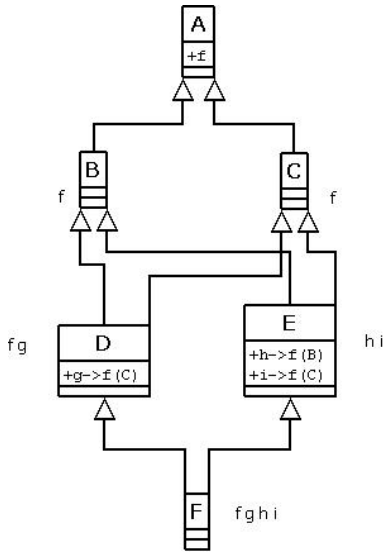


Figure 6.12: Select Problem

6.2.3 The "Select" Approach Does Not Solve All Ambiguities

The **select** clause in Eiffel is not consistent in all cases. An example of such a case can be found in figure 6.12. Reverse inheritance will not address these ambiguities as far as they are not addressed by ordinary inheritance.

The equivalent code can be seen in example 93:

In the class hierarchy presented above feature f will be inherited in class F through four different paths: *i)* [A, B, D, F], *ii)* [A, B, E, F], *iii)* [A, C, D, F] and, *iv)* [A, C, E, F]. In each class along the presented paths there are some versions of feature f renamed and selected. When the feature f of an F type instance is accessed there is no applicable version using such a combination. Using another combination the selection could be ambiguous, only a certain suitable combination makes the selection unique as it should.

The conclusion that can be drawn from this example is that in some special cases just the clause **select** and the feature name are not sufficient. Some other qualifications are still required.

6.3 Constraints on Exherited Features

6.3.1 Using the Frozen Keyword for Features

The impact of the keyword **frozen** has to be taken into account in several situations dealing with reverse inheritance. If we exherit frozen features from the subclasses then in the superclass they will be implicitly deferred and non-frozen as set with the previous rules. A feature may become frozen in the foster class only if it is moved up and is identical in all exherited classes. In single reverse inheritance a frozen attribute or method can be exherited as frozen in the superclass by moving it up. In multiple reverse inheritance, an attribute can be exherited as frozen only if it has the same type and it is frozen in all source classes. For methods the conditions are more strict since their implementations must be identical. A method can be exherited as frozen only if it has the same signature, the same body and the same frozen status in all exherited classes. Such a case seem to be an extremely rare event. We can conclude that moveup will be used to exherit also the frozen status of the feature if conditions hold.

Rule ***Exheriting Frozen Features*** *1.* A foster class *FC* cannot declare a feature as frozen except if the features exherited from all source classes are:

Example 93 Select Like Approach

```
class A
feature
  f ... -- Only name of feature is provided because
        -- it may apply to attribute, procedure, function, etc.
end
class B
 inherit A
end
class C
 inherit A
end
class D
 inherit B
  select f -- when f is called through a variable of type A
  end
 inherit C
  rename f as g
  end
end
class E
 inherit B
  rename f as h
  end
 inherit C
  rename f as i
  select i -- when f is called through a variable of type A
  end
end
class F
 inherit D
  select g -- when f is called through a variable of type B
  end
 inherit E
  select h -- when f is called through a variable of type C
  end
end
```

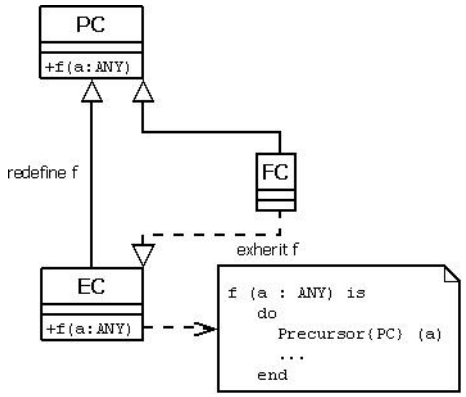


Figure 6.13: Main Configuration When Using the Precursor Keyword

1. attributes of the same type and are declared themselves as frozen;

2. methods with exactly the same signature and the same implementation and declared themselves as frozen.

Rule ***Exheriting Frozen Features*** *2.* A feature which is frozen in a subclass may always be exherited as a non-frozen feature in the foster class.

Rule ***Exheriting Frozen Features*** *3.* An exherited frozen feature which cannot be moved up from the subclass into the foster class will be always non-frozen in the foster class.

6.3.2 Impact of the precursor Keyword

A subtle point about reverse inheritance and dynamic binding is the behavior of the **Precursor** keyword in methods. Let us suppose that method f was inherited into class EC from a superclass PC, where it is effective and was redefined in EC. In that case, it is possible to call the inherited version of PC from the new version of EC using the **Precursor** keyword. The class configuration is pointed out in subsection 6.1.3 and the particular situation is depicted in figure 6.13.

Let us suppose that FC is inserted "in the middle", inherits from PC and exherits from EC. There are four different cases:

1. **The implementation of f is the one of PC**. This means f is inherited from PC and not redefined in EC and f is exherited from EC implicitly undefined. Thus, the precursor version is the same as without reverse inheritance, and there is no problem.

2. **The implementation of f is the one of EC**. Feature f is exherited from EC into FC and is undefined when inherited from PC. This means that the redefinition of f is effectively moved from EC to FC, and again there is no problem.

3. **The implementation of f is the one of FC**. Feature f is redefined in FC either when it is exherited from EC (and undefined when inherited from PC) or when it is inherited from PC (and undefined implicitly when exherited from FC) or both. In this case the **precursor** in the version of EC must be qualified in order to call the version from the parent class PC.

4. **Feature f is undefined in FC**. This happens if f is undefined when inherited from the parent class PC and when it is exherited from the exherited class EC. In this case **precursor** must be qualified with the name of the parent class PC since there are two method versions available.

Example 94 Catcall Example

```
class POLYGON
feature
 add_vertex() is do ... end
end
class RECTANGLE
inherit
 POLYGON
  export {NONE} add_vertext
 end
end
```

Example 95 Exportation and Exheritance

```
class EC1
 feature f {C1, C2, C3} is do ... end
end
class EC2
 feature f {C1, C2, C4} is do ... end
end
foster class FC
 exherit
  EC1
  EC2
  all
  export {C1, C2} f
end
```

6.3.3 Export and Exheritance

In ordinary inheritance each feature has a list of client classes which can access it through a qualified call. This list may be incrementally specified in the feature declaration clause, in the new export inheritance clause or in the feature redefinition clause [Mey02, Int06]. To export a feature to a client class means also to export that feature to all its descendants. This means that the export list for a feature cannot be diminished in the subclasses, the feature may add classes to the client list but it cannot remove them.

Allowing to diminish the list of clients for an inherited feature in the subclasses would make possible to hide that feature from descendants. Such an approach is senzitive to the polymorphic catcalls [Mey97]. CAT means Changing Availability of Type and catcalls may be caused by several situations like covariance, genericity or descendant hiding. Such a situation is depicted in example 94. The *add_vertex* method is hidden in *RECTANGLE* subclass and thus a *POLYGON* variable referring a *RECTANGLE* instance will fail in calling the *add_vertext* feature since it is unavailable.

We propose to provide symmetrical rules for export in reverse inheritance. In example 95 class *A* exherits feature *f* which in class *EC1* is exported to classes *{C1, C2, C3}* and in *EC2* is exported to classes *{C1, C2, C4}*. In foster class *FC* the export list of feature *f* contains the subset of the common client classes *{C1, C2}*. Extending the client class list in the foster class *{C1, C2, C3, C4}* would allow clients like *C3* respectively *C4* to access classes *EC2*, respectively *EC1* through polymorphic calls.

Rule ***Export List of the Exherited Features.*** In a foster class the export list for an exherited feature can be kept the same or diminished as the subset of the original common clients of the subclasses for that feature.

There are several reasons why we should keep or reduce the export list for an exherited feature. The

Example 96 Exheriting Creation Procedures

```
class EC1
 create make, default_create, build
 ...
end
class EC2
 create make, build, construct
 ...
end
foster class FC
 exherit
  EC1
  EC2
  all
 create make, build
end
```

first argument refers to the consistency and symmetry between reverse inheritance and ordinary inheritance. If in ordinary inheritance the export list in the subclass must be kept or enlarged with clients, reverse inheritance must keep as such or reduce the list of clients for an exherited feature. The second reason is related to avoiding catcall type polymorphism problems. The third reason is to avoid creating back doors for unauthorised clients to use classes.

6.3.4 Exheriting Creation Procedures

It does not seem that creation procedures should be handled in a special way. Ordinary inheritance has no effect over the creation procedures, so reverse inheritance should behave the same way. As foster classes implicitly exherit methods as deferred, these classes will be deferred also. In a deferred class it makes no sense of talking about creation procedures. Still, creation procedures can be exherited (or moved) as ordinary features. Sometimes it is a good idea to exherit creation procedures as deferred, because they can be used as any other procedure, but not for object creation purposes. If a creation procedure can be moved into the foster class, then it can be used as a regular feature or it can be added to the creation procedure list of that class.

In order to exherit creation procedures, first they have to be exherited as regular features. Then, we have to list them in the create section of the foster class if the corresponding features in subclasses were creation procedures also. In example 96 features like *make* and *build* can be exherited. Because the exherited features in the subclasses are all creation procedures then they can be listed or not as creation procedures in foster class *FC*.

Rule ***Exheritance of Creation Procedures.*** In a foster class the exherited creation procedures must have an implementation from one of the subclasses and all the corresponding features in the subclasses must be listed in their procedure creation list.

6.3.5 Exheritance of an Attribute with Assign Clause

There are attributes which have an assign clause attached and it is necessary to study their meaning when they are exherited. If a class needs to exherit the attribute along with the clause, it is necessary to exherit the setter method also. Otherwise the attribute will be exherited in read-only mode and the attribute may not be modified directly by clients through an assignment. Because it seems the more natural case, by default in single reverse inheritance, the assign method will be exherited with the attribute. It should be the same in the case of multiple exheritance when all subclasses have an assign method for this attribute. Otherwise, only the attribute should

Example 97 Exheritance of an Attribute with Assign Clause

```
class EC1
feature
  x:INTEGER assign put_x
  put_x(p:INTEGER) is do x:=p end
end
class EC2
feature
  y: INTEGER assign put_y
  put_y(p:INTEGER) is do y:=p end
end
foster class FC exherit
 EC1
  rename
   x as z,
   put_x as put_z
 end
 EC2
  rename
   y as z,
   put_y as put_z
 end
 all
end
```

be exherited. The syntax must allow to specify an assign method in the foster class through an attribute redefinition.

Rule ***Default Handling of Assign Clause***. By default, in single reverse inheritance, the assign method is implicitly exherited along with the corresponding attribute. It is the same in multiple exheritance if all subclasses have an assign method for this attribute.

In example 97, both attribute x of class *EC1* and attribute y of class *EC2* have an assign method, respectively *put_x* and *put_y*. Both attributes x and y as well as their assign methods (*put_x* and *put_y*), are renamed respectively into z and *put_z*. This means that if the attribute is exherited from different subclasses and its assign methods too, the exherited attribute will have automatically attached a deferred assign method in the superclass.

6.3.6 Exheritance When There is an Alias

Aliases represent a part of the renaming mechanism of Eiffel. If we allow to add an alias to a feature which is exherited, we may not modify the behavior of already existing descendant classes even if there is a new synonym for an existing feature (see figure 6.14). With ordinary inheritance, to an inherited feature with alias one can keep the original aliases and add new aliases or remove some of them [Mey02]. With reverse inheritance we keep the same philosophy: a feature can be exherited with all the aliases if they are the same in all subclasses (which is not a very probable case), or some of them can be removed, or new aliases can be created in the foster class. The behavior of the subclasses will be not affected if in the foster class an alias is added, because the renaming mechanism does not affect the semantics of an inherited or exherited feature. The difference between a feature and a feature name is a concept that is kept in the semantics of reverse inheritance, thus it respects the philosophy of the language.

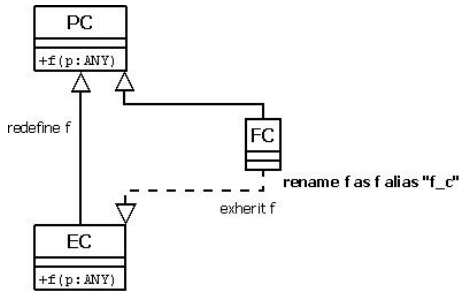


Figure 6.14: Adding an Alias When Exheriting

In figure 6.14 class *FC* is inserted in the middle of the hierarchy created by *PC* and EC. Feature f from class *EC* is exherited into class *FC* where *f* gets a new alias. If the aliases for feature *f* in class *FC* are newly added and different from the ones exherited from *EC* then in order to show that the semantics of aliases in context of reverse inheritance is consistent, we can consider that the new aliases of feature *f* in class *FC*, are implicitly removed in class *EC*. The new aliases of feature *f* from class *FC* can be used in any new subclass of *FC*, for instance.

Rule ***Aliases and Reverse Inheritance.*** The renaming mechanism, which includes the aliases mechanism, does not affect the semantics of reverse inheritance.

6.3.7 Exheriting Obsolete Features

Obsolete features are those old features in a class which are meant not to be used and which might be removed in the next releases of the class hierarchy. The first natural reaction would be not to allow the exheritance of such features, which may be no longer needed. Still, there are some good reasons to allow performing exheritance on them. If the class hierarchy is reused in a new context and no further evolution of that hierarchy is needed, then it is acceptable to exherit the obsolete features as they are desired, without any restrictions. This reason is motivated by the high degree of reuse which is intended to be outlined in the philosophy of reverse inheritance. If the exheritance of such features is necessary, it is recommended to adapt those features when exheriting them (rename, redefine, adapt, undefine), otherwise, this means that they are not considered as obsolete features and this leads to inconsistency. One of the main concepts of reverse inheritance philosophy is to favour feature reuse at the highest level possible. This is why obsolete features are allowed to be exherited without any restrictions, but a warning is issued.

6.3.8 Exheriting Once Features

Once features in Eiffel are executed at one moment after which the other several calls are ignored. If functions are involved, the first computed result is returned at each call. The **once** mechanism can be used for smart initializations or shared objects (like the Singleton design pattern [GHJV97]). There is a possibility to use this mechanism together with once keys allowing the possibility of selecting several behaviors: once for each instance, once for each thread, once for each process, once controlled by a user defined key.

It seems very natural to exherit **once** features like any ordinary features. The behavior of the feature, whether the body is executed or not, do not interfere with the mechanism of reverse inheritance. Sometimes it may be useful to have in the foster class a deferred feature which corresponds to some **once** or non-once features in the subclasses. The **once** keyword affects only the behavior of features at runtime and not the architecture of the object-oriented system.

In example 98 it is presented a combined case of exheriting once features. Features *init*, *setup* and *initialize* are exherited as a deferred feature *start* in the foster class. There can be noticed

Example 98 Exheriting Features of Type once

```
class EC1
 feature init once ... end
end
class EC2
 feature setup once("OBJECT") ... end
end
class EC3
 feature initialize is
 do ... end
end
foster class FC
 exherit
  EC1
   rename init as start
   undefine start
  end
  EC2
   rename setup as start
   undefine start
  EC3
   rename initialize as start
   undefine start
  end
  all
 feature start is deferred end
end
```

that feature *init* is not using any once key, so implicitly is set on *PROCESS*[2], the once key of feature *setup* is explicitly set up on *OBJECT*, while feature *initialize* is not a once feature. If it is decided that a once feature should be moved into the foster class then it is moved together with the once status.

Rule ***Exheritance of Once Features.*** The **once** mechanism does not affect the semantics of reverse inheritance.

6.4 Constraints on Foster Classes

6.4.1 Using the Frozen Keyword

In Eiffel the frozen keyword applied to a class will restrict that class from being inherited, thus having descendants. So, the downward evolution of the class is prohibited. If a class is declared as frozen it means that the designer has special reasons and he does not want to allow that class to be extended and its features to be redefined.

To preserve the symmetry principle of the language, we should allow foster classes like ordinary classes to be declared as frozen and thus to prohibit their upward evolution. This means that a frozen foster class cannot have new foster classes on its top. On the other hand a foster class which is declared as frozen in the context of ordinary inheritance can be extended by ordinary inheritance, so it can have new subclasses.

6.4.2 Using the Obsolete Keyword

An obsolete class in Eiffel is a class which does not meet the current standards and which may be erased in one of the new releases of the software. By reverse inheritance some features could be exherited into a new class which represents the part of the class that may be reused in the future. There is no reason for these features to be marked as obsolete. The default behavior of Eiffel when using such classes is to issue warnings. Thus, in the context of reverse inheritance, exheritance from obsolete classes is permitted, but the programmer is warned.

6.5 Summary

In the current chapter were presented several ideas about how to handle the special cases in which a class is the target of both ordinary and reverse inheritance. In a some general class configurations the effect of combining inheritance and exheritance clauses was experimented in order to obtain: a new version of the feature, a deferred feature, the inherited version, the exherited version. The problem is to get only one implementation for the features having multiple seeds due to the arrival of the foster class. The solution is based on the idea to undefine the conflicting implementation coming from the foster class.

The case of amphibious features is analysed, the features which are both inherited from a parent and exherited from an exherited class in the same foster class. The most important property of those features is that they cannot be unselected from exheritance. The superclasses of the foster class cannot bring new features except redefining a subset of existing ones. This restriction is necessary in order to preserve the non-destructive property of reverse inheritance class relationship. Foster classes can be built on the top of other foster classes, the only drawback is related to the lack of visibility for the exherited features when implicitly selected.

It is interesting though to analyze the configuration of a multiple inheritance diamond when classes are created in several orders. In most of the cases it is necessary to use reverse inheritance. Sub-cases were analyzed in the experiment of sharing and replicating features. When sharing features, it is natural to use only the inheritance/exheritance clauses. When replicating features in

[2]In the standard of ECMA [Int06] different variants of once are allowed: once per process, once per thread and once per object.

the diamond, which is done by renaming in all the combinations of ordinary and reverse inheritance the problem of feature selection in dynamic binding occurs. The feature selection mechanism from ordinary inheritance is used "a posteriori", meaning that the decision for the selection of a feature is taken in the latest built foster class, representing the base class of the hierarchy. The difference between the original selection mechanism and the one dedicated to reverse inheritance is the place for the specification of the "early" class to which the dynamic binding problem refers. Also, the effect of the **select** keyword is analyzed in the context of ordinary inheritance. The conclusion drawn is that in more complicated class configurations the mechanism does not allow the desired feature selection when dynamic linking problems occur. This problem is not addressed by ordinary inheritance, so will not try to solve it with reverse inheritance.

The exherited features may need several decisions regarding their properties like: frozen, precursor, export, creation nature, assign attachments, aliases, obsolete, once. Identical frozen candidates can be moved up in the foster class with the same frozen status. In practice such a situation is more probable for attributes than for methods. Another interesting issue is the one related to unqualified precursor calls initiated in the exherited classes. Such calls become ambiguous at the arrival of the foster class, so the solution is to make them explicit by attaching the name of the class to identify the called feature version. The exherited features are exported to the set of common clients. Exherited creation features must have implementations and all their candidates must be creation features too. Attribute assign clauses are exherited implicitly as long as their assign methods are exherited. Aliases are kept in the semantics of renaming in the context of reverse inheritance. Obsolete candidate features are exherited but a warning is issued. Once features are exherited as any regular features being orthogonal on the reverse inheritance class relationship. For foster classes we considered to analyze only the frozen and obsolete statuses. Frozen foster classes are designed quite symmetrically with frozen ordinary classes: the former prohibits upward class constructions while the latter prohibits downward hierarchy evolution.

Part III

Implementation

Chapter 7

Description of the Implementation

Part III of the thesis, starting with this chapter, presents the **operational semantics** of reverse inheritance in order to convince the reader about the feasibility of this class relationship. We propose an implementation solution for the reverse inheritance concept in the context of Eiffel language upon the **formal semantics** described in part II. The implementation is a full solution which offers the programmer a way to reuse already existing classes by reverse inheritance and to compile the resulted object-oriented system into an executable binary. The prototype stands as a proof of concept in order to demonstrate that the proposed class relationship is feasible and it can be used practically, on class hierarchies, facilitating their reuse in different contexts. The extended language will be referred to as **RIEiffel**.

Several Implementation Approaches When adding an extension to an already existing programming language there are several implementation choices. In the next subsections we propose three choices and we analyse their advantages and drawbacks, motivating the choice selected for implementation.

Modifying the Eiffel Compiler Apparently, a simple and straightforward idea is to modify the Eiffel compiler, including the new language extension and thus providing the executable of the object-oriented system. Such an approach implies writing the code expressing the semantics of reverse inheritance together with the rest of the compiler code. Thus the visibility of the reverse inheritance operational semantics would be diminished. On the other hand, it could be very difficult to prove that there are no deviations from the philosophy of the language and that illegal class constructions are not allowed. Also, such a task would be too big for a PhD research project. One more reason not to choose such an approach is that most Eiffel compilers do not generate platform native binary code, but C or Java [AG00] sources instead.

Generating C Source Code Another choice, which is used also by some Eiffel compilers, is to generate intermediary low level code which can be compiled afterwards by a platform independent compiler. For example ISE Eiffel [Sof08] compiler generates C code which is compiled further by gcc [SF08] on Unix/Linux [TOG08] or by Borland C [BI08] or Microsoft Visual C++ [Cor08b] on Microsoft Windows [Cor08a] platforms. Such an approach benefits from the fact that the resulted code is cross platform compilable, but would still lack in reverse inheritance semantics visibility.

Generating Eiffel Source Code Using Model Transformation Our chosen implementation solution is based on the fact that class hierarchies built with reverse inheritance allow the same facilities as those built with ordinary inheritance, but in different orders of construction, and that the resulted class hierarchies have the same semantic properties. The main idea of the implementation is to translate class hierarchies having both reverse inheritance and ordinary inheritance into semantically equivalent classes having just ordinary inheritance. This solution involves (see

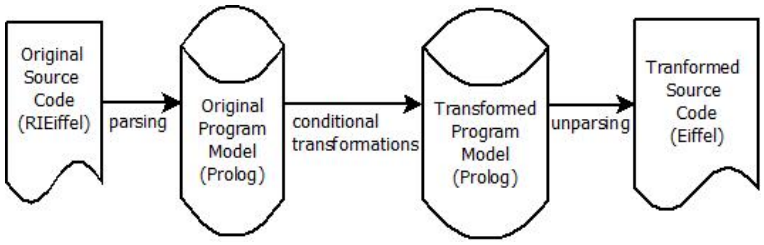


Figure 7.1: Generating Eiffel Source Code

figure 7.1): i) a translation from RIEiffel source into a Prolog model; ii) a transformation of the Prolog model to an equivalent Prolog model where the reverse inheritance facts are replaced with semantically equivalent ordinary inheritance facts; iii) a model translation into pure Eiffel source code compilable by an ordinary compiler. Thus the reverse inheritance semantics transformations are expressed using Prolog rules and the output is a pure Eiffel class hierarchy. Next, we define several notations related to the transformed items previously mentioned.

Original Source Code The **original source code** is the RIEiffel code which contains reverse inheritance and also ordinary inheritance. This code includes classes from different libraries which probably were developed in different contexts.

Original Prolog Model The **original Prolog model** is a factbase created by parsing the original source code. The original source code and the original Prolog model are semantically equivalent. Still some details like code indentation and organization are missing from the model, but these aspects do not change the logic of the original code. The model contains facts dealing with both reverse and ordinary inheritance. This model is subject for transformation.

Transformed Prolog Model The **transformed Prolog model** is obtained after transformations are applied on the original Prolog model. The transformed and original Prolog models are semantically equivalent. All reverse inheritance related facts are replaced with ordinary inheritance facts through transformations performed on the original Prolog model.

Transformed Source Code The **transformed source code** is obtained by reverse engineering from the transformed Prolog model. This code contains only ordinary inheritance and is compilable with an ordinary Eiffel compiler.

We have to state that this solution is just one implementation solution which requires the source code of the reused classes but its advantage is that it offers the possibility of proving the equivalence of the two class hierarchies and thus the consistency of the reverse inheritance semantics.

7.1 Eiffel Reverse Inheritance Reification in Prolog

In this section we present the Prolog model of reverse inheritance class relationship. As the new class relationship semantics cannot be isolated from the rest of the language, we will have to model also the pure Eiffel language elements. In order to be able to perform changes on the Eiffel code and to obtain a pure class hierarchy we need a model which can be changed easily. A Prolog fact base operated by conditional transformations represent a good choice for the implementation, as we will see in the next sections. The modelling of language entities through Prolog facts will be referred further as **reification**.

7.1.1 Reification of the RIEiffel Language

In order to integrate reverse inheritance class relationship in the Eiffel programming language, first we augment its grammar (see appendix A) with the elements of reverse inheritance. For the implementation we chose the grammar of the Eiffel GOBO library [Bez07]. Each grammar rule is analyzed and represented in the model by Prolog parameterised clauses.

The factbase is designed like a relational database. Usually, each fact has a unique key, which is represented by the first argument of that fact. Facts having the same name may be considered belonging to the same table. The identifiers of the facts having same names represent the primary keys of the table. Some facts model entities which have parent entities. This type of relations are represented by identifiers which are second arguments in the clause and they refer to parents. The links between the clauses are based on these identifiers. Most of the rules refer to their parent rule. For example the relation between a class declaration and its cluster is modelled as done in subsection B.1 of appendix B. Some rules have navigation capabilities in both ways. For example, in subsection B.5 of appendix B, when modelling formal arguments, we need to store their order, so for that we use a fact having a list as parameter. Also, each formal argument is modeled by a fact which keeps a link to the parent list. So the list points to its children (formal arguments) as well as the children point to the parent list (formal argument list). Some clauses have no own identifier at all, since they denote some optional single attribute of another clause. For example the **deferred** keyword for a class will be modelled using an attribute clause which will refer to the feature and without any other information in subsection B.5 of appendix B.

A special case is represented by modelling expressions in subsection B.8 of appendix B, where expressions use a downward referencing philosophy: each operator, unary or binary, will refer to its children. This decision was taken in order to reduce as much as possible the number of facts in the metamodel, thus decreasing its complexity. Another remark must be noted about different facts which have to be handled in a uniform manner like instructions. For such kind of facts we have modeled Prolog rules having the same name, one for handling each concrete instruction like: *creation*, *assign*, *conditional*, *multiBranch*, *loop*, *debug*, *check*.

7.1.2 Reification of the Exheritance Branch and Feature Selection Clauses

We will present in detail one fragment of the RIEiffel model, namely the reification of exheritance branches and exheritance candidate feature selection clauses. All fact arguments beginning with "#" symbol represent identifiers, having the behavior of indexes or keys in a database table.

The exheritance class relationship is expressed between the foster class and the exherited classes as types or instantiated classes when they are generic:

```
exheritance(#id,#classDecl,#classType).
```

- *#id* is the primary key;
- *#classDecl* is the identifier of the current, source class;
- *#classType* is the identifier of the exherited class type.

The redefinition in the context of reverse inheritance belongs to an exheritance branch like in ordinary inheritance:

```
redefine(#id,#exheritance,#featureDecl).
```

- *#id* is the primary key;
- *#exheritance* is the identifier of the exheritance branch where the redefine resides;
- *#featureDecl* is the identifier of the redefined feature.

The adapt and moveup clauses are linked to the exheritance class relationship and to a feature:

```
adapt(#id,#exheritance,#featureDecl).
moveup(#id,#exheritance,#featureDecl).
```

- *#id* is the primary key;
- *#exheritance* is the identifier of the exheritance branch;
- *#featureDecl* is the identifier of the feature to be adapted.

The exheritance selection mechanism is presented next:

```
selectExherit(#id,#exheritance).
```

- *#id* is the primary key;
- *#exheritance* is the identifier of the exheritance branch;

The feature which may be selected in the exheritance selection mechanism is next. One exheritance clause may have multiple features to be selected.

```
selectExheritFeature(#id,#selectExherit,#featureDecl).
```

- *#id* is the primary key;
- *#selectExherit* is the identifier of the parent fact;
- *#featureDecl* is the identifier of the feature to be selected.

Descendant class chains can be constructed using the following clause:

```
descendantQualification(#id,#selectExherit or #descendantQualification,#classDecl).
```

- *#id* is the primary key;
- *#descendantQualification* refers either to the *selectExherit* fact or to the next *descendantQualification* in the class chain;
- *#classDecl* is the identifier of the class in the chain.

The export declarations in the context of reverse inheritance are attached to the class and not to the inheritance clause, this is why they are modelled separately. Between a feature and export client class there is a many to many relationship (multiple features can be exported to multiple classes):

```
exportExherit(#id,#classDecl).
```

- models an export statement;
- *#id* is the primary key;
- *#classDecl* is the foster class identifier the export belongs to.

The classes which may be linked to an export statement are modelled next:

```
exportExheritClass(#id,#exportExherit,#classDecl).
```

- *#id* is the primary key;
- *#exportExherit* is the identifier of the export statement;

- *#classDecl* is the identifier of the class participating in the export statement.

Exported features are attached to the export statement by the following fact:

```
exportExheritFeature(#id,#exportExherit,#featureDecl).
```

- *#id* is the primary key;
- *#exportExherit* is the identifier of the export statement;
- *#featureDecl* is the identifier of the feature that is exported.

If one desires to export all features of a class the following clause must be attached to the export clause:

```
exportExheritFeatureAll(#id,#exportExherit).
```

- #id is the primary key;
- #exportExherit is the identifier of the export statement.

The selection mechanism of exheritance allows to select or deny a set of specific features through the following facts:

```
onlyFeature(#id,#classDecl,#featureDecl).
exceptFeature(#id,#classDecl,#featureDecl).
```

- *#id* is the primary key;
- *#classDecl* is the identifier of the class hosting the feature selection clauses;
- *#featureDecl* is the feature in the exherited class.

The selection mechanism can be set to select all exheritable features or no features at all, in order to create a new type:

```
allFeature(#classDecl).
nothingFeature(#classDecl).
```

- *#classDecl* is the identifier of the class hosting the feature selection clause.

7.1.3 Metamodel Validity Rules

It is very natural to validate the reverse inheritance metamodel by a set of rules in order to be able to check its consistency before the transformation. We intend to define only the validity rules regarding the reverse inheritance related mechanisms: feature selection, redefinition, adaptation, selection, assertion. From the technical point of view, validity rules are expressed as Prolog predicates operating on the model factbase. For some rules we present their Prolog code.

Type Checking

Any RIEiffel factbase is checked against a formal metamodel where all facts are described together with their arguments. Each argument has a type information allowing type checking. For example, the class declaration fact has as fourth argument a list of formal generic identifiers. The type checker will verify that each fact corresponding to that set of identifiers is a formal generic and not some other fact.

In example 99 we show the node definition for a class declaration. The definition contains information about the language name (in this case *RIEiffel*), node name (*classDecl*) and each fact parameter. The first argument is the unique identifier of the fact, named *id*, its multiplicity is

Example 99 RIEiffel Type Rules

```
%classDecl(#id,#cluster,'ClassName',[#formalGeneric,...]).
ast_node_def('RIEiffel',classDecl,[
 ast_arg(id,             mult(1,1,no),  id,   [classDecl]),
 ast_arg(parent,         mult(1,1,no),  id,   [cluster]),
 ast_arg(className,      mult(1,1,no),  attr, [atom]),
 ast_arg(formalGenerics, mult(1,*,ord), id,   [formalGeneric])
]).
```

Example 100 Single Selection Rule

```
checkSingleSelectionMechanism(FosterClassId):-
 exists(allFeature(FosterClassId))->
  not(exists(onlyFeature(_,FosterClassId,_));
  exists(exceptFeature(_,FosterClassId,_));
  exists(nothingFeature(FosterClassId))),
 !.
checkSingleSelectionMechanism(FosterClassId):-
 exists(onlyFeature(_,FosterClassId,_))->
  not(exists(allFeature(FosterClassId));
  exists(exceptFeature(_,FosterClassId,_));
  exists(nothingFeature(FosterClassId))),
  !.
checkSingleSelectionMechanism(FosterClassId):-
 exists(exceptFeature(_,FosterClassId,_))->
  not(exists(allFeature(FosterClassId));
  exists(onlyFeature(_,FosterClassId,_));
  exists(nothingFeature(FosterClassId))),
  !.
checkSingleSelectionMechanism(FosterClassId):-
 exists(nothingFeature(FosterClassId))->
  not(exists(allFeature(FosterClassId));
  exists(exceptFeature(_,FosterClassId,_));
  exists(onlyFeature(_,FosterClassId,_))),
  !.
```

one to one with the fact, not ordered, its kind is identifier and its type is *classDecl*. The second argument is named *parent*, the current fact has a one to one multiplicity relation with the cluster fact, its kind is identifier, and the type is *cluster*. The third argument is named className, has a one to one multiplicity relation with the attribute, its kind is attribute and the type is *atom*. The fourth argument is named formalGenerics, the current fact has a one to many multiplicity with the formal generic facts, it is an ordered set (*ord*), its kind is identifier and the type is *formalGeneric*.

Exherited Features Validity Rules

Regarding the aspect of feature factorization, several validity rules must hold on the factbase model.

Single Selection Rule The first rule from example 100 verifies that only one exherited feature selection mechanism of **all**, **only**, **except**, **nothing** will be used in a foster class.

Exheritable Selection Rule All explicitly selected features using **only**, **except** keywords from the exherited classes must exist and must have compatible signatures in all the exherited classes. In other words it does not make sense explicitly selecting or unselecting sets of features which are not exheritable.

Non-conflicting Selection Rule The selected features must not generate conflicts with the new features of the foster class. This means that we cannot select a feature for exheritance while it already exists in the foster class. In such a case the foster class should be designed with a redefinition clause.

Immediate Feature Selection Rule The exherited features must exist locally in the subclasses and not inherited by some ancestor which is not the parent of the foster class. Thus, we avoid creating new links between already existing classes.

Exherited Feature Redefinition Validity Rules

In the case of feature redefinition, performed in the foster class, several rules must be applied:

- for each redefined set of features in the exherited classes, a new feature must exist in the foster class. This means that the set of *redefineExherit* clauses must point to features present in all exherited classes, having the same final name relative to the foster class.
- the redefined set of features in the exherited classes must have covariant signatures with the newly defined feature in the foster class;
- a redefined attribute in the foster class cannot have corresponding method candidate features in the subclasses;
- a redefined attribute from the foster class cannot have corresponding deferred candidate features in the subclasses.

Exherited Feature Adaptation Validity Rules

Adapted features obey to the following rules:

- an adapted feature must have formal arguments or return type, otherwise there is nothing to adapt;
- one feature listed in the adaptation clause must have at least one conversion sequence in the foster class corresponding to an exherited class;
- the signature of the conversion sequence must be identical with the one from the subclass;
- the new implementation signature must be equal to the one of the new features from the foster class.

Exherited Feature Implementation Migration Validity Rules

The **moveup** mechanism is driven by the following rules:

- in a foster class the **moveup** clause for an exherited feature set can be used only on one exheritance branch. It does not make sense selecting multiple implementations for the same feature to migrate in the foster class thus arising a conflict.
- the dependencies of the moved features must be provided in the foster class by exheritance or by redefinition;
- for one exherited feature it cannot be applied both moveup and adapt;
- a feature cannot be both moved up and redefined.

Exherited Feature Selection Validity Rules

In the case of repeated inheritance the selection of replicated features must satisfy the next rules:

- the selected feature must belong to one of the exherited feature sets;
- all the classes from the selection clause must be direct or indirect subclasses of the foster class enclosing the select declaration.

Formal Generics Validity Rules

Since the foster class and the exherited classes may or not be generic and genericity can be constrained and unconstrained, we will set rules for all possible combinations.

Non-generic Foster Class and Generic Subclasses When the foster class is non-generic, it will refer to the exherited classes through class types which may have attached actual generics. Considering this, we issue the following rules:

- all formal generics of exherited classes must be instantiated with type *NONE*. Since these actual generics will be lost in the equivalent class hierarchy based on ordinary inheritance, we decided to use a special instantiation of the subclasses with t*ype NONE*.
- not all subclasses must be generic.

Generic Foster Class and Generic Subclasses In this case the instantiation information for the exherited classes will be reused for instantiating the foster class. Thus the following rules apply:

- each formal argument of the foster class must instantiate a formal generic in all exherited classes;
- if the formal generics of the foster class are constrained then the instantiated ones from the exherited classes must have the same constraint;

Generic Foster Class and Non-generic Subclasses An invalid case arises when the foster class is generic (unconstrained or constrained), the subclasses are non-generic and there is no instantiation information available.

7.2 Software Instrumentation

In this section we present the several software tools representing the prototype modules. The **RIEiffel** prototype is a **complete solution** for the implementation of the reverse inheritance class relationship. The prototype accepts as input a reverse inheritance re-engineered project and produces the executable object-oriented system. The prototype is composed of several modules, each corresponding to one phase of the proposed solution (see figure 7.2).

7.2.1 The Eiffel to Prolog Translator

The **ETransformer** module converts the RIEiffel source files into a factbase stored as a Prolog file. The module is written in Eiffel and it was built from a generated parser. The parser was generated from the RIEiffel grammar listed in appendix A using the **gelex** (a version of Flex [Fou95] generating Eiffel code) and **geyacc** (a version of Yacc [Joh79] or Bison [Fou06] generating Eiffel code) tools from the **GOBO** library [Bez07]. The parsing process is applied on the whole Eiffel class universe including both client and system libraries classes. This is necessary since the analysis may require also information from the system library classes. The generated parser builds the abstract syntax tree (AST) of the input sources written in RIEiffel. Then the AST is visited and using a set of semantic actions, manually written, the Prolog facts are generated.

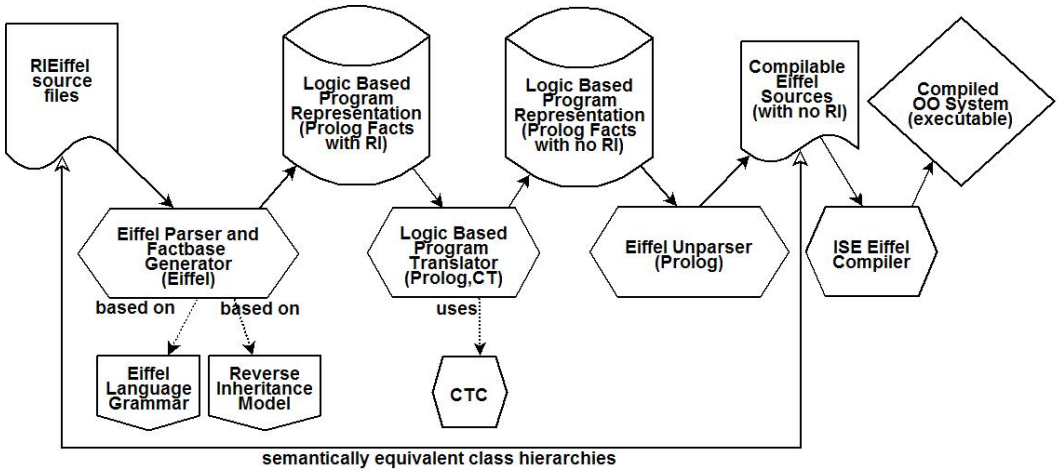


Figure 7.2: Software Instrumentation Overview

7.2.2 The Prolog to Prolog Translator

All the Prolog transformations which will be presented in section 7.3 are centralized in the **PTransformer** module of the prototype. The module takes as input the generated factbase and produces a new semantically equivalent factbase, which contains only pure Eiffel facts. The CTs are executed by the CTC interpretor [Kni06] which analyses them and performs the desired changes on the factbase.

The **PTransformer** module is tested automatically with two types of tests: structural and output. The structural tests consist in transforming a model input and comparing the output against a model oracle. The output tests compares the regenerated Eiffel code for the transformed models against an output source oracle.

7.2.3 The Prolog to Eiffel Translator

The transformed factbase must be translated back into pure Eiffel code in order to obtain the executable object-oriented system. The **Unparser** module of the prototype is responsible for this task. The regeneration is done in Prolog by unparsing rules for each type of facts from the factbase following the RIEiffel[1] grammar.

The unparsing process consists in checking the existence of facts and printing at output the related keywords and values following the rules of RIEiffel grammar. In example 101 we present one code fragment from the **Unparser** module.

As an observation, one can notice that we regenerate both Eiffel and RIEiffel code, because we output the adaptation parts for a feature if they exist. Such a regeneration is necessary for prototype testing and integration. Thus can be tested the equivalence between the initial source files and the model, by comparing the source with the unparsed ones obtained from the model.

7.3 Model Transformations

In this section we present the model transformations that must be performed on the factbase model representing Eiffel class hierarchies built with reverse inheritance in order to generate an equivalent model by eliminating the new class relationship.

[1] The grammar of RIEiffel includes the grammar of pure Eiffel. Any pure Eiffel source code will conform to the RIEiffel grammar.

Example 101 Unparsing Feature Declarations

```
unparseFeatureDecl(FeatureDeclId):-
 exists(featureDecl(FeatureDeclId,_,FeatureName)),
 tab(4),
 name(FeatureName,FeatureName2),
 writef('%s',[FeatureName2]),
 (exists(formalArguments(FormalArgumentsId,FeatureDeclId,_))->
  unparseFormalArguments(FormalArgumentsId);true),
 ((exists(typeMark(FeatureDeclId,TypeId))->
 (writef(':'),
  unparseType(TypeId)));true),
 ((exists(featureManifestConstant(FeatureDeclId,ManifestConstantId)),
   exists(manifestConstant(ManifestConstantId,Value,_))->
 (writef(' is %t',[Value])));true),
 ((exists(routine(RoutineId,FeatureDeclId))->
  unparseRoutine(RoutineId));nl),
 ((exists(attributeAdaptation(_,FeatureDeclId,_,_,_,_));
   exists(routineAdaptation(_,FeatureDeclId,_)))->
  unparseAdapted(FeatureDeclId);true).
```

Example 102 Conditional Transformation Structure

```
ct(
 name(arg1,arg2,...),
 condition(cond1,cond2,...),
 transformation(transf1,transf2,...)
).
```

7.3.1 Conditional Transformations

Conditional transformations (CTs) are defined and explained in [KK02, Kni06]. A CT is an abstraction composed of a condition and a transformation under a well defined set of formal rules. The CT based solution we are using in our program transformation is relying on:

- the representation of programs and models as logic factbases, as it was described in section 7.1;
- condition evaluation based on the declarative semantics of logic programs;
- information propagation via sets of substitutions computed for logic variables;
- the transformation of logic factbases.

Such an approach is applicable to arbitrary languages, models and artifacts. Example 102 presents the structure of a CT. Technically, it is defined as a Prolog fact called **ct**. It has three terms: the **name** term which is arbitrary, the **condition** term and the **transformation** term. The name may have attached a list of arguments representing the parameters of the CT. The **condition** term will have a sequence of facts representing the conditions, while the **transformation** term will have a sequence of predicates for creation and deletion of facts from the factbase. The predefined predicates for the factbase manipulation are named **add** and **delete**.

For example, the location of foster classes and the deletion of their foster attribute may be viewed in example 103. The name of the CT is *removeFoster*, the condition sequence calls two *exists* facts and the transformation uses the predefined **delete** predicate to erase facts from the factbase. The transformations will be applied to all sets of values selected by the condition sequence.

Example 103 Conditional Transformation Example

```
ct(
 removeFoster(FosterClassId),
 condition((
  exists(classDecl(FosterClassId,_,_,_)),
  exists(foster(FosterClassId))
 )),
 transformation(
  delete(foster(FosterClassId))
 )
).
```

The parameters of a CT can be of several types: input, output and input-output, like in any Prolog rule. The CTs communicate with each other through parameters and the propagation of information is determined by the operator. CTs are linked together in a sequence through special binary operators **ANDSEQ**, **ORSEQ** and **PROPSEQ**. The operators have two operands, right-hand side and left-hand side, which may be either CT or another operator.

ANDSEQ Operator

The ANDSEQ operator propagates the results forward from the first CT to the second one and also backward if necessary. The parameter value sets for which the condition fails in the second CT will be back-propagated to the first CT undoing its transformations. For example, a situation where such an operator is necessary is when we create a feature block within a foster class using a first CT while the second CT is responsible with the creation of the exherited features. If there is no feature to exherit then the creation of the feature block is cancelled due to the back-propagation nature of the ANDSEQ operator. As a consequence such an operator requires that the first operand must have output parameters which are connected to the input parameters of the second operand.

PROPSEQ Operator

The PROPSEQ operator propagates forward the values of the parameters from the first operand to the second one, but never cancels any transformations executed by the first one. For example, such an operator must be used when we compute the exherited features and only for them we want to create formal arguments.

ORSEQ Operator

The ORSEQ operator does not propagate anything from left-hand side operand to right-hand side operand like the previous two ones did. This operator just takes any propagated input from the context and feeds both inputs of the referred CTs. Such an operator can be used when the two CTs do not interact with each another. For example the deletion of exheritance clauses and the deletion of the foster class status are independent and they do not need to interact.

7.3.2 Main Conditional Transformation Diagram

In figure 7.3 we present the main CT tree performing the transformations. Because of complexity details the tree is not presented entirely but only its root and the subtrees as blocks. Each subtree will be presented in detail in the next subsections. On each operator node is marked the level of that node in the tree in order to have a better visualisation.

We visit the CT tree in preorder explaining the process at macro level. The *FeatureExheritance* CT subtree is responsible with the creation of exherited features and their signatures. The

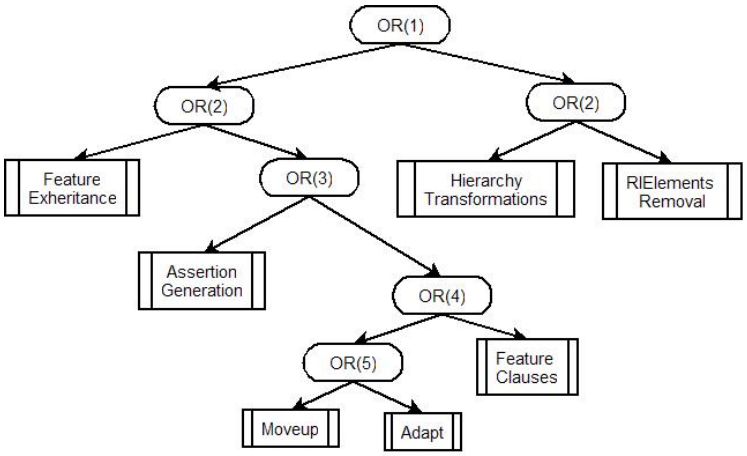


Figure 7.3: Main CT Diagram

AssertionGeneration CT subtree composes the combined preconditions and postconditions for the exherited features if possible. The *Moveup* CT subtree is in charge with the migration of a candidate feature implementation from the exherited class to the foster class and with its adaptation to the new context. The *Adapt* CT subtree deals with the generation of mediator methods to allow adaptation of features having different signatures and same semantics. The *FeatureClauses* CT subtree generates adaptation clauses for the newly exherited features. The *HierarchyTransformations* CT subtree changes the exheritance branches into inheritance ones and creates the necessary class types. Finally, the *RIElementsRemoval* CT subtree deletes all reverse inheritance related facts from the factbase like class keywords and feature selection clauses.

Another important aspect is the foster class order on which the transformations are executed. This order is important for foster classes having other foster classes on top of them. First the foster classes which are at the bottom are processed, then the next upper level until top foster classes are reached. The upward processing approach enables the propagation of exherited features from the bottom to top.

7.3.3 Feature Exheritance

In figure 7.4 we present the feature exheritance CT subtree. First we create some temporary maps for maps from redefined features to foster classes and from old features from exherited classes to redefined features within the foster class. These maps are created by *createRedefineFeatureInFosterClassMap* and *createOldFeature ToRedefinedFeatureMap* and will be used for later computation.

Next, we create a new feature block for the exherited features in the foster class using the *createExheritedFeatureBlockInFosterClass* CT. The *createExheritedFeaturesInFosterClass* CT from level 6 creates the exherited features in the foster class. Then deferred routines are attached to them. The map from candidate features to new features is computed by the *createCandidateToNewFeatureMap*. Such maps are useful in the computation of other CTs. In the process of exheritance new types may be scheduled for creation as it was explained in section 4.5. Such types are created by the *createTypesForNewFeatureInFosterClass* CT.

Finally, the formal arguments and the type marks are created. Formal arguments are created by the *createExheritedFeatureArguments* CT. The ordered list of formal arguments is set by the *createExheritedFeatureArgumentList* CT. Since the parent fact of the formal arguments is created after the children, we must update the children with their parent using the *updateExheritedFeatureArguments*. During the argument creation process some like types might need creation, which is done by the *createTypesForNewFeatureInFosterClass* CT. The process of type marks creation is

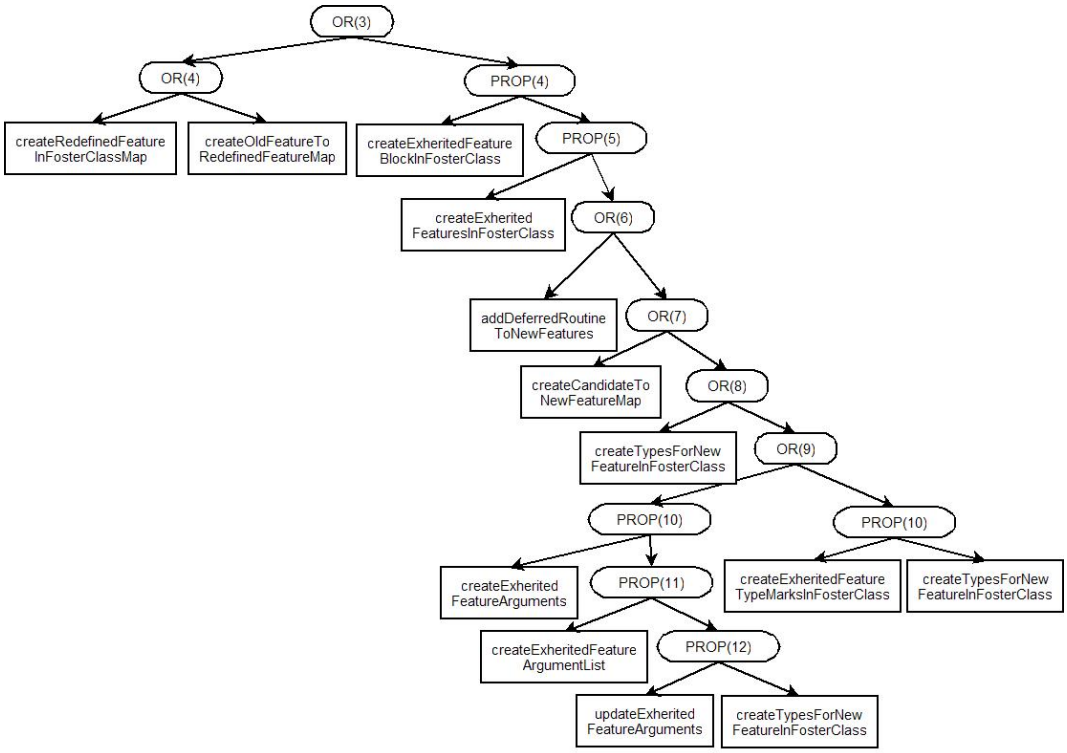


Figure 7.4: Feature Exheritance CT Subtree

done by a single CT *createExheritedFeatureTypeMarksInFosterClass* along with the type creation CT *createTypesForNewFeatureInFosterClass*. Next, we will present in detail the CTs which are most representative for the process.

Creating the Exherited Features in the Foster Class Since feature exheritance is one of the most challenging and complex parts of the implementation we will present the *createExheritedFeaturesInFosterClass* CT in detail in example 104. Each rule will be explained in detail in the next paragraphs. First, we compute the candidate feature set which contain tuples of features from the exherited classes having the same final name in the foster class. After this step the feature signatures are computed and types from corresponding positions are grouped together in order to compare them. Thus, we can decide whether they are compatible and if so, what is the representing type for the new feature in the foster class. In our representation, signatures are implemented as lists of type identifiers, having on the first position the return type and next the list of formal argument types.

For each exherited feature we perform several actions. First the feature is created, then a routine is attached to it and finally the routine is set as deferred.

Computing the Candidate Features The rule from example 105 computes the candidate features having the same final name in the foster class.

In example 105 we search for sets of features in the exherited classes. Features with the same final name will be taken from each exherited class to build a set. Such feature sets are candidates for exheritance. However, feature final name matching is not sufficient for exheritance, the exherited features require to also have compatible signatures. Signatures issues are discussed in one of the next paragraphs.

Example 104 Creating the Exherited Features in the Foster Class

```
ct(
 createExheritedFeaturesInFosterClass(FosterClassId),
 condition((
  computeCandidateFeatures(FosterClassId,FeatureName,ExheritedFeatureIdList),
  computeSelectedFeatures(FosterClassId,ExheritedFeatureIdList,SelectedFeatureIdList),
  concatenateFeatureSignatureTypeLists(SelectedFeatureIdList,TypeIdLists),
  createTypeIdListCorrespondence(TypeIdLists,CorrespondentTypeIdLists),
  maplist(selectRepresentantTypeId,CorrespondentTypeIdLists,RepresentantTypeIdList),
  once(exists(featureBlock(FeatureBlockId,FosterClassId)))
)),
transformation((
 new_node_id(FeatureDeclId),
 new_node_id(RoutineId),
 add(featureDecl(FeatureDeclId,FeatureBlockId,FeatureName)),
 add(routine(RoutineId,FeatureDeclId)),
 add(deferredFeature(RoutineId))
))
).
```

Example 105 Computing Candidate Features

```
computeCandidateFeatures(FosterClassId,FeatureName,FeatureList):-
 findall(
  (FosterClassId,FeatureName,FeatureList),
  computeCandidateFeatures0(FosterClassId, FeatureName, FeatureList),
  All
 ),
 sort(All,Sorted),
 member((FosterClassId,FeatureName,FeatureList),Sorted).
computeCandidateFeatures0(FosterClassId,FeatureName,FeatureList):-
 exists(foster(FosterClassId)),
 exists(exheritance(_,FosterClassId,ExheritedClassTypeId)),
 exists(classType(ExheritedClassTypeId,ExheritedClassId)),
 exists(classDecl(ExheritedClassId,_,_,_)),
 exists(featureBlock(FeatureBlockId,ExheritedClassId)),
 exists(featureDecl(FeatureDeclId,FeatureBlockId,_)),
 exheritedFeatureFinalName(FeatureDeclId,FosterClassId,ExheritedClassId,FeatureName),
 createExheritedClassList(FosterClassId,ExheritedClassIdList),
 forall(member(ExheritedClassId1,ExheritedClassIdList),
  exheritedFeatureFinalName(_,FosterClassId,ExheritedClassId1,FeatureName)),
 findall(FeatureDeclId1,
  exheritedFeatureFinalName(FeatureDeclId1,FosterClassId,_,FeatureName),
  FeatureList).
```

Example 106 Computing the Selected Features

```
computeSelectedFeatures(FosterClassId,ExheritedFeatureIdList,
 SelectedFeatureIdList):-
 findall(FeatureId,
  (member(FeatureId,ExheritedFeatureIdList),
   isFeatureSelectedForExheritance(FeatureId,FosterClassId)),
  SelectedFeatureIdList).

isFeatureSelectedForExheritance(_,FosterClassId):-
 exists(allFeature(FosterClassId)),
 !.
isFeatureSelectedForExheritance(FeatureId,FosterClassId):-
 exists(onlyFeature(FosterClassId,_))->
 exists(onlyFeature(FosterClassId,FeatureId)),
 !.
isFeatureSelectedForExheritance(FeatureId,FosterClassId):-
 exists(exceptFeature(FosterClassId,_))->
 not(exists(exceptFeature(FosterClassId,FeatureId))),
 !.
isFeatureSelectedForExheritance(_,FosterClassId):-
 not(exists(nothingFeature(FosterClassId))).
```

Computing the Selected Features From the set of exheritable features, we keep only the features which are selected for exheritance by the programmer. The selection can be performed by keywords like **all**, **nothing**, **only**, **except**.

In the implementation, a Prolog rule called *isFeatureSelectedForExheritance* is defined for each selection keyword. First the *allFeature* fact existence is tested for the given foster class. Next, if the previous rule fails, the *onlyFeature* fact existence is tested with foster class and feature identifiers. If the previous rule fails, then, the non-existence of the *exceptFeature* fact is checked, having the feature and foster class identifiers as arguments. Finally, the non-existence of *nothingFeature* fact is verified on the selected foster class. If there are no selection facts used the non-existence of *nothingFeature* will allow the selection of all candidate features. However, only one selection keyword may be used at one time.

Concatenating Feature Signature Type Lists After we found the sets of candidate features we concatenate their types in lists, resulting signatures. Such a signature may have one of the following forms:

```
01 [noReturnType, noFormalArguments]
02 [noReturnType, 101, 102, 103]
03 [101, NoFormalArguments]
04 [101, 101, 102, 103]
```

Line *01* contains a signature which has neither formal arguments nor return type. In line *02* we have a signature which has no return type but there are formal arguments having three type identifiers listed. In line *03* we have a return type on the first position but there are no normal arguments. Line *04* describes a signature which has return type *[101]* and three formal argument types *[101, 102, 103]*. We can notice that in our convention the return type identifier is on the first position in the list. Its absence is marked by a special atom named *noReturnType*. The formal arguments start on the list second position and take the next positions. The formal arguments absence is marked by the appearance of the *noFormalArguments* atom.

In example 107 we listed several rules dealing with feature signature manipulation.

Example 107 Concatenating Feature Signature Type Lists

```
createFeatureSignatureTypeIdList(FeatureDeclId,SignatureTypeIdList):-
 exists(featureDecl(FeatureDeclId,FeatureBlockId,_)),
 exists(featureBlock(FeatureBlockId,ExheritedClassId)),
 exists(classDecl(ExheritedClassId,_,_,_)),
 exists(classType(ExheritedClassTypeId,ExheritedClassId)),
 exists(exheritance(_,FosterClassId,ExheritedClassTypeId)),
 exists(foster(FosterClassId)),
 extractReturnTypeFromFeature(FeatureDeclId,ReturnTypeIdList),
 extractFormalArgumentTypesFromFeature(FeatureDeclId,
  FormalArgumentTypeIdList),
 append(ReturnTypeIdList,FormalArgumentTypeIdList,
 SignatureTypeIdList).
concatenateFeatureSignatureTypeLists([],[]).
concatenateFeatureSignatureTypeLists([FeatureDeclId|FeatureIdList],
 [SignatureTypeIdList|SignatureTypeIdLists]):-
 exists(featureDecl(FeatureDeclId,_,_)),
 createFeatureSignatureTypeIdList(FeatureDeclId,SignatureTypeIdList),
 concatenateFeatureSignatureTypeLists(FeatureIdList,SignatureTypeIdLists).
createTypeIdListCorrespondence(SignatureTypeIdLists,
 CorrespondingTypeIdLists):-
 maplist(length,SignatureTypeIdLists,LengthList),
 sort(LengthList,[MaxLength]),
 createTypeIdListCorrespondence0(SignatureTypeIdLists,1,MaxLength,
 CorrespondingTypeIdLists).
createTypeIdListCorrespondence0(_,Index1,Index2,[]):-
Index1 > Index2.
createTypeIdListCorrespondence0(SignatureTypeIdLists,
 Index1,Index2,[CorrespondingTypeIdList|CorrespondingTypeIdLists]):-
 Index1=<Index2,
 findall(TypeId,
 (member(SignatureTypeIdList,SignatureTypeIdLists),
 nth1(Index1,SignatureTypeIdList,TypeId)),
 CorrespondingTypeIdList),
 Index11 is Index1+1,
 createTypeIdListCorrespondence0(SignatureTypeIdLists,
 Index11,Index2,CorrespondingTypeIdLists).
```

Example 108 Creating Formal Arguments

```
ct(
createExheritedFeatureArgumentsInFosterClass(FosterClassId,FeatureBlockId,
 NewFeatureDeclId,ExheritedFeatureIdList,[_|RepresentingTypeIdList]),
condition((
 exists(featureBlock(FeatureBlockId,FosterClassId)),
 exists(featureDecl(ExheritedFeatureDeclId,FeatureBlockId,_)),
 selectFeatureForFormalArgumentExheritance(FeatureDeclId2,ExheritedFeatureIdList),
 exists(formalArguments(FormalArguments2,FeatureDeclId2,FormalArgumentIdList2)),
 member(FormalArgumentId2,FormalArgumentIdList2),
 nth1(Index,FormalArgumentIdList2,FormalArgumentId2),
 nth1(Index,RepresentingTypeIdList,RepresentingTypeId3),
 exists(formalArgument(FormalArgumentId2,FormalArguments2,FormalArgumentName2,_))
)),
transformation((
 new_node_id(FormalArgumentId3),
 add(formalArgument(FormalArgumentId3,FormalArguments2,FormalArgumentName2,
 RepresentingTypeId3))
))
).
```

The rule *createFeatureSignatureTypeIdList* takes as input parameter a feature identifier and returns its signature (the list of formal arguments and return types).

The rule *concatenateFeatureSignatureTypeLists* takes as input a list of features, computes for each feature its signature and returns a list of signatures. For example, suppose feature *701* has as signature list *[101, 101, 102, 103]*, feature *702 [101, 101, 102, 104]* and feature *703 [101, 101, 102, 105]*. The result of the rule will be *[[101, 101, 102, 103], [101, 101, 102, 104], [101, 101, 102, 105]]*.

The last rule *createTypeIdListCorrespondence* groups corresponding types having the same position in the lists in order to be compared. The effect applied on the previously obtained list is: *[[101, 101, 101], [101, 101, 101], [102, 102, 102], [103, 104, 105]]*. The intention of this arrangement is to facilitate the computation of the resulting types for the exherited feature in the foster class.

7.3.4 Exherited Feature Signatures Creation

Creating Formal Arguments

After the features are created in the foster class, their arguments must be created also: argument names are copied from one of the exherited classes candidate feature (if there is a feature implementation selected by **moveup**, that feature will provide the formal argument names), while their types are computed using a special algorithm which will be presented later in subsection 7.3.5.

After the creation of exherited features in the foster class, we have to complete them with the creation of formal argument and return type facts. In example 108 the formal arguments are created using the same names from the feature in the selected exherited class. The created formal argument facts have no reference to the parent fact and they are listed in the argument list of the selected subclass.

In example 109 we create the formal argument list fact which will refer the formal arguments created in the previous step. Iterating the argument list from the exherited feature in the subclass we can locate and build the new argument list for the foster class feature.

The third step of argument update, illustrated in example 110, consists in updating the parent of all created formal arguments and the like type if it is the case. Some like types require spe-

Example 109 Creating Formal Argument List

```
ct(
createExheritedFeatureArgumentListInFosterClass(FosterClassId,FeatureBlockId,
 NewFeatureDeclId,ExheritedFeatureIdList),
condition((
 exists(featureBlock(FeatureBlockId,FosterClassId)),
 exists(featureDecl(ExheritedFeatureDeclId,FeatureBlockId,_)),
 selectFeatureForFormalArgumentExheritance(FeatureDeclId2,ExheritedFeatureIdList),
 exists(formalArguments(FormalArgumentsId2,FeatureDeclId2,FormalArgumentIdList2)),
 findall(FormalArgumentId3,
 (
  member(FormalArgumentId2,FormalArgumentIdList2),
  exists(formalArgument(FormalArgumentId2,FormalArgumentsId2,FormalArgumentName,_)),
  exists(formalArgument(FormalArgumentId3,FormalArgumentsId2,FormalArgumentName,_)),
  FormalArgumentId2=\=FormalArgumentId3
 ),
 FormalArgumentIdList)
)),
transformation((
 new_node_id(FormalArgumentsId),
 add(formalArguments(FormalArgumentsId,ExheritedFeatureDeclId,FormalArgumentIdList))
))
).
```

Example 110 Updating Formal Arguments

```
ct(
updateExheritedFeatureArgumentsInFosterClass(FosterClassId,FeatureBlockId,
 NewFeatureDeclId,CorrespondentTypeIdLists),
condition((
 exists(featureBlock(FeatureBlockId,FosterClassId)),
 exists(featureDecl(ExheritedFeatureDeclId,FeatureBlockId,_)),
 exists(formalArguments(FormalArgumentsId,ExheritedFeatureDeclId,
  FormalArgumentIdList)),
 member(FormalArgumentId,FormalArgumentIdList),
 nth1(Index,FormalArgumentIdList,FormalArgumentId),
 nth0(Index,CorrespondentTypeIdLists,TypeIdList),
 exists(formalArgument(FormalArgumentId,_,FormalArgumentName,
  FormalArgumentTypeId)),
 computeLikeType(TypeIdList,FormalArgumentTypeId,FormalArgumentTypeId2)
)),
transformation((
 delete(formalArgument(FormalArgumentId,_,_,_)),
 add(formalArgument(FormalArgumentId,FormalArgumentsId,FormalArgumentName,
 FormalArgumentTypeId2))
))
).
```

Example 111 Creating Return Types

```
ct(
createExheritedFeatureTypeMarkInFosterClass(FosterClassId,FeatureBlockId,
 NewFeatureDeclId,[TypeIdList|_],[TypeId|_]),
condition((
 exists(featureBlock(FeatureBlockId,FosterClassId)),
 exists(featureDecl(ExheritedFeatureDeclId,FeatureBlockId,_)),
 computeLikeType(TypeIdList,TypeId,TypeId2),
 TypeId2\==noReturnType
)),
transformation(
 add(typeMark(ExheritedFeatureDeclId,TypeId2))
)
).
```

cial updates because like type exheritance requires the feature exheritance process to be finished completely.

Creating Return Types

After formal argument creation we create also the return types and we attach them to the newly created features.

In the fourth step of the process (example 111) facts for the return types are created. From the signature fact we extract the first type in the list and we create a type mark fact, if needed.

7.3.5 Type Exheritance

The feature exheritance rules call a set of other rules related to type exheritance. These rules are not grouped in a special CT, but they are very complex so they have to be presented separately. Type exheritance shows how to select the representing type from a list of correspondent types from the exherited classes in several contexts. The simplest case of type exheritance occurs when all correspondent types have the same identifiers, meaning that they denote the same type. The representing type is the unique type identifier. The types we consider are all the types from the Eiffel type system and they are: class types referring class declarations and having no actual arguments, separate and expanded type referring class types with the same restriction, like current types, bit types referring constants. Class types referring generic classes or having different but equivalent actual generics, like types having anchors, bit types referring constant features must be treated separately. For types having actual generics and different identifiers, the type exheritance process is recursive.

In example 112 we start from a type identifier list, we sort it in order to eliminate duplicates and finally it is expected to obtain a list having a single representing type identifier. For **like current** types, which depend very much on the class context they are used in, the type information is stored in a special structure having two elements *(TypeId, ClassDeclId)*. As we already know that they are **like current** types coming from subclasses, we are interested only in the first identifier *TypeId*, which has the same value for all **like current** types in the factbase. If none of the two rules matches the conditions, then the next rules for representing type computation will be used.

Exheriting Class Types Having Actual Arguments

In this subsection we analyze class type exheritance when types have no equal identifiers. This means that they might have actual arguments or that they are represented by formal generics. Here we deal only with class types having actual arguments. The candidate type selection is made

Example 112 Exheriting Types Having The Same Identifier

```
selectRepresentantTypeId(TypeIdList,RepresentantTypeId):-
 sort(TypeIdList,[RepresentantTypeId]),
 !.
selectRepresentantTypeId(TypeIdList,RepresentantTypeId):-
 maplist(getLikeCurrentType,TypeIdList,LikeCurrentTypeIdList),
 sort(LikeCurrentTypeIdList,[RepresentantTypeId]),
 !.
getLikeCurrentType((TypeId,_),TypeId).
```

Example 113 Exheriting Class Types Having Actual Arguments (1)

```
selectRepresentantTypeId(TypeIdList,RepresentantTypeId):-
 maplist(extractClassDeclFromClassType,TypeIdList,ClassDeclIdList),
 sort(ClassDeclIdList,[_]),
 maplist(extractClassTypeFromType,TypeIdList,ClassTypeIdList),
 selectRepresentantTypeId(ClassTypeIdList,RepresentantTypeId),
 !.
extractClassTypeFromType(TypeId,ClassTypeId):-
 exists(type(TypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)).
```

in two steps in order to have a better reuse of the rules: first the *classType* facts referred by the *type* facts are extracted (example 113) and then the representing type is computed (example 114).

In example 113 we show how the *classType* facts are extracted from the *type* facts and how the type selection rule is called. The only constraint invoked by this rule is that the base class of all types should be identical. The generic class which is instantiated is known in Eiffel terminology as base class. This operation is performed by the sorting rule which eliminates the duplicates. Afterwards, only one element should remain in the list.

In example 114 the representing type is computed in the following way:

- actual arguments are extracted, they represent the types which instantiate the generic class;
- the actual arguments are reorganized in lists of types based on position criteria (the very same principle was used for feature exheritance);
- for those lists the representing type is computed recursively;
- the resulted type is assembled to represent the final result.

The *selectRepresentantTypeId* rule does all the steps iterated: it uses *extractActualGeneric-TypeIdListFromClassType* in order to extract the type list representing the actual arguments, creates the correspondence list by calling the *createTypeIdListCorrespondence* rule, recursively computes the representing types for the actuals by calling the *selectRepresentantTypeId* rule, and finally creates a class type with the resulted actuals. We mention also that the actual types may be any kind of Eiffel types and we apply the same rules for them as we apply in the general process of type exheritance.

Exheriting Expanded Types In Eiffel variables of expanded types represent objects and not references to those objects. When such a type appears in the exherited features signatures then all the correspondent ones must be the same expanded type in order to be exherited.

In example 115 all types from the *ExpandedTypeIdList* parameter are checked to be expanded types by the rule *isTypeReferringExpandedType*. In the metamodel we designed an expanded type

Example 114 Exheriting Class Types Having Actual Arguments (2)

```
selectRepresentantTypeId(ClassTypeIdList,RepresentantTypeId):-
 member(ClassTypeId,ClassTypeIdList),
 exists(classType(ClassTypeId,ClassDeclId)),
 exists(classDecl(ClassDeclId,_,_,_)),
 !,
 maplist(extractActualGenericTypeIdListFromClassType,ClassTypeIdList,
  ActualGenericsTypeIdLists),
 createTypeIdListCorrespondence(ActualGenericsTypeIdLists,
  CorrespondentActualGenericsTypeIdLists),
 maplist(selectRepresentantTypeId,CorrespondentActualGenericsTypeIdLists,
  ActualGenericsRepresentantTypeIdList),
 createClassTypeWithActualGenericTypes(ClassDeclId,
  ActualGenericsRepresentantTypeIdList,RepresentantTypeId),
 !.
extractActualGenericTypeIdListFromClassType(ClassTypeId,TypeIdList):-
 exists(classType(ClassTypeId,ClassDeclId)),
 exists(classDecl(ClassDeclId,_,_,FormalGenericIdList)),
 length(FormalGenericIdList,Length),
 Length=\=0,
 extractActualGenericTypeIdListFromClassType0(ClassTypeId,ClassDeclId,
  FormalGenericIdList,TypeIdList),
 !.
extractActualGenericTypeIdListFromClassType(_,[]).
extractActualGenericTypeIdListFromClassType0(_,_,[],[]).
extractActualGenericTypeIdListFromClassType0(ClassTypeId,ClassDeclId,
 [FormalGenericId|FormalGenericIdList],
 [TypeId|TypeIdList]):-
 exists(formalGeneric(FormalGenericId,ClassDeclId,_)),
 exists(actualGenericType(_,ClassTypeId,FormalGenericId,TypeId)),
 extractActualGenericTypeIdListFromClassType0(ClassTypeId,ClassDeclId,
  FormalGenericIdList,TypeIdList).
```

Example 115 Exheriting Expanded Types

```
selectRepresentantTypeId(ExpandedTypeIdList,RepresentantTypeId):-
 maplist(isTypeReferringExpandedType,ExpandedTypeIdList,ResultList),
 sort(ResultList,['yes']),
 maplist(extractClassTypeFromExpandedType,ExpandedTypeIdList,ClassTypeIdList),
 selectRepresentantTypeId(ClassTypeIdList,TypeId),
 createExpandedTypeFromClassType(TypeId,RepresentantTypeId),
 !.
extractClassTypeFromExpandedType(TypeId,ClassTypeId):-
 exists(type(TypeId,ExpandedTypeId)),
 exists(expandedType(ExpandedTypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)),
 !.
isTypeReferringExpandedType(TypeId,'yes'):-
 exists(type(TypeId,ExpandedTypeId)),
 exists(expandedType(ExpandedTypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)),
 !.
isTypeReferringExpandedType(_,'no').
```

as referring to a class type. So, in this rule we detached the class types in a separate list through *extractClassTypeFromExpandedType* and we apply recursively the *selectRepresentantTypeId* rule. The result will be a class type which will create the resulted expanded type by the *createExpandedTypeFromClassType* rule.

Exheriting Separate Types

The **separate** keyword introduces the concept of inter-object concurrency in the Simple Concurrent Object-Oriented Programming (SCOOP) mechanism of Eiffel [FOP04]. A class declared as separate means that the class executes its own thread of control. Separate arguments mean that they are synchronization points among concurrent threads. To exherit a feature having a separate type implies that all corresponding types must be separate and must point to the same class. The resulted type is the same separate type present in the correspondence list.

The implementation provided is very similar to the one used with expanded types. In example 116 all types from the *SeparateTypeIdList* parameter are checked to be separate types by the rule *isTypeReferringSeparateType*. In the metamodel we designed a separate type as refering to a class type. So, in this rule we detached the class types in a separate list through *extractClassTypeFromSeparateType* and we apply recursively the *selectRepresentantTypeId* rule. The result will be a class type which will create the resulted separate type by the *createSeparateTypeFromClassType* rule.

Exheriting Like Types

Like types are special types which refer the type of a feature, formal argument or current class denoted by the **current** keyword. Such a type can be defined recursively involving a series of features or formal arguments. For example the type of a feature f can be like g and the type of g can be like h and so on, until the final feature has a non-anchored type. In order to compute the final type of an anchored type we have to call recursively several rules specialised in computing the final type for each kind of Eiffel type.

In example 117 the representing type is obtained after computing the final type for each component from the type list. If the two type sets, original and computed, are not equal that means that there were some anchored types or instantiated generics for which type inference

Example 116 Exheriting Separate Types

```
selectRepresentantTypeId(SeparateTypeIdList,RepresentantTypeId):-
 maplist(isTypeReferringSeparateType,SeparateTypeIdList,ResultList),
 sort(ResultList,[yes]),
 maplist(extractClassTypeFromSeparateType,SeparateTypeIdList,ClassTypeIdList),
 selectRepresentantTypeId(ClassTypeIdList,TypeId),
 createSeparateTypeFromClassType(TypeId,RepresentantTypeId),
 !.
extractClassTypeFromSeparateType(TypeId,ClassTypeId):-
 exists(type(TypeId,SeparateTypeId)),
 exists(separateType(SeparateTypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)),
 !.
isTypeReferringSeparateType(TypeId,'yes'):-
 exists(type(TypeId,SeparateTypeId)),
 exists(separateType(SeparateTypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)),
 !.
isTypeReferringSeparateType(_,'no').
 extractClassTypeFromSeparateType(TypeId,ClassTypeId):-
 exists(type(TypeId,SeparateTypeId)),
 exists(separateType(SeparateTypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)),
 !.
```

Example 117 Exheriting Like Types (1)

```
selectRepresentingTypeId(TypeIdList,(RepresentingTypeId,ScheduledTypeList)):-
 maplist(computeTypeFinalType,TypeIdList,FinalTypeIdList),
 TypeIdList\==FinalTypeIdList,
 selectRepresentingTypeId(FinalTypeIdList,(RepresentingTypeId,
 ScheduledTypeList)),!.
```

Example 118 Exheriting Like Types (2)

```
computeTypeFinalType(TypeId,TypeId):-
 exists(type(TypeId,ClassTypeId)),
 exists(classType(ClassTypeId,ClassDeclId)),
 exists(classDecl(ClassDeclId,_,_,_)),
 !.
computeTypeFinalType(TypeId,FinalTypeId):-
 exists(type(TypeId,ClassTypeId)),
 exists(classType(ClassTypeId,FormalGenericId)),
 exists(formalGeneric(FormalGenericId,_,_)),
 exists(classDecl(ExheritedClassId,_,_,FormalGenericIdList)),
 memberchk(FormalGenericId,FormalGenericIdList),
 exists(classType(ExheritedClassTypeId,ExheritedClassId)),
 exists(exheritance(_,_,ExheritedClassTypeId)),
 exists(actualGenericType(_,ExheritedClassTypeId,FormalGenericId,
  FinalTypeId)),
 !.
computeTypeFinalType(TypeId,TypeId):-
 exists(type(TypeId,ExpandedTypeId)),
 exists(expandedType(ExpandedTypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)),
 !.
computeTypeFinalType(TypeId,TypeId):-
 exists(type(TypeId,SeparateTypeId)),
 exists(separateType(SeparateTypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)),
 !.
computeTypeFinalType(TypeId,TypeId):-
 exists(type(TypeId,BitTypeId)),
 exists(bitType(BitTypeId,_)).
```

was executed. Unless this check is performed the representing type computing rule will go on an infinite recursion. So, next the representing type is computed using the rules defined in the previous subsections.

For the non-anchored types of Eiffel we defined the final type to be the type itself (see example 118). The only exception for these rules are the class types refering formal generic instantiated with some concrete types.

In example 119, for like types the final type may be computed immediately being either the type of a feature, formal argument or **current** or recursively if there is a chain of referred entities.

Exheriting Bit Types

Bit types are used in Eiffel to represent integers of different sizes. Bit types may be expressed using a manifest constant or an integer constant feature. No matter how the candidate bit types are expressed the result type must be a bit type of the same size. Even if the constant features referred by the bit types are exheritable, they cannot be redefined in the subclasses again as constant features. The chosen solution is to exherit the bit type using a manifest constant regardless of its origins in the subclasses. Another possible solution which affects more the exherited classes is to move up the constant feature. Such a solution can be triggered by selecting explicitly the implementation exheritance.

In example 120 the type representing selection rule checks all types to have the same size using the *maplist* operator with *getBitTypeValue* rule on the type list. The resulted list is sorted,

Example 119 Exheriting Like Types (3)

```
computeTypeFinalType((TypeId,ClassDeclId),RepresentingTypeId):-
 exists(type(TypeId,LikeTypeId)),
 exists(likeType(LikeTypeId,IdentifierId)),
 exists(identifier(IdentifierId,'current')),
 createClassTypeFromClassDecl(ClassDeclId,RepresentingTypeId),
 !.
computeTypeFinalType(TypeId,TypeId):-
 exists(type(TypeId,LikeTypeId)),
 exists(likeType(LikeTypeId,IdentifierId)),
 exists(identifier(IdentifierId,'current')),
 !.
computeTypeFinalType(TypeId,FinalTypeId):-
 exists(type(TypeId,LikeTypeId)),
 exists(likeType(LikeTypeId,Id)),
 ((exists(featureDecl(Id,_,_)),
   exists(typeMark(Id,TempTypeId)));
   exists(formalArgument(Id,_,_,TempTypeId))),
   exists(type(TempTypeId,_)),
 computeTypeFinalType(TempTypeId,FinalTypeId),!.
```

Example 120 Exheriting Bit Types

```
selectRepresentantTypeId(TypeIdList,RepresentantTypeId):-
 maplist(getBitTypeValue,TypeIdList,ResultSizeList),
 sort(ResultSizeList,[IntegerValue]),
 integer(IntegerValue),
 createBitType(IntegerValue,RepresentantTypeId),
 !.
getBitTypeValue(TypeId,IntegerValue):-
 exists(type(TypeId,BitTypeId)),
 exists(bitType(BitTypeId,XId)),
 ((exists(featureDecl(XId,_,_)),
  exists(featureManifestConstant(XId,ManifestConstantId)),
  exists(manifestConstant(ManifestConstantId,IntegerValue,'integer')));
 (exists(manifestConstant(XId,IntegerValue,'integer')))).
```

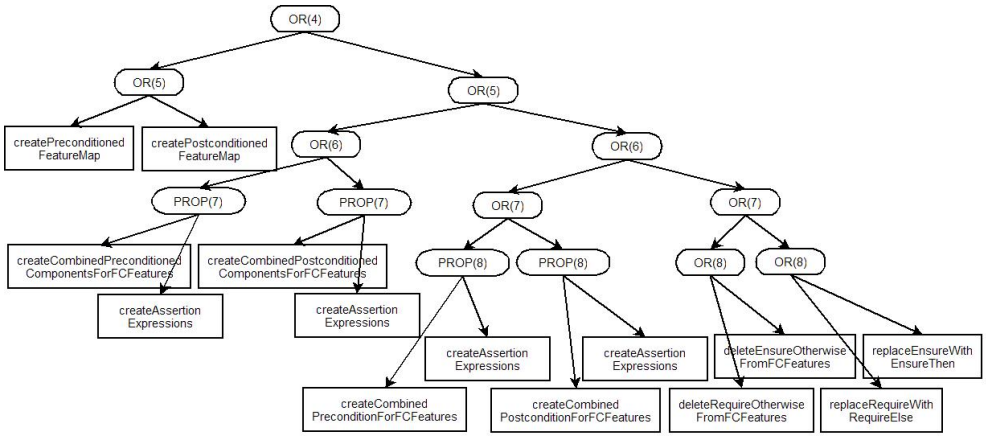


Figure 7.5: Combined Assertion Generation CT Subtree

duplicates are eliminated and a single integer is expected to remain. Afterwards, a bit type is created by the *createBitType* rule if the type does not already exist.

7.3.6 Combined Assertion Generation

In figure 7.5 we present the CT subtree responsible with the creation of combined preconditions and postconditions as they were presented in subsection 5.4.2.

First, all features which need preconditions or postconditions from the foster class are placed in two maps.

Next, the combined preconditions and postconditions components are created by the *createCombinedPrecondtionComponentsForFCFeatures* and *createCombinedPostcondtionComponentsForFCFeatures* CTs. Each of the two CTs are followed by the expression creation CT named *createAssertionExpressions*. Each component has the following form *current.generator.is_equal("EC1") and (x>1 and y>1)*. This precondition component corresponds to the *EC1* exherited class and *(x>1 and y>1)* are the and-ed preconditions for the feature in that class. For each exherited class such a component is created.

Finally, the *createCombinedPreconditionForFCFeatures* and *createCombinedPostconditionForFCFeatures* CTs will assemble the final assertions for the exherited feature. For example such an assertion will have the following form:

(current.generator.is_equal("EC1") and (x>1 and y>1)) or
(current.generator.is_equal("EC2") and (x>2 and y>2)) or
(current.generator.is_equal("EC3") and (x>3 and y>3)) or
(x>0 and y>0).

It can be noticed that the created components are or-ed altogether and the *(x>0 and y>0)* is the assertion expressed by *requireOtherwise* or *ensureOtherwise* from the foster class.

As a last step the *requireOtherwise*, *ensureOtherwise* facts are deleted and the *require* respectively *ensure* facts from the subclasses are replaced with *requireElse* and *ensureThen*.

7.3.7 Implementation Exheritance

In figure 7.6 there are only two CTs which clone in a generic manner the facts from the feature implementation located in the subclass into the foster class and performs the necessary context adjustments.

Cloning Cloning is a generic process driven by the formal metamodel of our implementation, a fragment is presented in section 7.1.3 example 99. We visit the original tree and the new tree

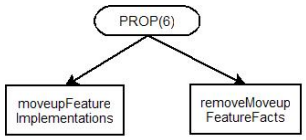


Figure 7.6: Implementation Exheritance CT Subtree

is created during the visiting process. For each source node a new node is created by cloning. The arguments of the source node are analyzed and they may be of several kinds: own identifier, parent identifier, child identifier, children identifier list, relation link. The own identifier is cloned by generating a new identifier for the new tree. The parent identifier is cloned by retrieving the identifier of the parent node in the new tree. That identifier was generated at the previous level as own identifier. The child identifier is cloned recursively because it can be seen as the root of another subtree to be cloned. A similar cloning procedure is applied to children identifier list but with multiple ramifications. There are nodes which refer the parent without the parent knowing it. Also these nodes are cloned recursively refering the new parent. In this stage references are kept as they are in the original tree.

Replacing References In the previous step we succeeded to create a new tree with the same structure but having relation links to the nodes of the original tree and not of the new tree as we partially intend to have. For example, calls to features from other classes will remain unchanged, but calls to features within the same class must be dispatched to features of the foster class. It would be inconsistent at the model level to have a call in the foster class refering a feature in the subclass. Such a call should be redirected to the correspondent new feature in the foster class, thus making the model consistent. For this purpose we will use two maps created in the feature exheritance process presented in subsection 7.3.3. The two maps used are *candidateToNewFeatureMap* and *oldFeatureToRedefinedFeatureMap*. It means that all references to candidate features from exherited classes will be dispatched to the corresponding new features of the foster class and references from old features within the exherited class are dispatched towards redefined features of the foster class.

During cloning and reference replacement new facts are created and some facts are replaced by creation and deletion. So the two CTs in figure 7.6 are responsible with fact manipulations. The former performs the cloning and reference replacement by adding facts, while the latter deletes the replaced facts.

7.3.8 Adapt Transformation

In this section we discuss issues related to exheriting features having same semantics but different, non-covariant signatures, which can be adapted to a common one by simple method or operator calls. The considered exheritance case refers to the "light" feature adaptation facility offered by reverse inheritance. From another point of view the adaptation mechanism can be considered a stronger redefinition mechanism, which does not cover renaming. The adaptation mechanism will handle only the cases which cannot be handled by redefinition. We have designed several types of adaptations: i) scale adaptations, which may imply or not the change of the signature; ii) parameter order change, iii) parameter number change, iv) return type change.

The implementation of such an adaptive feature behavior is difficult because of the Eiffel rule which states that the final feature name is unique at the level of a class. In other words, we cannot have two features with the same name and different signatures in the same class, as we would maybe like to have: one with the new desired homogeneous signature and the already existing non-homogeneous ones. This may cause problems in the implementation of adapted features.

In order to simplify the reverse inheritance semantics, we will not allow moving up an adapted feature implementation from an exherited class into the foster class. Instead, the body of an

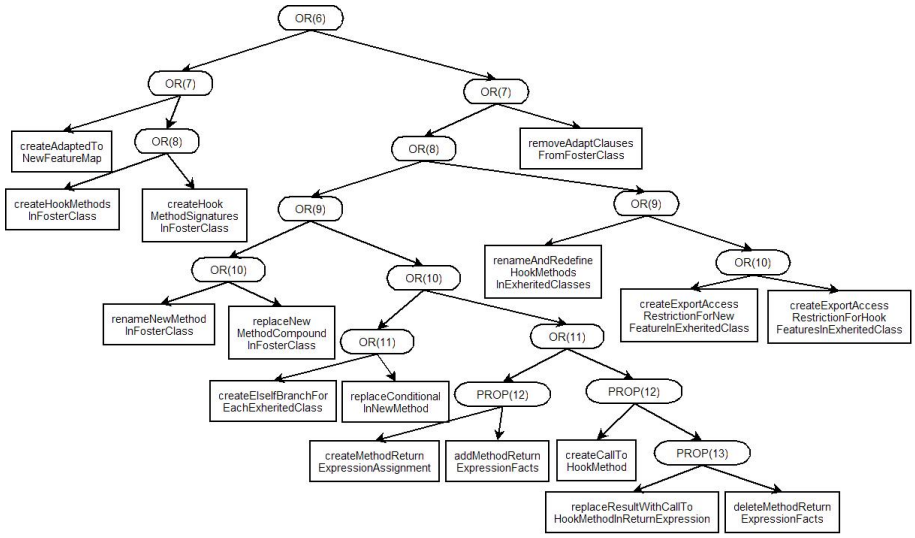


Figure 7.7: Adapt Transformation CT Subtree

adapted feature could be explicitly rewritten by the programmer at the foster class level. From the implementation point of view there is no problem to combine the transformation effects of the two concepts. The only factor which may affect the generated class hierarchy is the order of transformations: first the transformations for move up must be performed and only afterwards the transformations for the adaptation implementation. Also, we have to consider that an adapted feature at the foster class level can be deferred as well.

The adaptation expressions can be expressed in the foster class body, in part for each adapted class. For that matter, the syntax we designed allows to see the adapted candidate feature signature and the adaptation expression for it. Attribute and method adaptations are treated differently. For attributes there must be provided two adaptation expressions for each adapted subclass: one for getting and one for setting the attribute value. For methods there is one expression which adapts the return value of the adapted method from the subclass and also several expressions for adapting the arguments in calling the candidate feature from that subclass 5.2.

Method Adaptation

Expressing Method Adaptation In this subsection we will present the implementation of adapted methods. In example 121 we show a typical situation in which the programmer wants to use reverse inheritance in order to exherit features which have the same semantics but different signatures. Formal arguments and return types are optional in the structure of a method but in the context of an adaptation they cannot miss both, otherwise there is nothing to adapt.

In figure 121 we present a basic example of one foster class named *SHAPE* and two exherited classes named *RECTANGLE* and *TRIANGLE*, all having a *computation* feature. This feature has the same semantics but different signatures and scale representations in the *RECTANGLE* and *TRIANGLE* subclasses. We want to exherit it under a common signature in the foster class *SHAPE*. It can be noticed that the features from class *RECTANGLE* are using type *INTEGER* and are expressed in decimeters (dm), while the features in class *TRIANGLE* are using type *NATURAL* and are expressed in centimeters (cm).

In example 122 we present several possible use cases of the classes instances:

- a call on an instance of *SHAPE* through a reference of type *SHAPE* will execute the new code added in the foster class, so the return value will be zero;

Example 121 Method Adaptation Example

```
foster class SHAPE
exherit
 RECTANGLE
  adapt computation
 end
 TRIANGLE
  adapt computation
  end
 all
feature
 computation(x:REAL):REAL is --cm
 adapted
  --cm -> dm ; dm -> cm
  {RECTANGLE}(y:INTEGER):INTEGER is (x.rounded):result*10
  --cm -> m ; m -> cm
  {TRIANGLE}(z:NATURAL):NATURAL is (x.rounded.as_natural_16):result*100
 do
  result:=0
 end
end
class RECTANGLE
create make
feature
 w,h:INTEGER --dm
 make(pw,ph:INTEGER) is
 do w:=pw; h:=ph end
 computation(y:INTEGER):INTEGER is do result:=(w+h)*2*y end --dm
end
class TRIANGLE
create make
feature
 a,b,c:NATURAL --m
 make(pa,pb,pc:NATURAL) is
 do a:=pa; b:=pb; c:=pc end
 computation(z:NATURAL):NATURAL is do result:=(a+b+c)*z end --m
end
```

Example 122 Method Adaptation Use Example

```
class ROOT_CLASS
create make
feature
 s:SHAPE
 r:RECTANGLE
 t:TRIANGLE
 make is
 do
  create s
  create r.make(10,20) --dm
  create t.make(2,2,2) --m
  io.put_string("Using SHAPE object and SHAPE reference ")
  --io.put_real(s.computation(1)) is replaced by
  io.put_real(s.adapted_computation(1)) --1 EURO / cm
  io.new_line
  io.put_string("Using RECTANGLE object and SHAPE reference ")
  s:=r
  --io.put_real(s.computation(1)) --1 EURO / cm
  io.put_real(s.adapted_computation(1)) --1 EURO / cm
  io.new_line
  io.put_string("Using TRIANGLE object and SHAPE reference ")
  s:=t
  --io.put_real(s.computation(1)) --1 EURO / cm
  io.put_real(s.adapted_computation(1)) --1 EURO / cm
  io.new_line
  io.put_string("Using RECTANGLE object and RECTANGLE reference ")
  io.put_integer(r.computation(10)) --10 EURO / dm
  io.new_line
  io.put_string("Using TRIANGLE object and TRIANGLE reference ")
  io.put_natural(t.computation(100)) --100 EURO / m
  io.new_line
 end
end
```

- a call on an instance of *RECTANGLE* or *TRIANGLE* through a reference of type *SHAPE* will provide a uniform result computed using centimeter representation of data;
- a call on an instance of *RECTANGLE* through a reference of type *RECTANGLE* will provide the native implementation computed using decimeters;
- a call on an instance of *TRIANGLE* through a reference of type *TRIANGLE* will provide the native implementation computed using meters.

Because of the unique final feature name rule of Eiffel, at foster class level we cannot use the same name for the adapted feature, so we called it *adapted_computation*. Next, we present in detail one implementation solution.

Method Adaptation Implementation Overview The proposed implementation solution from figure 123 is based on the generation of a **combined mediator method** at the level of the foster class. The **mediator method** translates a call to the signature of the new feature (from the foster class) towards each original local implementation (from the exherited class). So, the exherited feature final name is changed by adding the "adapted" suffix. We must specify that by **changing the name** we mean textually replacing the feature name occurrences with the augmented one, while by **renaming** we refer to the ordinary renaming mechanism of Eiffel. We remind also that Eiffel programming language does not allow overloading for methods, so multiple features with the same name may not coexist in the same class. The motivation behind changing the name of the exherited feature in the foster class is to be able to add the redefined mediator feature in the adapted exherited class(es) by inheritance. The non-adapted exherited classes features will not be affected.

Generating the Combined Mediator Method in the Superclass The first step involved by such a transformation is to generate the mediator method in the superclass. The mediator method will have the signature of the desired new feature and also the new augmented name. The body of this method will guard all the adaptation expressions for each exherited class. In order to select the different adaptation code executions the type of the instance is tested statically. There should be a branch expressing the adaptation for each subclass and one for the foster class. In our example:

- an instance of type *RECTANGLE* will execute the *rectangle_computation* method;
- an instance of type *TRIANGLE* will execute the *triangle_computation* method;
- an instance of type *SHAPE* will execute the redefined code.

The mediator method purpose is to be inherited in the subclasses and to use the implementation of the hook methods found at that level. This approach concentrates all adaptations at the level of the foster class while other possible implementation is to distribute each adaptations in its corresponding subclass.

Adaptation Calls The adaptation calls involve two filters: the adaptation of input arguments used for the call and the adaptation of the returned or output value of the call. In example 123 in the context of class *RECTANGLE*, argument x of type *REAL* is rounded to an *INTEGER* value by calling the *rounded* method and passed to the original *rectangle_computation* method. In the context of class *TRIANGLE*, the very same argument x of type *REAL* is converted to a *NATURAL* value by calling *rounded* and *as_natural_16* methods and passed to the *triangle_computation* method.

Example 123 Method Adaptation Implementation Solution

```
class SHAPE
feature
 adapted_computation(x:REAL):REAL is --cm
 do
  if current.generator.is_equal ("RECTANGLE") then
   result:=rectangle_computation(x.rounded)*10 --cm
  elseif current.generator.is_equal("TRIANGLE") then
   result:=triangle_computation(x.rounded.as_natural_16)*100 --cm
  elseif current.generator.is_equal("SHAPE") then
   -- put the implementation if the feature is effective
   result:=0
  else
   -- do nothing
  end
 end
feature {NONE} --hook methods
 rectangle_computation(x:INTEGER):INTEGER is do end --dm
 triangle_computation(x:NATURAL):NATURAL is do end --m
end
class RECTANGLE
inherit SHAPE
 rename rectangle_computation as computation
 export {NONE} adapted_computation, triangle_computation
 redefine computation
end
create make
feature
 w,h:INTEGER
 make(pw,ph:INTEGER) is do w:=pw; h:=ph end
 computation(y:INTEGER):INTEGER is do result:=(w+h)*2*y end --dm
end
class TRIANGLE
inherit SHAPE
 rename triangle_computation as computation
 export {NONE} adapted_computation, rectangle_computation
 redefine computation
end
create make
feature
 a,b,c:NATURAL
 make(pa,pb,pc:NATURAL) is do a:=pa; b:=pb; c:=pc end
 computation(z:NATURAL):NATURAL is do result:=(a+b+c)*z end --m
end
```

Generating Empty Hook Methods in the Superclass The methods called from the mediator method will be called **hook methods** since they represent the original methods from the subclasses for the adapted feature. The implementation is using the idea from the **Template Method** design pattern [GHJV97]. The hook methods are empty body methods generated in the superclass and having the same signature as the ones from the subclasses. The name of such a method is composed out of the name of the subclass concatenated with the original name of the subclass method. Such methods may be deferred if the foster class is deferred, but they cannot be deferred by default because it will force this restriction on the foster class, which might not be always desired.

New Method Access in the Superclass If an adapted method call is used in the superclass that call should be redirected towards the adapted method. For instance in class *SHAPE* any call to *computation* method will be routed to *adapted_computation* method. Such a code may exist in a redefined exherited feature. Writing a call of a method which does not exist explicitly in the foster class may affect the readability of the class hierarchy.

Renaming and Redefining the Hook Methods in Subclasses In the subclasses all hook methods are renamed towards the original features. They must be redefined also because their implementation changes. In particular, when the foster class is deferred they might not be redefined since in the superclass they are deferred. With the help of renaming the hook methods from the superclass are linked with the original methods from the subclasses.

Access Restriction in the Subclasses All the hook methods from the superclass are generated and inherited in all the subclasses and in this sense we can limit their use by using the export clause in order to make them secret, except the redefined ones. For example, in class *RECTANGLE* feature *triangle_computation* is made secret since it does not make sense to be used by any client of this class. Also, the adapted feature from the superclass is not allowed to be available to any clients of the descendant classes. For example *adapted_computation* cannot be used by any clients of *RECTANGLE* or *TRIANGLE* classes.

Client Access Regarding the client access to the newly created class hierarchy there are some modifications to be made. All calls to the adapted features made through superclass type references must be replaced by the new augmented name of the feature. For example the calls to *computation* feature in class *ROOT_CLASS* are replaced by calls to *adapted_computation* when reference of type *SHAPE* was used. All the other calls made through references of subclasses types are kept as such.

Attribute Adaptation

Expressing Attribute Adaptation In this subsection we present a typical situation of a common attribute present in all exherited classes and having different scale representations which is desired to be factored. The intention is to reuse the attribute by exheritance and to get a uniform representation at the foster class level.

In example 124 we start from two subclasses *RECTANGLE* and *TRIANGLE* which use two representations: decimeters and meters to store their perimeter feature. A new *SHAPE* class is constructed on the top of the two classes using reverse inheritance and a new *perimeter* attribute is desired in this class to be expressed in centimeters and still to correspond to the two perimeter features from the subclasses expressed in decimeters and meters. As a secondary goal the creation method is also exherited.

In example 125 we present how such an adapted class hierarchy is intended to be used. When using the *SHAPE* references on any instance the *perimeter* attribute will return a value expressed in centimeters, while using *RECTANGLE* or *TRIANGLE* type references the perimeter will be provided in decimeters and meters.

Example 124 Attribute Adaptation Example

```
foster class SHAPE
exherit
 RECTANGLE
  adapt make, perimeter
 end
 TRIANGLE
  adapt make, perimeter
 end
 all
create make
feature --adapted
 make(p:REAL) is --cm
 adapted
  {RECTANGLE} (i:INTEGER) is ((p/10).rounded)
  {TRIANGLE} (n:NATURAL) is ((p/100).rounded_as_natural_16)
 do
  perimeter:=p
 end
perimeter:REAL is --cm
 adapted
  {RECTANGLE} INTEGER is ((precursor/10).rounded) : result*10
  {TRIANGLE} NATURAL is ((precursor/100).rounded_as_natural_16) : result*100
 end
end
class RECTANGLE
create make
feature --creation
 make(i:INTEGER) is do perimeter:=i end
feature --original attribute declaration
 perimeter:INTEGER --dm
end
class TRIANGLE
create make
feature --creation
 make(n:NATURAL) is do perimeter:=n end
feature --original attribute declaration
 perimeter:NATURAL --m
end
```

Example 125 Attribute Adaptation Use Example

```
class ROOT_CLASS
create make
feature
 s:SHAPE
 r:RECTANGLE
 t:TRIANGLE
 make is
 do
  create s.adapted_make(100) --cm
  create r.make(10) --dm
  create t.make(1) --m
  io.put_string("Using SHAPE object and SHAPE reference ")
  --io.put_real(s.perimeter) will be replaced by
  io.put_real(s.get_perimeter) --cm
  io.new_line
  io.put_string("Using RECTANGLE object and SHAPE reference ")
  s:=r
  --io.put_real(s.perimeter) will be replaced by
  io.put_real(s.get_perimeter) --cm
  io.new_line
  io.put_string("Using TRIANGLE object and SHAPE reference ")
  s:=t
  --io.put_real(s.perimeter) will be replaced by
  io.put_real(s.get_perimeter) --cm
  io.new_line
  io.put_string("Using RECTANGLE object and RECTANGLE reference ")
  io.put_integer(r.perimeter) --dm
  io.new_line
  io.put_string("Using TRIANGLE object and TRIANGLE reference ")
  io.put_natural(t.perimeter) --m
  io.new_line
 end
end
```

Attribute Adaptation Implementation Overview The implementation solution for adapting attributes is based on the method adaptation technique used earlier combined with a common refactoring operation of replacing direct attribute accesses with setter and getter method calls. In a good object-oriented design the object state represented by attributes should be accessible only through accessor methods and not directly. In Eiffel attributes can be accessed for writing only inside the class they are defined in, while they are accessible for reading to all clients of its hosting class.

In the implementation depicted in examples 126 and 127 the effect of attribute perimeter exheritance from the subclasses is simulated. We will focus our attention on the attribute exheritance and not on the creator method adaptation. The order in which the code modifications are presented is a valid order to execute them also.

Generating a New Attribute in the Superclass The first step of the transformation chain is to create a new attribute with an augmented name in the superclass. In our example we created *adapted_perimeter* attribute of type REAL to store values for any *SHAPE* instances that might be created if the class is not deferred.

Generating the Combined Mediator Accessor Methods A second step of the code generation is the creation of mediator accessor methods, getter and setter for the adapted attribute. These methods are capable of handling all the adaptations expressions for the subclass adapted attributes. For each adapted subclass attribute there will be a decisional branch and within them the set and get methods are expressing the desired scale transformation. In each branch one hook method from the corresponding subclass will be used. For example, in the context of class *RECTANGLE* the getter will call the *rectangle_get_perimeter hook* method multiplied by 10 in order to transform the value from decimeters into centimeters. The hook method will be redefined in the *RECTANGLE* subclass to get the value of the original *perimeter* attribute.

Adaptation Calls Since we are adapting attributes, for each attribute we have to provide two expressions: one for putting a value into the original attribute and the other to get the value from it. Thus, the first adaptation expression containing the **precursor** keyword will be used in the setter method, while the other one containing the **result** keyword will be used in the getter method.

Generating Empty Hook Methods in the Superclass In order to access the attributes from the subclasses we generate empty body hook setter and getter methods for the attributes of each subclass. These methods will be redefined in the subclasses and they will access the adapted original attributes. We need the hook methods because adapted attributes may have different signatures in the subclasses and this pair of redefined methods will assure the access to the subclass attributes from the superclass level.

New Attribute Access in the Superclass Since our technique emulates the homogeneous scale of an attribute, in the foster class any direct access to the attribute must be replaced with an access to the augmented attribute. For instance in method *make* of example 124, the *perimeter* attribute from the assignment must be replaced with the *adapted_perimeter*. Another option would be to use calls to the generated accessor methods.

Access Restrictions in Subclasses Following the same idea used for methods we decided to restrict the access to the generated pairs of accessors for the subclasses clients. The same principle applies also to the generated attribute. In our example features like *adapted_perimeter*, *get_perimeter*, *set_perimeter*, *rectangle_get_perimeter*, *rectangle_set_perimeter* are not exported to subclass clients.

Example 126 Attribute Adaptation Solution (1)

```
class SHAPE
create adapted_make
feature --generated mediator creator method
 adapted_make(p:REAL) is
 do
  if current.generator.is_equal("RECTANGLE") then
   rectangle_make(p.rounded)
  elseif current.generator.is_equal("TRIANGLE") then
   triangle_make(p.rounded.as_natural_16)
  elseif current.generator.is_equal("SHAPE") then
   --perimeter:=p is replaced by
   adapted_perimeter:=p
  else
   --perimeter:=p is replaced by
   adapted_perimeter:=p
 end
end
feature --generated hook methods for the creator method
 rectangle_make(i:INTEGER) is do end
 triangle_make(n:NATURAL) is do end
feature --generated attribute for storing data for SHAPE instances
 adapted_perimeter:REAL --cm
feature --generated accessors methods containing all adaptations
 get_perimeter:REAL is do --cm
  if current.generator.is_equal ("RECTANGLE") then
   result:=rectangle_get_perimeter*10 --cm
  elseif current.generator.is_equal("TRIANGLE") then
   result:=triangle_get_perimeter*100 --cm
  elseif current.generator.is_equal("SHAPE") then
   result:=adapted_perimeter
  else
   --code for direct subclasses of the foster class
   result:=adapted_perimeter
  end
 end
 set_perimeter(x:REAL) is do --cm
  if current.generator.is_equal ("RECTANGLE") then
   rectangle_set_perimeter((x/10).rounded) --dm
  elseif current.generator.is_equal("TRIANGLE") then
   triangle_set_perimeter((x/100).rounded.as_natural_16) --m
  elseif current.generator.is_equal("SHAPE") then
   adapted_perimeter:=x
  else
   adapted_perimeter:=x
  end
 end
feature --generated hook methods for the adapted attribute
 rectangle_get_perimeter:INTEGER is do end --dm
 rectangle_set_perimeter(x:INTEGER) is do end --dm
 triangle_get_perimeter:NATURAL is do end --m
 triangle_set_perimeter(x:NATURAL) is do end --m
end
```

Example 127 Attribute Adaptation Solution (2)

```
class RECTANGLE
inherit
 SHAPE
  rename rectangle_make as make
  export
  {NONE} adapted_make, triangle_make,
   adapted_perimeter,
   get_perimeter,set_perimeter,
   rectangle_get_perimeter,rectangle_set_perimeter,
   triangle_get_perimeter,triangle_set_perimeter
   redefine make,rectangle_get_perimeter,rectangle_set_perimeter
 end
create make
feature --creation
 make(i:INTEGER) is do perimeter:=i end
feature --original attribute declaration
 perimeter:INTEGER --dm
feature {NONE} --generated redefined hook accessor methods
 rectangle_get_perimeter:INTEGER is do result:=perimeter end --dm
 rectangle_set_perimeter(x:INTEGER) is do perimeter:=x end --dm
end
class TRIANGLE
inherit
 SHAPE
  rename triangle_make as make
  export
   {NONE} adapted_make, rectangle_make,
   adapted_perimeter,
   get_perimeter,set_perimeter,
   rectangle_get_perimeter,rectangle_set_perimeter,
   triangle_get_perimeter,triangle_set_perimeter
  redefine make,triangle_get_perimeter,triangle_set_perimeter
end
create make
feature --creation
 make(n:NATURAL) is
 do perimeter:=n end
feature --original attribute declaration
 perimeter:NATURAL --m
feature {NONE} --generated redefined hook accessor methods
 triangle_get_perimeter:NATURAL is do result:=perimeter end --m
 triangle_set_perimeter(x:NATURAL) is do perimeter:=x end --m
end
```

Client Access Most of the code changes implied by this implementation affect the new clients of the class hierarchy. As Eiffel does not allow value assignment for attributes, only the attribute queries are affected. So, each attribute query attached to an instance referred by the foster class will be replaced with the corresponding accessor method. For instance in our example all three *perimeter* accesses to objects referred by *SHAPE* reference will be replaced by *get_perimeter* method call. The *TRIANGLE* and *RECTANGLE* reference calls are not affected by any change, meaning that they will access the original *perimeter* feature.

Conclusions

The presented implementation keeps untouched the code of the original features from the subclasses and all their clients. However, hook methods are inherited, redefined or made private in the exherited classes. This will slightly modify the text of the subclasses in the inheritance section. Since the implementation implies **changing** the name of the new feature with an augmented one, it may affect the readability of the generated code. If the programmer deliberately wants to **rename** the new feature from the exherited classes, then a new name is provided for the new feature.

The clients of the foster class which may call the new feature are the newly built classes of the current redesign process because only these classes "know" about the foster class. Those classes must be regenerated in order to make those calls pointing to the augmented feature. Even the foster class may have local calls to the adapted features which must be changed. In practice, the new augmented name must be concatenated with a unique numerical key in order to assure that there will be no accidental name clashes.

The mediator method format may differ depending whether the adapted features have or not formal arguments and return type. In case there are no formal arguments in the original feature from exherited class then any mediator formals will be ignored. In case the original feature has no return type then the assignment of the call to the **result** variable will be replaced with a simple call of the original exherited feature.

Another conclusion can be drawn related to the interaction between method and attribute adaptations, explicitly to the order of their execution. First, method adaptations should be performed the because they may use some adapted attributes, which may have to be replaced later, and only afterwards should be performed the attribute adaptation.

7.3.9 Feature Clauses Generation

In figure 7.8 we describe how clauses are set for the new features. The *addRedefineClausesForNewFeaturesInTheExheritedClasses* CT creates redefine clauses in the exherited classes for the newly migrated features in the foster class. Similarly, the *addRedefineClausesForRedefinedFeaturesInTheExheritedClasses* CT creates redefine clauses on the inheritance branches of the exherited classes for the redefined features of the foster class. The *addUndefinedClausesInTheExheritedClasses* CT handles the case when a new feature in the foster class is effective (redefined or moved up) and one or more of the candidates are deferred. In such a case on the corresponding inheritance branch the undefinition keyword must be set. The *moveRenameFromFosterClassToExheritedClass* CT handles the case of renamed features in the foster class. In reverse inheritance the source features from the exherited classes are renamed in the foster class to a common name, but in the equivalent ordinary inheritance the feature from the foster class will be renamed back to its original name of the feature in the exherited class.

The CT named *createNewFeatureBlocksForExportedFeatures* is responsible with the creation of feature blocks for the features which are exported to different client classes. In the *createNewFeaturesClientClasses* CT we compute the set of common clients for each exherited feature relying on the candidate features clients. When no common clients can be computed and no export clauses are present for a set of candidate features then the feature in the foster class is exported to class *NONE*, meaning that the feature is private, usable only within the class. The global *exportExheritFeature* clauses from the foster class are translated into *featureClientClass* clauses by

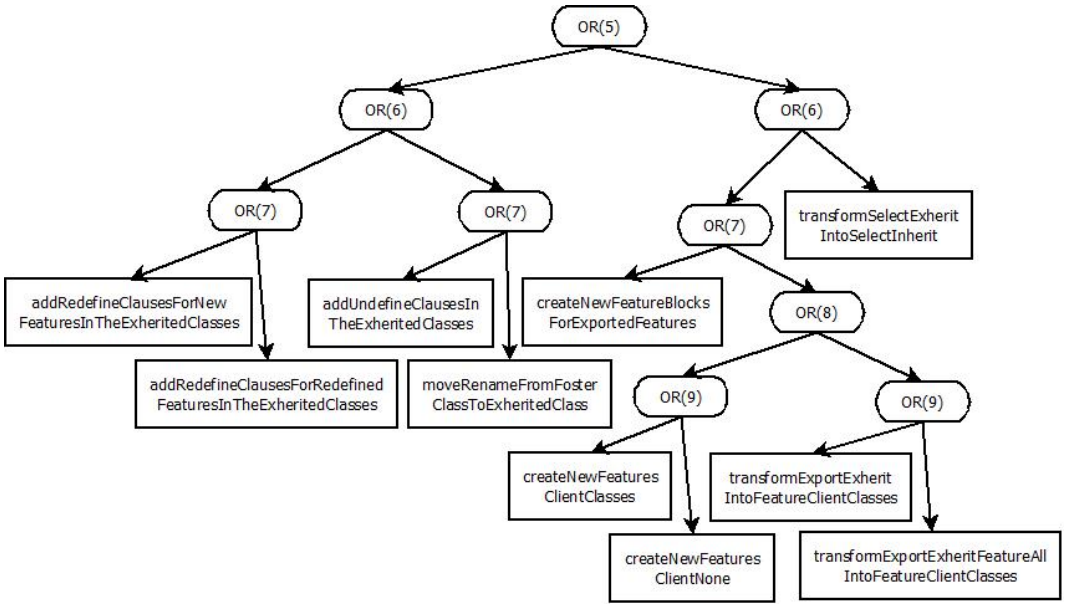


Figure 7.8: Feature Clauses Generation CT Subtree

the *transformExportExheritIntoFeatureClientClasses*. Finally, the *exportExheritFeatureAll* clauses are translated into *featureClientClass* clauses. Thus each client class of the clause is added to the non-empty client list of features.

7.3.10 Hierarchy Transformations

Class hierarchy transformations are presented in the CT subtree of figure 7.9. The first CT named *addCallReceiverPrecursorTypeAttribute* deals with issues discussed in subsection 6.3.2. Ambiguous class configurations are detected and the precursor attribute is added. The following CT, named *createFosterClassType* creates the class type corresponding to the inheritance branch in the case of non-generic foster and exherited classes. Next, the exheritance branch is transformed in inheritance branch by the *transformExheritIntoInheritSingle* CT.

In the context of generic foster classes and generic exherited classes we create several foster class types by the *transformExheritIntoInheritMultiple* CT. The types will be updated separately with actual generics. The first update CT named *updateActualGenericsForTheFosterClass1* deals with the case of exherited classes instantiated with formal generics from the foster class (see subsection 5.3.1). The second update CT named *updateActualGenericsForTheFosterClass2* handles the case of exherited classes instantiated with types equal to the constraint types of both exherited and foster classes (see subsection 5.3.2).

7.3.11 Reverse Inheritance Elements Removal

In figure 7.10 we present the CT subtree dealing with the elimination of reverse inheritance semantical elements from the model. Thus, the feature selection clauses are deleted from the factbase: *allFeature*, *nothingFeature*, *onlyFeature*, *exceptFeature* using CTs like: *removeAllFeatureSelection*, *removeNothingFeatureSelection*, *removeAllFeatureSelection*, *removeExceptFeatureSelection*. Next, if the foster class has deferred features then the foster class itself is set as deferred by the *addDeferredToClass* CT. Finally, the *foster* facts are eliminated using the *removeFoster* CT.

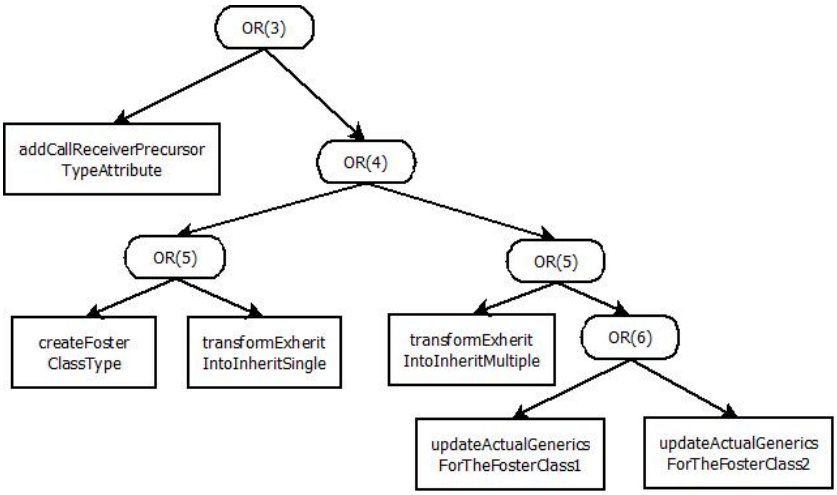


Figure 7.9: Hierarchy Transformations CT Subtree

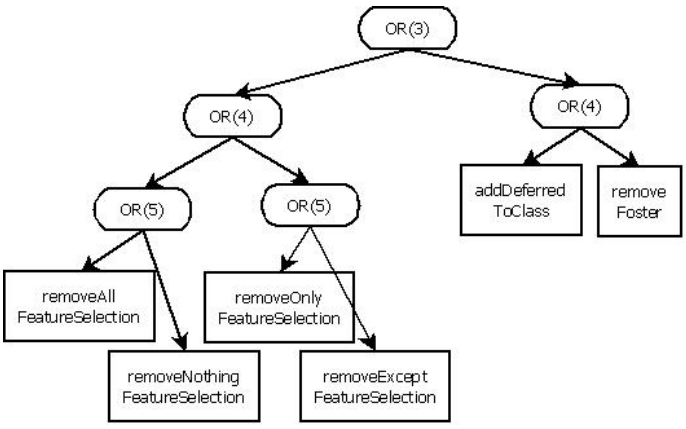


Figure 7.10: Reverse Inheritance Elements Removal CT Subtree

7.4 Summary

At the beginning of this chapter we presented several implementation possibilities for reverse inheritance. The adopted implementation solution does not invent a new programming language, but extends an existing one, so it resulted RIEiffel. The prototype is based on the GOBO Eiffel free library, thus it can be tested and used in industrial projects. As a consequence the grammar of GOBO was augmented with the reverse inheritance rules and was listed in appendix A. The grammar is designed incrementally, so any pure Eiffel source file will conform to the extended grammar.

In the implementation we took the decision to express the semantics of reverse inheritance through Prolog fact transformations. In section 7.1 we presented the metamodel for the logic representation of Eiffel programs. First, the pure Eiffel entities and afterwards the reverse inheritance extension elements are modeled. The factbase metamodel respects the 3NF form for relational databases. For each significant grammar rule a Prolog fact is designed to represent a certain language entity. We have to admit that the metamodel design is not homogeneous because some facts have relations with their parents in both ways and other facts only in one direction. This decision was taken in order to optimize the model, to minimize the number of facts and to capture some semantical aspects. The link between *formalArguments* and *formalArgument* is bidirectional and also the children are ordered in the parent, since argument order may not be ignored. The language entities were modeled as flexible facts linked together by identifiers. Adding and removing language entities are expressed at model level by adding or removing facts. The factbase model is structured in two main parts: Eiffel language related facts and RIEiffel specific facts. The Eiffel facts are structured as follows: class header, formal generics, inheritance clauses, creators, features, instructions, expressions. The RIEiffel model contains the exheritance clauses facts and the adaptation facts. Still some details like spaces and comments are not modelled in our design since they have no semantical value. As a consequence the reverse engineered sources may not have the same textual organization.

A Prolog metamodel which consists in describing each rule arguments, allows checking automatically the type consistency of the factbase. In the metamodel are also included navigation information for the nodes to be displayed in a graphic user interface. Validity rules are stated around the reverse inheritance concepts like: feature factorization, redefinition, adaptation, selection, export and around some complex contexts of reverse and ordinary inheritance. These rules substitute the semantical analysis which should be performed by a compiler against the reverse inheritance extension of a class. For example the exherited feature selection clauses must select the features in an unique manner.

The semantics of reverse inheritance is implemented by three modules of a prototype. The first one is a RIEiffel parser which produces the Prolog facts. The second one is a translator written in Prolog and using the CT concept which generates a semantically equivalent model. The third module is an unparser of the Prolog factbase producing pure Eiffel code.

In section 7.3 the transformations are described via the conditional transformation (CT) abstraction. The CTs from the transformation graph were designed to be executed sequentially in order to perform the necessary transformations. The *ANDSEQ*, *ORSEQ* and *PROPSEQ* operators were used to compose the CTs and thus to express the logic of the transformation. The order of some CTs is arbitrary for the transformation. For example, the deletion of feature selection facts can be interchanged with the deletion of foster facts. Some CTs may not be even executed during transformation. For example, when exheriting a feature with no return types the CT responsible with the return type creation will not be executed. The transformations keep as much time as necessary the reverse inheritance information in the model, being eliminated only at the end. This will help us better locate the model elements which need early transformations and avoid using new intermediary facts. In the transformation CT tree were used *ORSEQ* operators and some *PROPSEQ* operators. The main transformation flow is based on a foster class parameter. At the beginning of the transformation sequence some CTs propagate the newly created features in order to equip them with formal arguments and return types.

Further on, we presented a set of transformations on the factbase model which express the

semantics of reverse inheritance. We decided to use a declarative language, Prolog, since the semantics of reverse inheritance could be more easily expressed than using an imperative language based on AST (Abstract Syntax Tree) and Visitor design pattern [GHJV97]. The implementation is based on the CT abstractions which uses parameters and operators to interact with each other.

The feature exheritance is not a trivial process. First the candidate features are computed, next their signatures are compared. In this process the combination of all compatible types is taken into account. If we obtain successfully the list of types from the signature we can start creating the formal arguments and return types. During this creation process, like types are analysed again since there is a chance that the anchor was exherited and the new type could refer it. At this stage new type facts may appear which must be added to the factbase. Next, the feature clauses are transformed in order to produce the equivalent code. Renamings are reversed, features from the subclasses are renamed back to their original names. Redefinition clauses of exheritance are translated into equivalent inheritance redefinition clauses for both new features and redefined features in foster class. The exheritance branches are transformed into inheritance branches using two strategies depending whether the exherited classes are generic or not. The actual generics used to instantiate the exherited classes will be reversed to instantiate the foster class. Finally, the feature selection mechanisms and the foster modifiers are deleted from the factbase.

Chapter 8

Evaluation of the Approach

In this chapter we will compare the reverse inheritance approach to class hierarchy reorganization strategies, class reuse mechanisms, software adaptation and evolution techniques. We will highlight the similarities and differences between the reverse inheritance class relationship and other related mechanisms or reegineering models. The closest concept to reverse inheritance is ordinary inheritance. In some situations reverse inheritance has adapter design pattern capabilities. A concrete reenginering method for creating abstract superclasses can be organized around the reverse inheritance concept. The use of reverse inheritance implies an atomated refactorization process which can make the original code adapt or evolve. Algorithms optimizing feature factorization have the same goals as reverse inheritance, but use different means. Reverse inheritance is also a class reuse concept and is part of class reuse mechanisms like traits and mixins.

8.1 Reverse Inheritance vs. Ordinary Inheritance

The most similar mechanism, which by the way gave birth to reverse inheritance, is ordinary inheritance [Fro02]. While ordinary inheritance represents a top-down approach, reverse inheritance is a bottom-up technique of class organization. Ordinary inheritance affects the subclasses letting the superclasses intact, while reverse inheritance creates the superclass letting subclasses unmodified. We consider that ordinary and reverse inheritance are symmetrical meaning that any class construction built with reverse inheritance can be reproduced with ordinary inheritance. Reverse inheritance introduces no new, incompatible concepts than the already existing and maybe some natural deriving ones like adapt, because adaptations are a must in an environment where classes come from different hierarchies. Common features will be hosted in the foster class and they will be present also in the exherited classes. The behavior of such class hierarchies is the same in the two cases. In inheritance the things are slightly different because common features are declared only in the superclass and they are inherited in the subclasses. Subtyping relationship between classes can be selected in both versions of inheritance (conforming and non-conforming).

Inheritance clauses generally refer to one inherited feature, while in exheritance they may refer to all the exherited features from the exherited classes. The inheritance clauses refer to one feature in the superclass while exheritance clauses may refer one (rename, moveup, adapt) or multiple exherited features (redefine).

8.2 Reverse Inheritance and Design Patterns

In [GHJV97] are presented several design patterns which are a collection of general solutions to commonly occurring problems. The class reorganization strategies may be applied either at design time or may be applied afterwards. The second possibility requires changing the original code but reverse inheritance helps avoiding this. Because design patterns are based basically on polymorphism, dynamic binding, features which are offered by inheritance, we may say that design

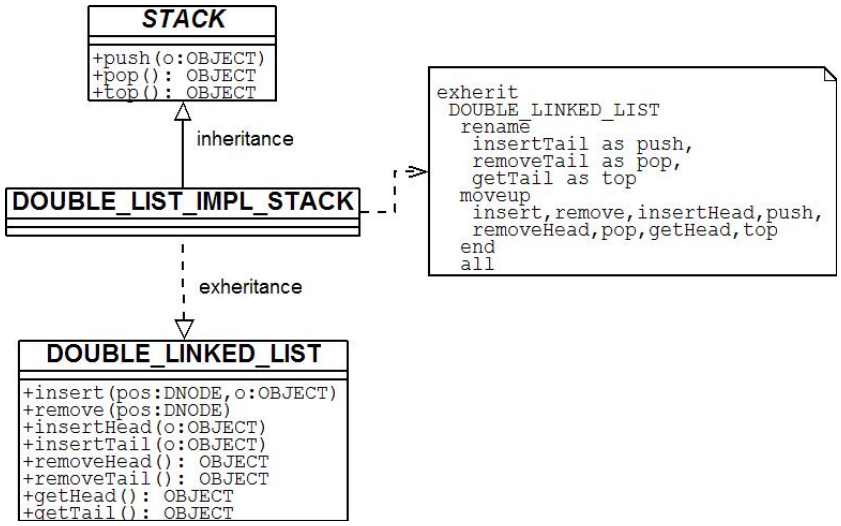


Figure 8.1: Adapter Using Reverse Inheritance

patterns may rely on reverse inheritance also. In some cases reverse inheritance will help building better design solutions than ordinary inheritance. With respect to these ideas we will analyse three examples of design patterns applied using reverse inheritance: adapter, strategy, template method.

Adapter Design Pattern Using Reverse Inheritance Reverse inheritance may help in using the Adapter design pattern in some cases, because it adapts subclass interfaces to the interface of the foster class.

In figure 8.1 and example 128 we show how reverse inheritance can help in the implementation of the Adapter design pattern [GHJV97]. The basic usage of this design pattern is to adapt the interface of a class to a different interface of another class. In our case we want to implement the *STACK* interface using the *DOUBLE_LINKED_LIST* class. For that we created a new class *DOUBLE_LIST_IMPL_STACK* which is the superclass of *DOUBLE_LINKED_LIST* and at the same time a subclass of *STACK*. From the implementation of the adapted class we will import all the necessary features in the foster class by listing them in the **moveup** clause. The case of single inheritance allows exheritance of features in more relaxed conditions than exheritance from multiple classes because we do not have to find the features having the same signature in all exherited classes. We can notice that the methods in the *STACK* interface have different names from the methods of the adapted class, so renaming is used to change their names.

The other possibility is to inherit from *DOUBLE_LINKED_LIST* class, to rename the *insertTail* and *removeTail* methods and to prohibit the inheritance of the other unnecessary features like *insertHead*, *removeHead*.

Template Method Design Pattern Using Reverse Inheritance When moving the implementation of a feature in a foster class we can obtain the effects of the Template method design pattern (see figure 8.2 and example 129).

In figure 8.2 and example 129 we present a situation where we have a transaction class *TRANSACTION_SOCGEN* which implements a checking template method, meaning that each transaction should check the bank, the credit of the owner, the loan the owner may have got, the stock of the bank and the future income the owner may have. All these checkings are implemented in different manners for each particular bank. This checking template can be reused for implementing another transaction for a different bank. Using exheritance we will exherit the implementation of method *check* into a separate superclass TRANSACTION. Also the checking operations are

Example 128 Adapter Using Reverse Inheritance (Eiffel Code)

```
deferred class STACK
feature
 push(o:OBJECT) is deferred end
 pop:OBJECT is deferred end
 top:OBJECT is deferred end
end
class DOUBLE_LIST_IMPL_STACK
inherit
 STACK
exherit
 DOUBLE_LINKED_LIST
  rename
   insertTail as push,
   removeTail as pop,
   getTail as top
  moveup
   insert,remove,insertHead,push,
   removeHead,pop,getHead,top
 end
 all
end
class DOUBLE_LINKED_LIST
feature
 insert(pos:DNODE;o:OBJECT) is do ... end
 remove(pos:DNODE) is do ... end
 insertHead(o:OBJECT) is do ... end
 insertTail(o:OBJECT) is do ... end
 removeHead:OBJECT is do ... end
 removeTail:OBJECT is do ... end
 getHead:OBJECT is do ... end
 getTail:OBJECT is do ... end
end
```

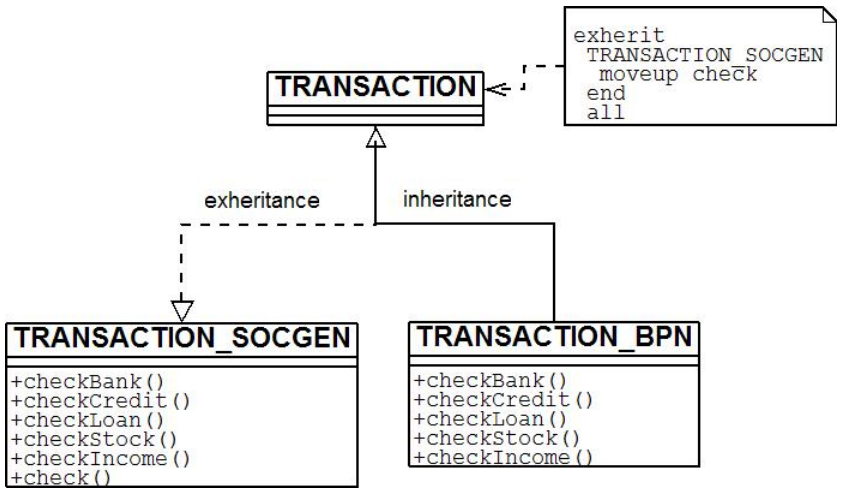


Figure 8.2: Template Method Using Reverse Inheritance

Example 129 Template Method Using Reverse Inheritance (Eiffel Code)

```
class TRANSACTION_SOCGEN
 feature
 checkBank is do end
 checkCredit is do end
 checkLoan is do end
 checkStock is do end
 checkIncome is do end
 check is
 do
  checkBank
  checkCredit
  checkLoan
  checkStock
  checkIncome
 end
end
foster class TRANSACTION
exherit
 TRANSACTION_SOCGEN
  moveup check
 end
all
end
class TRANSACTION_BPN
inherit
 TRANSACTION_SOCGEN
  redefine checkBank,checkCredit,checkLoan,CheckStock,checkIncome
 end
feature
 checkBank is do end
 checkCredit is do end
 checkLoan is do end
 checkStock is do end
 checkIncome is do end
 check is do ... end
end
```

exherited as deferred features since they are needed by the *check* method. From the newly constructed superclass we can inherit the template method and reimplement the checking operations in class *TRANSACTION_BPN*.

Reverse inheritance is a different way of reusing behavior and state from classes, the same thing could be done also by ordinary inheritance. In this case we could non-conformly inherit from class *TRANSACTION_SOCGEN*, redefine the checking operations and not export the other unnecessary features. In this solution we performed just a class reuse operation without having any type relationship between the classes. If we consider to inherit conformly and to prevent the export of unnecessary features then the class instances may be target to invalid CAT[1] calls. The advantage of reverse inheritance solution stands in offering the application designer a new supertype holding the common behaviour and state.

8.3 Reverse Inheritance and Abstract Superclass Creation by Refactorings

In [OJ93] is described a manual method of reorganizing class hierarchies by creating a new abstract superclass for a set of subclasses, using refactorings [Opd92, Fow99]. It is explained step by step the process of creating an abstract superclass: adding function signatures to the superclass, making function bodies compatible, moving variables and migrating common code to the superclass. In our work dedicated to Eiffel we encapsulated all these operations in the semantics of reverse inheritance. The main difference is that the transformations proposed alter the subclasses while our approach keeps them intact, having the possibility of cancelling later easily the modifications. We will show a parallel between the two approaches, so we translated the use case of [Opd92] in order to fit the syntax of Eiffel. In example 130 is presented the equivalent class *MATRIX* in Eiffel.

The original *MATRIX* class contains:

- same attribute for storing state: *rows*, *columns*, *elements*;
- accessor methods for each element of the modelled matrix: *get* and *put* which use a linear formula for indexing;
- a creator method *matrix* which is the equivalent of the C++ constructor;
- special matrix operators like *matrixMultiply*, *rotate*, *matrixInverse*.

In example 131 we created the new class *ABSTRACT_MATRIX* which exherits state and behavior from the original *MATRIX* class. The *rows* and *columns* attributes were exherited as effective. The *elements* attribute is redefined at foster class level as an array of class *ANY*. The *get* and *put* accessors were exherited as deferred since in *SPARSE_MATRIX* class they will have a different implementation based on storing non-null values and their coordinates. The operations methods *matrixMultiply*, *rotate*, *matrixInverse* were exherited as deferred also since they have to be redefined at foster class level using the *get* and *put* accessors. The operation features will act like template methods in the foster class and in the subclasses they will reuse the redefined local accessors.

Later from the *ABSTRACT_MATRIX* class there can be derived the *SPARSE_MATRIX* class presented in example 132. The *elements* attribute has to be redefined as an array of *SPARSE_ELEMENT* instances. The *get* and *put* accessors must be redefined since they have to perform searches in the array containing the values along with their coordinates and not direct indexing like original *MATRIX* class did. A possible implementation for the *SPARSE_ELEMENT* class is presented in example 132 and it must contain attributes and accessor for storing the value and its coordinates.

[1] Changing Availability of Type

Example 130 Initial Matrix Class

```
class MATRIX
create matrix
feature --attributes
 rows, columns : INTEGER
 elements : ARRAY[INTEGER]
feature --accessors
 get(rowNum:INTEGER;colNum:INTEGER):INTEGER is
 do
  result:=elements.item(rowNum * columns + colNum)
 end
 put(newVal:INTEGER;rowNum:INTEGER;colNum:INTEGER) is
 do
  elements.put(rowNum * columns + colNum,newVal)
 end
feature --creators
 matrix(numRows:INTEGER; numCols:INTEGER) is do
  create elements.make(0,9999)
 end
feature --operations
 matrixMultiply(m2:MATRIX):MATRIX is do ... end
 rotate is do ... end
 matrixInverse is do ... end
end
```

Example 131 Abstract Matrix Class

```
foster  class ABSTRACT_MATRIX
exherit
 MATRIX
  moveup rows, columns
 end
 only elements, get, put, matrixMultiply, rotate, matrixInverse
 redefine elements, matrixMultiply, rotate, matrixInverse
feature --attributes
 elements : ARRAY[ANY]
feature --accessors
 get(rowNum,colNum:INTEGER):INTEGER is deferred end
 put(newVal:INTEGER;rowNum:INTEGER;colNum:INTEGER) is deferred end
feature --operations
 matrixMultiply(m2:ABSTRACT_MATRIX):ABSTRACT_MATRIX is
 do
  -- must be reimplemented accessing only get and put
 end
 rotate is
 do
  -- must be reimplemented accessing only get and put
 end
 matrixInverse is
 do
  -- must be reimplemented accessing only get and put
 end
end
```

Example 132 Sparse Matrix Class

```
class SPARSE_MATRIX
inherit
 ABSTRACT_MATRIX
  redefine elements, get, put
 end
feature --attributes
 elements : ARRAY[SPARSE_ELEMENT]
feature --accessors
 get(rowNum,colNum:INTEGER):INTEGER is
 do
  -- a search in the array is performed
 end
 put(newVal:INTEGER;rowNum:INTEGER;colNum:INTEGER) is
 do
  -- a search in the array is performed
 end
end
class SPARSE_ELEMENT
feature --attributes
 rowNum : INTEGER
 colNum : INTEGER
 value : INTEGER
feature --accessors
 getRow:INTEGER is do result:=rowNum end
 setRow(row:INTEGER) is do rowNum:=row end
 getCol:INTEGER is do result:=colNum end
 setCol(col:INTEGER) is do colNum:=col end
 getValue:INTEGER is do result:=value end
 setValue(v:INTEGER) is do value:=v end
end
```

We showed that the matrix abstraction process can be performed successfully with reverse inheritance and ordinary inheritance without affecting the original class. There are some differences due to the fact that reverse inheritance keeps intact the behavior of the exherited classes. The first difference refers to the impossibility of renaming the original *MATRIX* class because of reverse inheritance imposed restrictions.

The exheritance of the *elements* attribute is not present in the [OJ93] example, but we redefined it as an array of *ANY* references at the superclass level in order to be reused. In the sparse matrix class this array is redefined as an array of sparse elements in both approaches. If we choose not to exherit this member in the superclass, we will have to add a new array of sparse elements in the sparse matrix class implementation.

The *columns* and *rows* attributes were listed in the **moveup** clause of reverse inheritance while in the [OJ93] example they moved the attributes manually or automatically. In both approaches the effect will be the same.

The accessor methods *get* and *set* are exherited as deferred in the abstract class and then reimplemented in the sparse matrix class. For this only the selection of features in the exheritance clause was necessary, while in the [OJ93] example they had to explicitly copy the signature in the superclass manually or automatically.

The matrix operations in our approach have to be redefined at foster class level using the accessor methods since in [OJ93] example they are directly modified to use accessors and then it is moved in the superclass. In both approaches they will operate as template methods.

We can conclude that both methods have the same goals for the given example. The [OJ93] method has more flexibility since they use ad-hoc refactoring operations while reverse inheritance has strict rules. The refactorings in [OJ93] method are manual or semiautomatic while in reverse inheritance the refactorings are expressed implicitely by the semantics.

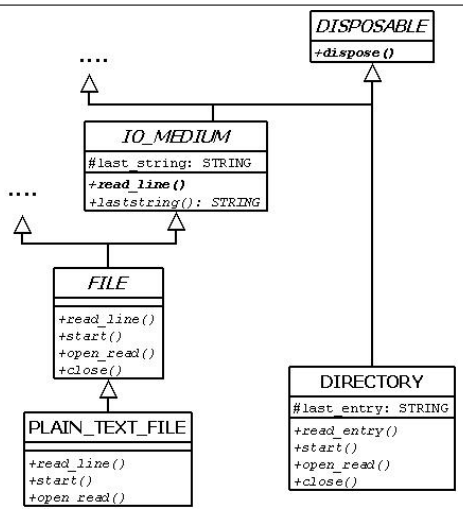
8.4 Reverse Inheritance and Other Class Reuse Mechanisms

In [Fow99] are presented several techniques of restructuring code by altering its internal structure without changing external behavior. We adhere to this restriction in the sense that we do not change the behavior of the exherited classes. Some code reorganization techniques will be used in our work when implementing the semantics of reverse inheritance in terms of equivalent pure language constructs. By proving that each semantical element of reverse inheritance can be expressed using pure Eiffel language constructs we can assure the feasibility of our approach. We mention also that this is not the only possible implementation.

In [DHLR02] is presented an algorithm that reorganizes class hierarchies based on Galois lattice for optimizing factorization of features. In this work the changes are intended to be performed on a class hierarchy in order to avoid flaws regarding factorization. Modifications of attributes to all occurrences is considered time consuming and error prone. Multiple unnecessary declarations of features makes the hierarchy less understandable and usable. In our approach reverse inheritance helps modifying the class hierarchy in order to reflect the new desired model of the application and also to reduce the presence of redundant attributes and methods. The difference between the two approaches is that the reorganization algorithm proposed in [DHLR02] is automatic and it may modify the relations between classes in order to perform optimizations, while exheritance must be used as a tool for redesign first and then by automatic translation the executable system can be obtained.

In [SDN02, SDNB03] is presented the trait model which can be viewed as a class reusing mechanism. Traits are reusable and composable parts of a class which can be connected together. Also traits must respect a connection protocol between them, so they must be designed in a special way. The trait model can be applied only in frameworks in which reusable traits already exist. Reverse inheritance is designed so that it can be applied to any set of class hierarchies written in Eiffel. Comparing the two models, traits are more likely oriented toward designing a system whose parts should be highly reused, while reverse inheritance helps reusing already designed systems.

Example 133 Excerpt of Eiffel Library



8.5 Experimenting with Reverse Inheritance on Eiffel Kernel Library

In previous sections we intend to show the main facets of reverse inheritance and its implementation. The aim of this section is to experience reverse inheritance for implementing the reuse of the Eiffel library[2].

Let us consider the kernel Eiffel library (around 300 classes), especially its kernel (120 classes). In the latter we are interested in an excerpt of this kernel (see figure 133) focusing only on a few classes and features needed by our case-study. Class *DIRECTORY* provides 73 features to handle directories[3].

Class *PLAIN_TEXT_FILE* has a total of 318 features and manages creation and use of plain text files as its name suggests it. Class *FILE* (resp. *IO_MEDIUM*) contains a total of 307 (resp.120) features. Class *DISPOSABLE* contains 33 features coming mainly (31 of them) from class *ANY* which is inherited implicitly by every class.

As figure 133 shows it, class *DIRECTORY* and class *PLAIN_TEXT_FILE* have common functionalities but they are not factorized in common inherited classes. The subset of features shows that both of these classes may open a file or directory in the reading mode (*open_read*), make the cursor at the first position (*start*) and close it (*close*). Class *DIRECTORY* allows access to the name of the files or directories that one directory may contain. The routine *read_entry* reads the next entry and makes it accessible through the attribute *last_entry*. Class *PLAIN_TEXT_FILE* provides *read_line* and *laststring*. They work in the same way as *read_entry* and *last_entry*. Feature *read_line* retrieves the next line of the file and *laststring* is a function allowing access to it.

Let us suppose that we need to write a small program for visiting a file system and printing on the screen the contents of directories (name of included files like the UNIX command "ls") or text files like the UNIX command "cat". To implement this case study in an appropriate way we need to have a common ancestor (we called it *GEN_FILE*) for *PLAIN_TEXT_FILE* and *DIRECTORY* like it is experienced in the source code presented in example 134. It prints on the screen the contents of each plain text file or directory found in the file system.

Routine *get_next* (line *04*) returns an instance of *PLAIN_TEXT_FILE* or *DIRECTORY* which corresponds to the next file to be considered. Due to class *GEN_FILE* we benefit (lines

[2]This library comes with EiffelStudio and may be found at *http://www.eiffel.com/products/studio/*.

[3]The number of features given in this paragraph includes the inherited features, as if the hierarchy was flattened.

Example 134 Implementation of the Case Study

```
01 ...
02 f: GEN_FILE
03 ...recursive visit of the file system
04 f := get_next
05 f.open_read
06 from f.start
07 until stop
08 loop
09 f.read_line
10 stop := f.laststring = void
11 if not stop then
12 io.putstring (f.laststring)
13 end
14 end -- loop
15 ...
```

05 to *12*) from the polymorphism and we do not have to consider differently the two categories of files in the source code.

But this is not as straighforward as it may look like. Figure 133 shows that the features which are needed by class *DIRECTORY* are all declared in this class but this is not the case for class *PLAIN_TEXT_FILE*: some of them are declared in class *IO_MEDIUM* and some others in *FILE*.

To make *GEN_FILE* the direct ancestor of *DIRECTORY* and *PLAIN_TEXT_FILE* with all the common features is then not sufficient because we must avoid the creation of any new inheritance paths in order to not introduce any conflict or additional ambiguities. The consequence is that we may not exherit inherited features, so that a reverse inheritance relationship must target a class where the feature is declared or at least redefined. This is why we define two foster classes instead of only one, with an inheritance relationship between them: *GEN_IO_MEDIUM* and *GEN_FILE* (see figure 8.3).

The source code of these foster classes is shown in figure 135. It is worth to note that the use of single exheritance between *IO_MEDIUM* and *GEN_IO_MEDIUM* (line *03*) would allow without any further constraint to move (using the clause **moveup**) the implementation of the two features into *GEN_IO_MEDIUM*. We did not select this possibility because we do not intend to share the implementation of these features between *DIRECTORY* and *PLAIN_TEXT_FILE*. To select it would force us to undefine them (clause **undefine**) when inheriting from *GEN_IO_MEDIUM* in class *GEN_FILE*.

Moreover the clause **only** (line *04*) is optional. But not using it in *GEN_IO_MEDIUM* would make all the features of *IO_MEDIUM* deferred features of the foster class. In class *GEN_FILE* we use multiple exheritance (lines *12* to *17*). Again, the clause **only** (lines *18-19*) is optional. Not using it would make all factorizable features of *DIRECTORY* and *FILE*, deferred features of *GEN_FILE* (there are *15* of them). Finally, the signature of *read_line* and *read_entry* may not be the same. This would be solved using the clause **adapt** for some scale adaptation but for handling the difference of parameter order and/or parameter number depending on the situation.

Looking to classes *FILE* and *DIRECTORY*, we have such features: *open_read*, *close* and *start* which are present in both classes with the same signature. Class *FILE* contains an attribute *last_string* which contains the last string read by the routine *read_line*. Class *DIRECTORY* includes a routine *read_entry* for reading the content of an entry (mainly the corresponding file name) and a routine *laststring* which allow to access to the last entry read by the previous routine. Then we propose to define the following foster class (see figure 135).

Our compiler **ETransformer** generates the prolog facts (there are 150.000 facts and their overall size is 6 MBytes) in less than 7 seconds. Then we perform the transformation and then the

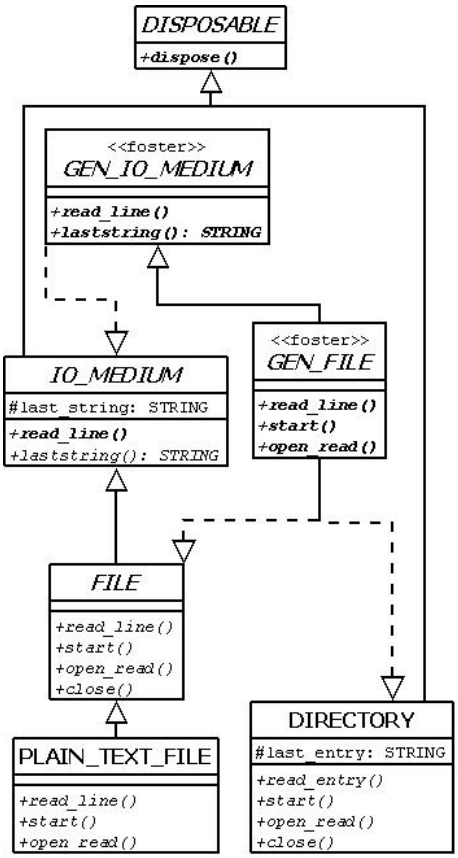


Figure 8.3: Adaptation of the Eiffel Library

Example 135 Foster Classes for Adapting the Library

```
01 deferred foster class GEN_IO_MEDIUM
02 exherit
03  IO_MEDIUM
04  only read_line, laststring
05 feature
06 end -- class GEN_IO_MEDIUM
07
08 foster class GEN_FILE
09 inherit
10  GEN_IO_MEDIUM
11 exherit
12  DIRECTORY
13  rename
14   read_entry as read_line,
15   lastentry as laststring
16  end
17  FILE
18  only open_read, start, read_line,
19   laststring, close
20 feature
21 end -- class GEN_FILE
```

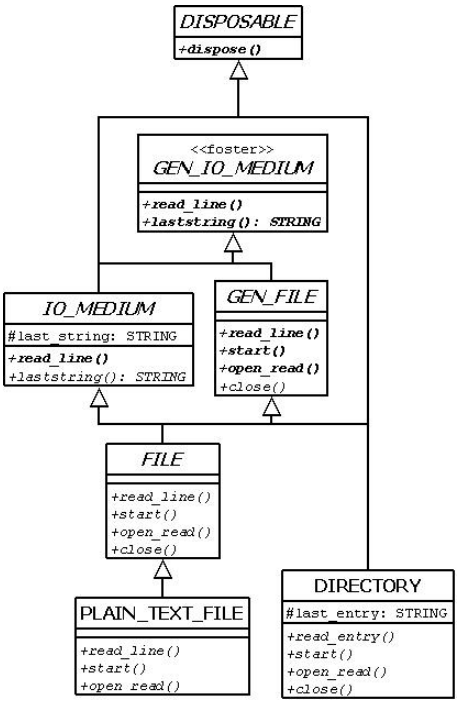


Figure 8.4: Eiffel Library after Transformation

regeneration of Eiffel classes. These two phases take about 90 seconds. It is straighforward that an optimized implementation fully integrated in the Eiffel compiler would have a much smaller overhead. The generated classes (see figure 8.4) are recorded in a temporary directory. Then this directory is given to the Eiffel compiler in order to get the application ready to be executed. Presently the process is not fully automated and the different steps (fact generation, transformation and source (re)generation, Eiffel compilation) are initiated manually by the user but this can be done easily and be fully integrated.

Summary This experiment of our approach suggests a number of comments. It stresses that our first intent is not to refactor a class-hierarchy, but to adapt its content in order to reuse it for the purpose of a given application. Nevertheless our approach by model transformation allows also to generate a new pure Eiffel hierarchy without any reverse inheritance relationship. This new hierarchy could be considered as an evolution of the original one and replace it.

The rules that had been set (e.g. no additional inheritance path, no introduction of new features in foster classes, attribute may not be merged when they do not have a common seed...), may appear as a limitation - for example we had been forced to create two foster classes *GEN_IO_MEDIUM* and *GEN_FILE* instead of one. But these rules are necessary in order to preserve the original semantics of existing Eiffel constructs and the strong-typing. The main outcome is that the behaviour of existing classes is not modified so that it doest not impact the robustness of existing code.

Reverse-inheritance is more than another approach for improving the reuse; it is the counterpart of ordinary inheritance. Accordingly, reverse inheritance is not designed to solve all the problems related to reusability. But, it is fully integrated in the language making it more understandable by Eiffel progammers. The various clauses provided for this new relationship and their possible combinations are in line with the complexity but also the expressiveness of ordinary inheritance.

This makes it fully integrated in the language. The effort for implementing reverse-inheritance relationship within Eiffel is significant but not more than some other interesting features of this

language. The fact that Eiffel is a strongly-typed language increases the complexity of the mechanisms to be implemented but allows a better control of the reuse.

There are two main research directions related to source-code reuse:

i) to consider existing source-code in its current state and to adapt it locally in order to be able to use it for building an application or,

ii) to improve the quality of the source-code in order to make it more reusable.

There is no doubt that object-oriented languages do not fully achieve software reusability even if the contribution is significant thanks to ordinary inheritance, genericity, assertions, etc.

Adaptation of existing classes or their refactoring may be handled either by new paradigms (and some extensions of existing ones), or by external tools provided by the programming environment.

In general refactoring facilities are included in the programming environment whereas adaptation capabilities are more likely introduced in the programming language itself. As it had been mentioned earlier, reverse inheritance belongs to the second category. A lot of other paradigms or language extensions intend to improve software reusability.

Chapter 9

Conclusions and Perspectives

9.1 Contributions

Analysis of the Class Reuse Mechanisms In this work we analysed: the features of ordinary inheritance, the existing elements of reverse inheritance in literature [**C**CL$^+$05d], other reuse mechanisms. We identified some potential uses of reverse inheritance in Java, but with not so much benefits [**C**Pc05, **C**CL05b, **C**RC$^+$06a, CR**C**06b]. In the next paragraphs are presented the main challenges and our proposed solutions, which are scientific contributions, in order to implement this class relationship.

Exheritance Definition We defined reverse inheritance semantics relying on seven principles [SL**C**09]. In the definition several choices were made to facilitate simplicity. The choice for the syntactical elements was made taking into account the expressiveness of the resulted class hierarchy. In order to avoid confusion we pointed out the difference between single, multiple reverse inheritance and several independent reverse inheritance class relationships.

Feature, Type and Implementation Exheritance We designed feature exheritance [**C**CL$^+$04c, **C**CL04a, **C**CL04b, **C**CL$^+$05c] as the main concept in obtaining a uniform interface for all the different subclasses. The choices for factorization are multiple, depending on the number of features that are needed to be exherited. One can choose all possible features except some, which may have the same name but representing the same feature, or can choose no features to be factored implicitly, but the explicitly listed ones. All features or no features may be exherited also. The nature of the feature attribute or method is not important when setting the implicit rules for factorization. For simplicity, we decided that it is better that implicitly both attributes and methods are exherited as deferred features in the foster class.

Another challenge was the definition of type exheritance rules in the exheritance of candidate signatures. We defined type exheritance rules for each sort of type from the Eiffel type system [**C**LS07, SL**C**09]. Identical class types are exherited as such. Class types with actual generics must have the same structure in order to be exherited. Expanded types and separate types must have the same referred class type in order to be exherited. Like types are exherited together with their anchor when possible in the case of feature and formal argument. Otherwise, the target type is used for exheritance. Like current types are exherited as such when all candidates are like current. Bit types are exherited in the foster class using the common manifest integer constant.

When implementation is subject to factorization, we found out that there are quite strong restrictions to be imposed, in order to obtain a valid class hierarchy. We found solutions and decent compromises to all related problems [**C**LS07, SL**C**09]. The exheritance of implementations calling precursor disables implementation reverse inheritance. The main problem refers to dependencies that can be handled either by exheriting them or by reimplementing them at the foster class level, but without affecting the behavior of the original classes. Redefinition of features may also change

the availability of types in the context of foster class. Fortunately such problems can be detected statically, at compile time.

Type Conformance Another important feature of reverse inheritance we pointed out is the type conformance between subclasses and the foster class. We made the substitution principle of polymorphism to work exactly like in ordinary inheritance. Dynamic binding of common features still holds in the context of reverse inheritance. Because of symmetry reasons, we created the concept of non-conformance reverse inheritance. In some versions of the Eiffel language, non-conforming inheritance is used as solution to solve dynamic binding problems. The exheritance class relationship in this case can be used only for reusing implementation from the subclasses. Generic classes instantiated with classes related by reverse inheritance keep their superclass/subclass behavior. Using reverse inheritance between classes working as types, covariant feature redeclarations can be obtained. The expanded status of a class is orthogonal on the ordinary or reverse sort of inheritance. A delicate aspect is related to the behavior added in the superclass, which can be achieved either by reverse inheritance from the subclasses or from a potential superclass. Our approach is strongly based on the fact that in ordinary and reverse inheritance features can be redefined. We restricted some exheritance clause combinations in context of attributes, methods and mixes of attributes and methods. Moving a candidate feature and redefining another candidate, or moving at the same time several candidate features are not valid actions.

Feature Adaptations, Genericity and Assertions Redefinition Since adaptations are the core of the exheritance mechanism permitting the use of several different classes under a common interface, we dedicated a special chapter to present the related problematics. First we presented the classic adaptation mechanisms like redefinition, undefinition and renaming and how they work in the context of reverse inheritance. We designed a special set of adaptations [**C**CL05a, **C**CL07a, **C**CL07b] the ones related to signatures, which may be different and still have a common signature in the foster class. In [LHQ94] was presented an example of parameter order adaptation without any syntax definition and underlying implementation. Our syntax for this mechanism allows adaptation code to be written for formal arguments and return types. We consider that our designed adaptations are not a severe deviation from the philosophy of the language.

Regarding genericity we analysed the cases of unconstrained and constrained generic subclasses. We allowed for a foster class to instantiate and exherit several generic classes. If the foster class is also generic then non-generic features and also generic features can be exherited, since there is no constraint on the generic parameters and they can be instantiated with any type. A special and useful situation of reuse arises in case the foster class is generic and the subclasses are non-generic, and some concrete features from the subclasses are exherited as generic features. This behavior is somehow asymmetric related to ordinary inheritance and probably difficult to implement. For constrained genericity exheritance cases, in order to be able to factor generic features there must be a conforming supertype of all corresponding generic parameter constrained type. If such a type does not exist, we can always provided it by reverse inheritance.

The redefinition of assertions (preconditions, postconditions and invariants) is a very problematic issue related to reverse inheritance. Precondition in foster class must be stronger than those in subclasses, so the **AND** logical operator must be used [LHQ94]. This approach is not always applicable when preconditions from the subclasses are contradictory. **False** is the strongest precondition, but it will forbid method code execution, resulting failure. For postconditions and invariants the **OR** logical operator should be used. The problem of postconditions is not so severe since **true** is the weakest postcondition that always checks. Another problem arises for all assertions when a feature present in the logical expression is not exherited in the foster class [LHQ94]. For this problem we proposed as strategy to eliminate the missing logical variable, replacing it with a neutral constant (**true** or **false**), trying to affect as less as possible the logical expression. However, the burden of guaranteeing which assertion is correct, is left in the responsibility of the programmer. Using such keywords makes the programmer aware. A different approach, which we implemented in the prototype, is to create a combined precondition and postcondition using

the cheating server technique inspired from [Int06], but the non-exheritable features would still make impossible the exheritance of that feature. This technique implies creating a complex assertion expression based on static type checking of the current instance and the execution of the corresponding assertion found in the subclass.

Coupling Exheritance with Inheritance We studied several class configurations made of foster classes and ordinary classes. The most interesting case is the one of amphibious features which cannot be prevented from exheritance. In such cases the prototype generates equivalent class hierarchies with the appropriate inheritance clauses in order to avoid implementation conflicts due to the newly added class.

The classic diamond multiple inheritance class configuration can be obtained also using reverse inheritance, but adding the classes in different orders, thus having different time stamps. In the case of sharing multiple inherited features there are no problems because the features share the same seed, but when replication is needed and the class in the top of the hierarchy is added last, then the selection of the appropriate implementation for a replicated feature in the bottom class must be specified in the top class, since it is the last one added. The classic solution of using **select** for solving dynamic binding problems in ordinary inheritance has some drawbacks in more complex class hierarchies, by not permitting a free selection of a desired implementation for a feature. This issue is not solved in inheritance nor in exheritance.

We analysed the impact of some keywords related to the status of features and classes in the context of reverse inheritance. The most interesting one is related to the use of **precursor**. Related to feature export, our conclusion is that the foster class can restrict the set of subclass common clients, but not to allow new ones. Creational procedures can be exherited, but in the foster class they must be also declared in the creation procedure list. The assign clause of an attribute can be exherited together with the attribute, if they are present in all the subclasses.

Thus, we defined our semantics for the class reuse mechanism fulfilling our first objective.

Prototype Implementation We decided to integrate reverse inheritance in Eiffel because of its philosophy, it includes multiple inheritance, unique names for features, redefinition clauses. Regarding the implementation architecture [**C**KLS07], firstly we designed the reader module of the prototype which translates RIEiffel into Prolog by extending the grammar of a GOBO compiler with reverse inheritance specific syntax. We added new syntax for: the foster keyword in the class header, the exheritance branch with all its clauses, the feature selection mechanism after the inheritance/exheritance section, the adaptation syntax in the body of a method. We modelled all the entities of the RIEiffel language in Prolog including both ordinary and reverse inheritance. With respect to the ideea of generating Prolog representation from program source code we developped an automated grammar driven approach applicable to any programming language [**C**JM08, J**C**M08].

In the second transformation module we implemented the designed concepts of reverse inheritance by Prolog CTs[1] which manipulate the original source code in form of facts. Each informal rule in the semantics part has its counter part in the set of Prolog CTs executed by the CTC transformation engine. All the semantics from the previous paragraphs is implemented by CTs written in Prolog.

Finally, we wrote an unparser, the writer module, to translate back Prolog facts into pure Eiffel source code. Arround the prototype modules we designed a testing framework, animated by Perl scripts, using output testing and structural testing strategies [**C**JF09]. We consider that implementing reverse inheritance semantics by CTs executed by the CTC engine is a validation of the transformation engine itself.

Conditional Transformations Implementing Reverse Inheritance Semantics We designed a set of CTs for common signature feature exheritance. The feature signature is decom-

[1] The Conditional Transformation is a pair of a precondition and a transformation action written in Prolog and processed by the CTC transformation engine.

posed into: name, argument and return type. All these are analyzed in order to produce a common signature for the new feature in the superclass. We wrote the type exheritance CTs which take into account our proposed rules related to the whole Eiffel type system and determines the representant type for the superclass computed from a set of subclass types. We wrote these CTs in order to generate static type checking conditions for preconditions and postconditions. Our export CTs computes the common set of client classes for an exherited feature. Using these CTs we generate the mediator feature and the glue code in order to adapt attributes and methods. Our class reorganization CTs transform exheritance into inheritance branches and also deals with genericity: creates new types and instantiates them as needed. Usually, new language concepts are implemented in small experimental languages which are easy to prototype. We integrated reverse inheritance in Eiffel which is an industrial strength programming language.

Thus, we showed that reverse inheritance is implemented in an industrial strength language achieving our second objective.

A Posteriori Class Reusability According to our objectives we designed reverse inheritance with a posteriori class reusability facilities like: i) *to create new abstract supertypes*; ii) *to factor features from classes*; iii) *to combine the implementations of features.* The use of the class relationship creates a common supertype for the subclasses through which these classes can be manipulated. By common features factorization through implementation exheritance and later by inheritance in new descendant classes we provide free reuse of the exherited code with new code in two ways: by calling the exherited code or by redefining its calls.

Class Evolution and Adaptation By reverse inheritance we offer also facilities which promote class evolution and adaptation: i) *to redefine the implementations of features*; ii) *to adapt the implementation of features*; iii) *to cancel the implementation of a feature.* Redefining a feature in a superclass with a general behavior facilitates building healthy and evolved class hierarchies. Our adaptation mechanism allows adjusting different feature signatures to a common one, in order to be uniformly reused. We consider that this mechanism makes the classes evolve. Cancelling the implementation of a feature is a natural concept for generalization class relationship.

A Posteriori Class Hierarchy Reorganization Our concept of reverse inheritance helps also a posteriori class hierarchy reorganization by *adding an abstraction layer into a class hierarchy.* We designed the class relationship with the capability of combining it with ordinary inheritance, thus building complex class hierarchies. For the unforseen design changes reverse inheritance together with ordinary inheritance makes possible to add a new class between two existing ones without affecting the rest of existing classes.

Pragmatics Assessing the principles of ordinary inheritance, the backbone of object-oriented technology, will facilitate learning easily the principles of reverse inheritance. We base this idea on the symmetry between the two class relationships. Our grammar extension is quite small containing only a few rules. Renaming, undefinition and redefinition concepts have the same semantics and the same syntax for reverse inheritance.

Thus, we showed that reverse inheritance has evolutionary and adaptability capabilities, accomplishing our third objective.

Feasible and Effective Class Reuse Mechanism Taking into account all developed reverse inheritance features at both conceptual rules and prototype level, the thesis of this work is that ***reverse inheritance class relationship is a feasible and effective class reuse mechanism***. According to our accomplished objectives ***we fulfilled our main goal of class reuse***.

9.2 Future Work

Like Type Class Relationship A first perspective is to develop and explore further the like type class reuse mechanism, which allows reuse in a slightly different manner. Single non-conforming reverse inheritance is equivalent to single like type.

Formal Semantics Another perspective is to define the reverse inheritance of Eiffel semantics in relation to the one of ordinary inheritance using formalisms based on lattices. Thus, the soundness of the type system can be proved in the context of reverse inheritance.

Implementation Optimization A different perspective is linked to the optimization of the current prototype implementation. We can optimize transformation speed by rewriting some of the generic transformations. For example, the generic implementation for cloning an instruction list can be done with specific cloning rules. Thus, at each node in the AST only a small subset of facts corresponding to that node are iterated and not all the facts from the factbase.

Module Generalization Since the grammars for most programming languages are available freely on the Internet, the reader and the writer modules could be automatically generated for any language by a grammar driven approach. In this direction we did some efforts resulting Prolog facts, following a strict generation pattern, very close to the AST. This generated Prolog model is very rigid including unnecessary facts for some tokens and rules. A specification of mapping rules between the grammar and the Prolog facts would permit the generation of customized Prolog facts, thus eliminating all unwanted details.

Enhanced Portability Another perspective for the reverse inheritance extension is to be ported on the latest versions of the GOBO Eiffel library [Bez07], as an experimental feature. We intend to design the integration schema to be as independent as possible of the GOBO library further evolution. The RIEiffel grammar files will override the standard ones from the library, the AST files must be regenerated using the **gelex** and **geyacc** tools. The source files containing the classes modelling the Prolog facts can be copied in a new folder of the existing directory structure. Such an integration task could be easily automated by a shell script.

Developer Documentation A special attention should be given to technical writing. All prototype modules should be described in details for any other future developers.

Bibliography

[AAS01] Walid Al-Ahmad and Eric Steegmans. Integrating extension and specialization inheritance. *Journal of Object-Oriented Programming*, December 2001.

[AB91] Serge Abiteboul and Anthony Bonner. Objects and views. In *SIGMOD'91 Conference Proceedings, International Conference on Management of Data*, pages 238–247, San Francisco, California, March 1991. ACM Press.

[ABV92] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An object-oriented language-database integration model: The composition-filters approach. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 372–395, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[AG00] K. Arnold and J. Gosling. *The Java Programming Language*. Sun Microsystems, 3rd edition, USA, 2000.

[Agh86] Gul Agha. An overview of actor languages. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, pages 58–67, New York, NY, USA, 1986. ACM Press.

[Aks96] M. Aksit. Separation and composition of concerns in the object-oriented model. *ACM Comput. Surv.*, 28(4es):148, 1996.

[BDMN79] G.M. Birtwhistle, O.J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Chartwell-Bratt Ltd, 1979.

[BDW03] Alexander Bergel, Stephane Ducasse, and Roel Wuyts. Classboxes: A minimal module supporting local rebinding. In *JMLC'03: Proceedings of Joint Modular Language Conference*. Springer, 2003.

[Bez07] Eric Bezault. GOBO Eiffel Project. http://www.gobosoft.com, November 2007.

[BI08] Borland International. Borland C++ 5.5 - Command-line compiler, 2008.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications. Second Edition*. Addison-Wesley, 1994.

[**C**CL04a] **Ciprian-Bogdan Chirila**, Pierre Crescenzo, and Philippe Lahire. A reverse inheritance relationship dedicated to reengineering: The point of view of feature factorization. In *MASPEGHI Workshop at ECOOP 2004, MechAnisms for SPEcialization, Generalization and inHerItance*, Oslo, Norway, June 2004.

[**C**CL04b] **Ciprian-Bogdan Chirila**, Pierre Crescenzo, and Philippe Lahire. A reverse inheritance relationship for improving reusability and evolution: The point of view of feature factorization. Research Report, 6 pages, ISRN I3S/RR-2004-22-FR, OCL Project, Laboratoire Informatique, Signaux et Systmes de Sophia-Antipolis (UNSA / CNRS), France, http://www.i3s.unice.fr/~mh/RR/2004/liste-2004.html, September 2004.

[CCL+04c] **Ciprian-Bogdan Chirila**, Pierre Crescenzo, Philippe Lahire, Dan Pescaru, and Emanuel Ţundrea. Factoring mechanism of reverse inheritance. In *International Conference on Technical Informatics CONTI 2004, Periodica Politechnica, Transactions on Automatic Control and Computer Science, ISSN 1224-600X*, volume 49, pages 131–136, Timisoara, Romania, May 2004.

[CCL05a] **Ciprian-Bogdan Chirila**, Pierre Crescenzo, and Philippe Lahire. Reverse inheritance: An approach for modeling adaptation and evolution of applications. Research Report, 14 pages, ISRN I3S/RR-2005-05-FR, OCL Project, Laboratoire Informatique, Signaux et Systmes de Sophia-Antipolis (UNSA / CNRS), France, http://www.i3s.unice.fr/~mh/RR/2005/liste-2005.html, February 2005.

[CCL05b] **Ciprian-Bogdan Chirila**, Pierre Crescenzo, and Philippe Lahire. Towards reengineering: An approach based on reverse inheritance - application to java. Research Report, 70 pages, I3S/RR-2005-05-FR, OCL Project, Laboratoire Informatique, Signaux et Systmes de Sophia-Antipolis (UNSA / CNRS), France, http://www.i3s.unice.fr/~mh/RR/2005/liste-2005.html, September 2005.

[CCL+05c] **Ciprian-Bogdan Chirila**, Pierre Crescenzo, Philippe Lahire, Dan Pescaru, and Emanuel Ţundrea. Factoring mechanism of reverse inheritance. Research Report, 6 pages, ISRN I3S/RR-2005-05-FR, OCL Project, Laboratoire Informatique, Signaux et Systmes de Sophia-Antipolis (UNSA / CNRS), France, http://www.i3s.unice.fr/~mh/RR/2005/liste-2005.html, September 2005.

[CCL+05d] **Ciprian-Bogdan Chirila**, Pierre Crescenzo, Philippe Lahire, Dan Pescaru, and Emanuel Ţundrea. Survey on reverse inheritance. *Scientific Bulletin of Politehnica University of Timisoara, Transactions on Automatic Control and Computer Science, Vol. 50 (64), ISSN 1224-600X*, 2005.

[CCL07a] **Ciprian-Bogdan Chirila**, Pierre Crescenzo, and Philippe Lahire. Reverse inheritance: Improving class library reuse in Eiffel. Research Report, 14 pages, ISRN I3S/RR-2007-10-FR, OCL Project, Laboratoire Informatique, Signaux et Systmes de Sophia-Antipolis (UNSA / CNRS), France, http://www.i3s.unice.fr/~mh/RR/2007/liste-2007.html, March 2007.

[CCL07b] **Ciprian-Bogdan Chirila**, Pierre Crescenzo, and Philippe Lahire. Reverse inheritance: Improving class library reuse in Eiffel. Poster presentation at LMO (Langages et Modeles a Objets) 2007 Conference, May 2007.

[CJF09] **Ciprian-Bogdan Chirila**, Calin Jebeleanu, and Krisztina Francz. Testing techniques for a logic representation generator. In *In Proceedings of 2009 IEEE 5-th International Conference on Intelligent Computer Communication and Processing, ICCP 2009,* **IEEE Xplore Rated**, pages 207–210, Cluj-Napoca, Romania, August 27-29 2009.

[CJM08] **Ciprian-Bogdan Chirila**, Calin Jebelean, and Anca Maduta. Towards automatic generation and regeneration of logic representation for object-oriented programming languages. In *In Proceedings of International Conference on Technical Informatics, CONTI 2008*, volume 2, pages 13–18, Timisoara, Romania, June 5-6 2008. Politehnica Publishing House Timisoara.

[CKLS07] **Ciprian-Bogdan Chirila**, Gunter Kniesel, Philippe Lahire, and Markku Sakkinen. RIEiffel Project Website. http://nyx.unice.fr:9000/trac, December 2007.

[CLS07] **Ciprian-Bogdan Chirila**, Philippe Lahire, and Markku Sakkinen. Towards fully-fledged reverse inheritance in Eiffel. Research Report ISRN I3S/RR-2007-12-FR, OCL Project, Laboratoire Informatique, Signaux et Systmes de Sophia-Antipolis (UNSA / CNRS), France, http://www.i3s.unice.fr/~mh/RR/2007/liste-2007.html, March 2007.

[CMR02] Yania Crespo, Jos Manuel Marques, and Juan Jos Rodriguez. On the translation of multiple inheritance hierarchies into single inheritance hierarchies. In *In European Conference on Object-Oriented Programming*, 2002.

[Cor08a] Microsoft Corporation. Microsoft Windows, 2008.

[Cor08b] Microsoft Corporation. Visual C++, 2008.

[**C**Pc05] **Ciprian-Bogdan Chirila**, Dan Pescaru, and Emanuel Ţundrea. Foster class model. In *In Proceedings of SACI 2005 2nd Romanian-Hungarian Joint Symposium on Applied Computational Intelligence, ISBN 963-7154-39-6*, pages 265–272, Timisoara, Romania, May 2005.

[**C**RC$^+$06a] **Ciprian-Bogdan Chirila**, Monica-Naomi Ruzsilla, Pierre Crescenzo, Philippe Lahire, Dan Pescaru, and Emanuel Ţundrea. Towards a reengineering tool for Java based on reverse inheritance. In *In Proceedings of SACI 2006 the 3-rd Romanian-Hungarian Joint Symposium on Applied Computational Intelligence, ISBN 963-7154-46-9*, pages 364–375, Timisoara, Romania, May 2006.

[CR**C**06b] Smaranda-Claudia Chirila, Monica-Naomi Ruzsilla, and **Ciprian-Bogdan Chirila**. Reverse inheritance features applied in coding Java mobiles applications. In *In Proceedings of International Conference on Technical Informatics - CONTI'2006, ISBN (10): 973-625-319-8 ISBN (13): 978-973-625-319-5*, volume 2, pages 43–46, Timisoara, Romania, June 2006.

[CRM99] Yania Crespo, Juan Jos Rodriguez, and Jos Manuel Marques. Obtaining generic classes automatically through a parameterization operator. a focus on constrained genericity. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, volume 31, pages 166–176, 1999.

[DHLR02] Michel Dao, Marianne Huchard, Therese Libourel, and Cyril Roume. Evaluating and optimizing factorization in inheritance hierarchies. In *Proceedings of the Inheritance Workshop at ECOOP 2002*, Malaga, Spain, June 2002.

[FOP04] Oleksandr Fuks, Jonathan S. Ostroff, and Richard F. Paige. SECG: The SCOOP-to-Eiffel code generator. *Journal of Object Technology*, 3:143–160, 2004.

[Fou95] GNU Software Foundation. Flex - a fast scanner generator. http://www.gnu.org/software/flex, March 1995.

[Fou06] GNU Software Foundation. Bison - GNU parser generator. http://www.gnu.org/software/bison, 2006.

[Fow97] Martin Fowler. Dealing with roles. In *Inproceedings of the 4-th Annual Conference on the Pattern Languages of Programs*, Monticello, Illinois, USA, September 1997.

[Fow99] Martin Fowler. *Refactoring Second Edition*. Addison-Wesley, 1999.

[FPB$^+$02] Jeff Ferguson, Brian Patterson, Jason Beres, Pierre Boutquin, and Meeta Gupta. *C# Bible*. Wiley Publishing, Inc., 10475 Crosspoint Boulevard, Indianapolis, IN 46256, 2002.

[Fro02] Peter H. Frohlich. Inheritance decomposed. In *Proceedings of the Inheritance Workshop at ECOOP 2002*, Malaga, Spain, June 2002.

[GHJV97] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

[GSR96] Georg Gottlob, Michael Schrefl, and Brigitte Rock. Extending object-oriented systems with roles. In *ACM Transactions on Information Systems*, volume 14, pages 268–296, July 1996.

[Hil99] Rich Hillard. View and viewpoints in software systems architecture. In *First Working IFIP Conference on Software Arhitecture (WICSA 1)*, pages 22–24, San Antonio, Texas, February 1999.

[HN96] Michael Van Hilst and David Notkin. Using role components to implement collaboration-based design. In *Proceedings of Conference on Object-Oriented Programing, Systems, Languages and Applications (OOPSLA'96)*, California, USA, 1996. ACM Press.

[Int06] ECMA International. Standard ECMA-367 Eiffel: Analysis, design and programming language. www.ecma-international.org, June 2006.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press, 1992.

[JCM08] Calin Jebelean, **Ciprian-Bogdan Chirila**, and Anca Maduta. Generating logic based representation for programs. In *In Proceedings of 2008 IEEE 4-th International Conference on Intelligent Computer Communication and Processing, ICCP 2008,* **ISI, IEEE Xplore Rated**, pages 145–151, Cluj-Napoca, Romania, August 28-30 2008.

[Joh79] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[Kee89] Sonja E. Keene. *Object-Oriented Programming in Common Lisp. A Programmer's Guide to CLOS*. Addisson Westley, 1989.

[Ken99] Elisabeth A. Kendall. Role model designs and implementations with aspect-oriented programming. In *Proceedings of the 1999 Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, Denver, Colorado, USA, November 1999.

[KK02] Günter Kniesel and Helge Koch. ConTraCT - conditional transformations for incremental compilation of aspects. Demonstration at 1st International Conference on Aspect-Oriented Software Development, June 2002.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[Kni06] Günter Kniesel. A logic foundation for conditional program transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn, January 2006.

[KR93] Harumi A. Kuno and Elke A. Rundensteiner. Developing an object-oriented view management system. In *IBM Centre for Advanced Studies Conference archive Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering*, volume 1, pages 548–562, Toronto, Ontario, Canada, July 1993.

[Kri96] B. B. Kristensen. Object-oriented modelling with roles. In *Object Oriented Information Systems*, Dublin, Ireland, 1996.

[Lad02] Ramnivas Laddad. I want my AOP. *Java World*, January 2002.

[LHQ94] Ted Lawson, Christine Hollinshead, and Munib Qutaishat. The potential for reverse type inheritance in Eiffel. In *Technology of Object-Oriented Languages and Systems (TOOLS'94)*, 1994.

[LW94] Barbara Liskov and Jeanette Wing. A behavioural notion of subtyping. In *ACM Transactions on Programming Languages and Systems*, November 1994.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction 2nd ed.* Prentice Hall, 1997.

[Mey02] Bertrand Meyer. Eiffel: The language. http://www.inf.ethz.ch/ meyer/, September 2002.

[MS98] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1445, pages 355–382. Springer-Verlag, 1998.

[OJ93] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring, 1993.

[OMG04] Object Management Group. UML Superstructure version 2.0. www.omg.org/uml, October 2004.

[Opd92] William Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Ped89] C. H. Pedersen. Extending ordinary inheritance schemes to include generalization. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 407–417. ACM Press, 1989.

[Pon02] Claudia Pons. Generalization Relation in UML Model Elements. In *Inheritance Workshop of European Conference on Object-Oriented Programming*, 2002.

[Sak02] Markku Sakkinen. Exheritance - Class generalization revived. In *Proceedings of the Inheritance Workshop at ECOOP*, Malaga, Spain, June 2002.

[SB00] Yannis Smaragdakis and Don Batory. Mixin-based programming in C++. In *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE)*, 2000.

[SB02] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[Sch98] Herbert Schildt. *C++ The Complete Reference, Third Edition.* McGraw-Hill, 1998.

[SDN02] Nathanael Schärli, Stéphane Ducasse, and Oscar Nierstrasz. Classes = traits + states + glue (beyond mixins and multiple inheritance). In *Proceedings of the International Workshop on Inheritance*, Malaga, Spain, June 2002.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proceedings of the Inheritance Workshop at ECOOP 2003*, Darmstadt, Germany, July 2003.

[SF08] GNU Software Fundation. GCC, the GNU compiler collection, 2008.

[Ska93] John Max Skaller. Mixin article from the comp.lang.c++ newsgroup. http://cpptips.hyperformix.com/cpptips/mixins, 1993.

[SLC09] Markku Sakkinen, Philippe Lahire, and **Ciprian-Bogdan Chirila**. Towards fully-fledged reverse inheritance in Eiffel. In *In Proceedings of 11th Symposium on Programming Languages and Software Tools SPLST 09 and 7th Nordic Workshop on Model Driven Software Engineering NW-MODE 09*, pages 132–146, Tampere, Finland, 2009.

[SN88] Michael Schrefl and Erich J. Neuhold. Object class definition by generalization using upward inheritance. In *IEEE Transactions*, 1988.

[Sof08] Eiffel Software. Eiffel studio, 2008.

[SS77] John Miles Smith and Diane C.P. Smith. Database Abstractions: Aggregation and Generalization. In *ACM Transactions on Database Systems*, volume 2, pages 105–133, June 1977.

[Str97] Bjarne Stroustrup. *The C++ Programming Language Third Edition*. Addison-Wesley, 1997.

[Str02] Bjarne Stroustrup. Multiple inheritance for c++. In *European UNIX Users' Group Conference*, Helsinky, Finland, May 2002.

[Tai96] Antero Taivalsaari. On the notion of inheritance. In *ACM Computing Surveys, No. 3*, volume 28, September 1996.

[Tea03] The AspectJ Team. The aspectj programming guide. Technical report, Xerox Corporation, Palo Alto Research Center, Incorporated, 2003.

[TOG08] The Open Group. The Unix System, 2008.

[TUI05] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyama. An adaptive object model with dynamic role binding. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 166–175, New York, NY, USA, May 2005. ACM Press.

[VN96] Michael VanHilst and David Notkin. Using C++ templates to implement role-based designs. In *ISOTAS '96: Proceedings of the Second JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37, London, UK, 1996. Springer-Verlag.

[WSM06] Alessandro Warth, Milan Stanojevic, and Todd Millstein. Statically scoped object adaption with expanders. In *Proceedings of Conference on Object-Oriented Programing, Systems, Languages and Applications (OOPSLA'06)*, Portland, Oregon, USA, 2006. ACM Press.

[WZ88] P. Wegner and S. B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *on ECOOP '88 (European Conference on Object-Oriented Programming)*, pages 55–77, London, UK, 1988. Springer-Verlag.

Appendix A

Eiffel Reverse Inheritance BNF Grammar Rules

```
Class_declaration: Indexing_opt Class_header Formal_generics_opt Obsolete_opt
Inheritance_opt Exheritance_opt Creators_opt Features_opt Invariant_opt E_END
Indexing_opt: -- /* empty */
| E_INDEXING Index_list
Index_list: -- /* empty */
| Index_clause
| Index_list Index_clause
| Index_list ';' Index_clause
Index_clause: Index_terms
| E_IDENTIFIER ':' Index_terms
Index_terms: Index_value
| Index_terms ',' Index_value
Index_value: E_IDENTIFIER
| Manifest_constant
Class_header: Header_mark_opt E_CLASS E_IDENTIFIER
Header_mark_opt: -- /* empty */
| E_DEFERRED [E_FOSTER]
| E_EXPANDED [E_FOSTER]
| E_SEPARATE
| E_FOSTER
Formal_generics_opt: -- /* empty */
| '[' Formal_generic_list ']'
Formal_generic_list: -- /* empty */
| E_IDENTIFIER Constraint_opt
| Formal_generic_list ',' E_IDENTIFIER Constraint_opt
Constraint_opt: -- /* empty */
| E_ARROW Class_type
Obsolete_opt: -- /* empty */
| E_OBSOLETE E_STRING
Inheritance_opt: -- /* empty */
| E_INHERIT Parent_list
Parent_list: -- /* empty */
| Parent
| Parent_list Parent
| Parent_list ';' Parent
```

```
Parent: Class_type Feature_adaptation_opt
Feature_adaptation_opt: -- /* empty */
| Feature_adaptation1
| Feature_adaptation2
| Feature_adaptation3
| Feature_adaptation4
| Feature_adaptation5
Feature_adaptation1: Rename New_exports_opt Undefine_opt Redefine_opt
Select_opt E_END
Feature_adaptation2: New_exports Undefine_opt Redefine_opt Select_opt E_END
Feature_adaptation3: Undefine Redefine_opt Select_opt E_END
Feature_adaptation4: Redefine Select_opt E_END
Feature_adaptation5: Select E_END
Rename: E_RENAME Rename_list
Rename_list: -- /* empty */
| Feature_name E_AS Feature_name
| Rename_list ',' Feature_name E_AS Feature_name
New_exports: E_EXPORT New_export_list
New_exports_opt: -- /* empty */
| New_exports
New_export_list: -- /* empty */
| New_export_item
| New_export_list New_export_item
| New_export_list ';' New_export_item
New_export_item: Clients Feature_set
Feature_set: Feature_list
| E_ALL
Feature_list: -- /* empty */
| Feature_name
| Feature_list ',' Feature_name
Clients: '{' Class_list '}'
Clients_opt: -- /* empty */
| Clients
Class_list: -- /* empty */
| E_IDENTIFIER
| Class_list ',' E_IDENTIFIER
Redefine: E_REDEFINE Feature_list
Redefine_opt: -- /* empty */
| Redefine
Undefine: E_UNDEFINE Feature_list
Undefine_opt: -- /* empty */
| Undefine
Select: E_SELECT Feature_list
Select_opt: -- /* empty */
| Select
Exheritance_opt: -- /* empty */
| E_EXHERIT Heir_list Exherited_feature_list Foster_adaptation_opt
Heir_list: -- /* empty */
| Heir
| Heir_list Heir
| Heir_list ';' Heir
Heir: Class_type Feature_adaptation_opt
Feature_adaptation_opt: -- /* empty */
| Feature_adaptation11
```

```
| Feature_adaptation12
| Feature_adaptation13
| Feature_adaptation14
Feature_adaptation11: Rename Adapt_opt Moveup_opt Select_RI_opt E_END
Feature_adaptation12: Adapt Moveup_opt Select_RI_opt E_END
Feature_adaptation13: Moveup Select_RI_opt E_END
Feature_adaptation14: Select_RI E_END
Adapt: E_ADAPT Feature_list
Adapt_opt: -- /* empty */
| Adapt
Moveup: E_MOVEUP Feature_list
Moveup_opt: -- /* empty */
| Moveup
Select_RI: E_SELECT Qualified_Feature_list
Select_RI_opt: -- /* empty */
| Select_RI
Qualified_feature_list: Feature_list_RI
| Qualified_feature_list ';' Feature_list_RI
Feature_list_RI: Feature_list
| Descendant_qualification ':' Feature_list
Descendant_qualification: Class_name
| Descendant_qualification '.' Class_name
Exherited_feature_list:
E_ONLY Feature_list | E_EXCEPT Feature_list | E_ALL | E_NOTHING
Foster_adaptation_opt:
New_exports_opt Redefine_opt
-- New_exports_opt from Inheritance
-- Redefine_opt from Inheritance
Creators_opt: -- /* empty */
| Creation_clause
| Creators_opt Creation_clause
Creation_clause: E_CREATION Clients_opt Procedure_list
Procedure_list: -- /* empty */
| E_IDENTIFIER
| Procedure_list ',' E_IDENTIFIER
Features_opt: -- /* empty */
| Feature_clause
| Features_opt Feature_clause
Feature_clause: E_FEATURE Clients_opt Feature_declaration_list
Feature_declaration_list: -- /* empty */
| Feature_declaration
| Feature_declaration_list Feature_declaration
| Feature_declaration_list ';' Feature_declaration
Feature_declaration: New_feature_list Declaration_body
Declaration_body: Formal_arguments_opt Type_mark_opt Constant_or_routine_opt
Adapted_opt
Constant_or_routine_opt: -- /* empty */
| E_IS Feature_value
Feature_value: Manifest_constant
| E_UNIQUE
| Routine
New_feature_list: New_feature
| New_feature_list ',' New_feature
New_feature: Feature_name
```

```
| E_FROZEN Feature_name
Feature_name: E_IDENTIFIER
| E_PREFIX E_STRING
| E_INFIX E_STRING
Formal_arguments_opt: -- /* empty */
| '(' Entity_declaration_list ')'
Entity_declaration_list: -- /* empty */
| Entity_declaration_group
| Entity_declaration_list Entity_declaration_group
| Entity_declaration_list ';' Entity_declaration_group
Entity_declaration_group: Identifier_list ':' Type
Identifier_list: E_IDENTIFIER
| Identifier_list ',' E_IDENTIFIER
Type_mark_opt: -- /* empty */
| ':' Type
Routine: Obsolete_opt Precondition_opt Local_declarations_opt
Routine_body Postcondition_opt Rescue_opt E_END
Routine_body: E_DEFERRED
| E_DO Compound
| E_ONCE Compound
| E_EXTERNAL E_STRING External_name_opt
External_name_opt: -- /* empty */
| E_ALIAS E_STRING
Local_declarations_opt: -- /* empty */
| E_LOCAL Entity_declaration_list
Precondition_opt: -- /* empty */
| E_REQUIRE Assertion
| E_REQUIRE E_ELSE Assertion
| E_REQUIRE E_OTHERWISE Assertion -- New for RI!
Postcondition_opt: -- /* empty */
| E_ENSURE Assertion
| E_ENSURE E_THEN Assertion
| E_ENSURE E_OTHERWISE Assertion -- New for RI!
Invariant_opt: -- /* empty */
| E_INVARIANT Assertion
Assertion: -- /* empty */
| Assertion_clause
| Assertion Assertion_clause
| Assertion ';' Assertion_clause
Assertion_clause: Expression
| E_IDENTIFIER ':' Expression
Adapted_opt: /* empty */
| E_ADAPTED Adapted_list E_END
Adapted_list: Adapted_item
| Adapted_list Adapted_item
| Adapted_list ';' Adapted_item
Adapted_item: '{' Class_type_list '}' Attribute_adaptation
| '{' Class_type_list '}' Routine_adaptation
Class_type_list: Class_type
| Class_type_list ',' Class_type
Attribute_adaptation:
Adapted_type E_IS '(' Expression ')' Adapted_result
Adapted_type: Type
| E_LIKE E_PRECURSOR
```

```
Adapted_result: ':' Expression
-- May contain 'Result'.
Routine_adaptation:
Adapted_formals Adapted_type_mark_opt E_IS
Adapted_actuals Adapted_result_opt
| Adapted_type E_IS Adapted_result
Adapted_formals: '(' Entity_declaration_list ')'
| '(' E_LIKE E_PRECURSOR ')'
Adapted_type_mark_opt: Type_mark_opt
| ':' E_LIKE E_PRECURSOR
Adapted_actuals: '(' Actual_list ')'
| '(' E_PRECURSOR ')'
-- The expressions in Actual_list may contain names of formal arguments
-- of the foster class routine.
Adapted_result_opt: /* empty */
| Adapted_result
Rescue_opt: -- /* empty */
| E_RESCUE Compound
Type: Class_type
| E_EXPANDED Class_type
| E_SEPARATE Class_type
| E_LIKE E_CURRENT
| E_LIKE E_IDENTIFIER
| E_BITTYPE Integer_constant
| E_BITTYPE E_IDENTIFIER
Class_type: E_IDENTIFIER Actual_generics_opt
Actual_generics_opt: -- /* empty */
| '[' Type_list ']'
Type_list: -- /* empty */
| Type
| Type_list ',' Type
Compound: -- /* empty */
| Instruction
| Compound Instruction
Instruction: Creation
| Call
| Assignment
| Conditional
| Multi_branch
| Loop
| Debug
| Check
| E_RETRY
| ';'
Creation: '!' Type '!' Writable Creation_call_opt
| E_BANGBANG Writable Creation_call_opt
Creation_call_opt: -- /* empty */
| '.' E_IDENTIFIER Actuals_opt
Assignment: Writable Assign_op Expression
Assign_op: E_ASSIGN
| E_REVERSE
Conditional: E_IF Expression E_THEN Compound Elseif_list Else_part E_END
Else_part: -- /* empty */
| E_ELSE Compound
```

```
Elseif_list: -- /* empty */
| E_ELSEIF Expression E_THEN Compound
| Elseif_list E_ELSEIF Expression E_THEN Compound
Multi_branch: E_INSPECT Expression When_list Else_part E_END
When_list: -- /* empty */
| E_WHEN Choices E_THEN Compound
| When_list E_WHEN Choices E_THEN Compound
Choices: -- /* empty */
| Choice
| Choices ',' Choice
Choice: Choice_constant
| Choice_constant E_DOTDOT Choice_constant
Choice_constant: E_IDENTIFIER
| Integer_constant
| E_CHARACTER
Loop: E_FROM Compound Invariant_opt Variant_opt E_UNTIL Expression
E_LOOP Compound E_END
Variant_opt: -- /* empty */
| E_VARIANT -- Not standard.
| E_VARIANT Expression
| E_VARIANT E_IDENTIFIER ':' Expression
Debug: E_DEBUG Debug_keys_opt Compound E_END
Debug_keys_opt: -- /* empty */
| '(' Debug_key_list ')'
Debug_key_list: -- /* empty */
| E_STRING
| Debug_key_list ',' E_STRING
Check: E_CHECK Assertion E_END
Call: Call_chain
| E_RESULT '.' Call_chain
| E_CURRENT '.' Call_chain
| '(' Expression ')' '.' Call_chain
| E_PRECURSOR Actuals_opt
| E_PRECURSOR Actuals_opt '.' Call_chain
| '{' E_IDENTIFIER '}' E_PRECURSOR Actuals_opt
| '{' E_IDENTIFIER '}' E_PRECURSOR Actuals_opt '.' Call_chain
Call_chain: E_IDENTIFIER Actuals_opt
| Call_chain '.' E_IDENTIFIER Actuals_opt
Actuals_opt: -- /* empty */
| '(' Actual_list ')'
Actual_list: -- /* empty */
| Actual
| Actual_list ',' Actual
Actual: Expression
| '$' Address_mark
Address_mark: Feature_name
| E_CURRENT
| E_RESULT
Writable: E_IDENTIFIER
| E_RESULT
Expression: Call
| E_RESULT
| E_CURRENT
| E_PRECURSOR
```

```
| '(' Expression ')'
| Boolean_constant
| E_CHARACTER
| E_INTEGER
| E_REAL
| E_STRING
| E_BIT
| E_LARRAY Expression_list E_RARRAY
| '+' Expression %prec E_NOT
| '-' Expression %prec E_NOT
| E_NOT Expression
| E_FREEOP Expression %prec E_NOT
| Expression E_FREEOP Expression
| Expression '+' Expression
| Expression '-' Expression
| Expression '*' Expression
| Expression '/' Expression
| Expression '^' Expression
| Expression E_DIV Expression
| Expression E_MOD Expression
| Expression '=' Expression
| Expression E_NE Expression
| Expression '<' Expression
| Expression '>' Expression
| Expression E_LE Expression
| Expression E_GE Expression
| Expression E_AND Expression
| Expression E_OR Expression
| Expression E_XOR Expression
| Expression E_AND E_THEN Expression %prec E_AND
| Expression E_OR E_ELSE Expression %prec E_OR
| Expression E_IMPLIES Expression
| E_OLD Expression
| E_STRIP '(' Attribute_list ')'
Attribute_list: -- /* empty */
| E_IDENTIFIER
| Attribute_list ',' E_IDENTIFIER
Expression_list: -- /* empty */
| Expression
| Expression_list ',' Expression
Manifest_constant: Boolean_constant
| E_CHARACTER
| Integer_constant
| Real_constant
| E_STRING
| E_BIT
Boolean_constant: E_TRUE
| E_FALSE
Integer_constant: E_INTEGER
| '-' E_INTEGER
| '+' E_INTEGER
Real_constant: E_REAL
| '-' E_REAL
| '+' E_REAL
```

Appendix B

Eiffel Reverse Inheritance Reification in Prolog

B.1 Reification of Class Header

Project

The first fact modelled is the one locating the Eiffel project. We are interested in the name of the project, its location along with the same information about the transformed project or output project:

```
project('ProjectName','Location','OutputProjectName','OutputProjectLocation').
```

This fact in particular has no own identifier, the contained data is global.

Clusters

Clusters are used as modules for grouping classes. In particular in Eiffel there are no visibility rules between clusters like in other programming languages. Clusters are modelled in the following way:

```
cluster(#id,'ClusterName').
```

- *#id* is the the primary key;
- *ClusterName* is the name of the cluster.

Class Declarations

Classes are declared taking into account information about: its location given by the cluster it belongs to, its name and the list of its formal generics. The relation between class declaration facts and formal generic facts is one to many. If the class is not generic then it has no generic parameters and the list is empty. It is necessary to keep them in a list because their order is important for the consistency of the model. A simple reference from formal generic to parent class would not be sufficient.

```
classDecl(#id,#cluster,'ClassName',[#formalGeneric,...]).
```

- *#id* is the identifier of the class;
- *#cluster* is the identifier of the cluster the class belongs to;

- *ClassName* is the name of the class;
- *#formalGeneric* is the identifier of a formal generic parameter.

Obsolete Messages

Obsolete messages are used for the issue of warnings in case the class they belong to, is deprecated.

```
obsoleteMessage(#classDecl or #routine,#manifestConstant)
```

- *#classDecl* is the identifier of the class;
- *#routine* is the identifier of the obsolete routine;
- *#manifestConstant* is the id of a string manifest constant containing the obsolete message of that class.

Index Clauses

The index clauses are used to retain metadata about a class. A clause may have literal identifiers and the literal identifiers may have values. The values are expressed using manifest constant facts which will be modelled later on.

```
indexClause(#id,#classDecl).
```

- a class declaration may have zero or more index clauses;
- *#id* represents the primary key;
- #classDecl points to the class declaration to which the list belongs to.

The index clause may have an attached identifier. These facts are optional, so they are modelled as attribute facts for the index clause facts.

```
indexClauseIdentifierAttribute(#indexClause,'IdentifierName').
```

- an index clause may have an identifier attached;
- *IdentifierName* is the name of the identifier attached.

Index values facts model the values that are attached to index clauses:

```
indexValue(#id,#indexClause,#identifier or #manifestConstant).
```

- one index clause may have attached one or more index values;
- *#id* is the primary key;
- *#indexClause* the identifier of the clause referred;
- *#manifestConstant* is the identifier of the constant.

Class Declaration Attributes

A class declaration may be augmented with several keywords deferred, expanded, separate, foster. These keywords are modelled through the following facts:

```
deferredClass(#classDecl).
expandedClass(#classDecl).
separateClass(#classDecl).
foster(#classDecl).
```

- all of these facts refer to class declarations;
- all clauses are optional for a class declaration.

B.2 Reification of Formal Generics

Formal generic parameters belong to a generic class. They are equipped with their unique identifier, next comes the identifier of the parent class and finally the name of the parameter.

```
formalGeneric(#id,#classDecl,'GenericParameterName').
```

- one classDecl may have zero or more formal generic parameters, it is correlated with the list from the foster class declaration;
- *#id* is the primary key;
- *#classDecl* represents the identifier of the class the generic parameter belongs to;
- *GenericParameterName* is the name of the generic parameter.

A formal generic may be restricted to a certain type, this constraint is modelled by the following fact:

```
constrainedClassType(#formalGeneric,#classType).
```

- a class type may have zero or more generic types;
- *#formalGeneric* represents the identifier of the generic parameter referred by the constraint class type;
- *#classType* is the identifier of the class type used for the constraint.

B.3 Reification of Inheritance

In this section we present the elements which reifies the ordinary inheritance class relationship. To express inheritance we need the identifier of the subclass and the identifier of the superclass type. It is a type and not just a simple class because in case of generic superclasses we have to provide also the actual generic parameters. This information belongs to the type and not to the instantiated class.

```
inheritance(#id,#classDecl,#classType).
```

- *#classDecl* represents the identifier of the current class which is the source of the inheritance relationship;
- *#classType* represents the identifier of the target class type of the relationship.

In inheritance the renaming of feature is possible, this aspect is modeled by the following Prolog fact:

```
rename(#id,#inheritance or #exheritance,#featureDecl,'FeatureNewName').
```

- *#id* is the key of the node;
- *#inheritance* is the identifier of the inheritance link the rename belongs to. Also we can notice that *#exheritance* is a valid choice, meaning that rename can be used in both inheritance and exheritance class relationships.
- *#featureDecl* is the identifier of the feature that is renamed;
- *FeatureNewName* is the new name of the feature.

In the context of inheritance, more specifically on the inheritance branches we can set the inherited features visibility by performing feature export. Between a feature and export client class there is a many to many relationship, meaning that a group of several features may be exported to a group of several classes.

```
exportInherit(#id,#inheritance).
```

- *#id* is the primary key;
- *#inheritance* is the inheritance branch identifier the export belongs to.

The next fact is used for modelling the classes that belong to a certain class group used for export purposes.

```
exportInheritClass(#id,#exportInherit,#classDecl).
```

- *#id* is the primary key;
- #exportExherit is the identifier of the export statement;
- *#classDecl* is the identifier of the class participating in the export statement.

Next we have the model of the features which are exported by the following fact:

```
exportInheritFeature(#id,#exportInherit,#featureDecl).
```

- *#id* is the primary key;
- *#exportExherit* is the identifier of the export statement;
- *#featureDecl* is the identifier of the feature that is exported.

The **all** keyword for an export statement is modelled by the following clause:

```
exportInheritFeatureAll(#id,#exportInherit).
```

- *#id* is the primary key;
- *#exportExherit* is the identifier of the export statement.

This means that all inherited features are exported to a certain group of classes.

When we want to undefine, redefine or select a feature on an inheritance branch we can use the following Prolog facts:

```
undefine(#id,#inheritance or #exheritance,#featureDecl).
redefineInherit(#id,#inheritance,#featureDecl).
selectInherit(#id,#inheritance,#featureDecl).
```

- *#id* is the key of the node;
- *#inheritance* or *#exheritance* is the identifier of the inheritance link that undefine, redefine or select belongs to;
- *#featureDecl* is the identifier of the feature referred from the superclass or exherited class.

B.4 Reification of Creators

The creators of an Eiffel class are modelled using the following fact:

```
creator(#id,#classDecl).
```

- *#id* is the primary key;
- *#classDecl* is the id of the class the creator belongs to.

A class has zero or more creators and that is why they have to be indexed.
The next fact models the features that are used for creation.

```
creatorFeature(#id,#creator,#procedure).
```

- *#id* is the primary key;
- *#creator* is the identifier of the creator the client the feature is accessible from;
- *#procedure* is the identifier of the feature that is declared as creator.

The client class that may use a creator can be set using the following fact:

```
creatorClientClass(#id,#creator,#classDecl).
```

- *#id* is the primary key;
- *#creator* is the identifier of the creator the client class can access;
- *#classDecl* represents the identifier of the client class that can access the creator feature.

B.5 Reification of Features

In this subsection we will present the facts which model the feature related entities. The feature block models a set of features grouped in the code using the **feature** keyword.

```
featureBlock(#id,#classDecl).
```

- *#id* is the primary key;
- *#classDecl* is the class the feature block belongs to.

The feature declaration fact models a feature declaration storing the identifier of the feature block and the feature name:

```
featureDecl(#id,#featureBlock,'FeatureName').
```

- *#id* is the primary key;
- *#featureBlock* refers to the feature block that the feature declaration belongs to.

The next fact models the characteristic of a feature to be an attribute:

```
attribute(#featureDecl).
```

- *#featureDecl* is the identifier of the feature.

The next two facts model the type of the operator when the feature denotes an operator by its name:

```
prefix(#featureDecl).
infix(#featureDecl).
```

The characteristic of a feature as being frozen is modelled using the following fact:

```
frozen(#featureDecl).
```

The following fact models the client classes for a block of features:

```
featureClientClass(#id,#featureBlock,#classDecl).
```

- *#id* is the primary key;
- *#featureBlock* is the identifier of the referred feature block;
- *#classDecl* is the identifier of the client class.

The formal argument list of a feature is modelled using the following fact:

```
formalArguments(#id,#featureDecl,[#formalArgument,...]).
```

- *#id* is the primary key;
- *#featureDecl* is the identifier of the referred feature owning the formal arguments;
- *#formalArgument* is the identifier of the formal argument from the ordered list.

The formal argument itself is modelled using the next fact, by storing the formal arguments fact which is the parent, the name and the type identifier:

```
formalArgument(#id,#formalArguments,'ArgumentName',#type).
```

- *#id* is the primary key;
- *#formalArguments* points to the feature the signature belongs to;
- *ArgumentName* is the name of the formal argument;
- *#type* is the identifier of the argument type.

The type mark of the feature represents its return type. Each feature may have attached one type mark, in case of procedures it is optional.

```
typeMark(#featureDecl,#type).
```

- *#featureDecl* points to the feature who owns the type mark;
- *#type* represents the return type of the feature.

Some features may have constant values attached to them in this case they are constant features in the class. To declare such a feature we will use the following definition:

```
featureManifestConstant(#featureDecl,#manifestConstant).
```

- *#featureDecl* is the identifier of the feature the constant belongs to;
- *#manifestConstant* is the identifier of the manifest constant attached to the feature.

To specify whether a feature is unique we can use the following rule:

```
unique(#featureDecl).
```

A routine can be attached to a feature by using the following fact:

```
routine(#id,#featureDecl).
```

- *#id* is the primary key;
- *#featureDecl* is the identifier of the feature the routine is attached to.
- The routine can be either deferred or once (meaning that it will be executed once in the runtime):

```
deferredFeature(#routine).
once(#routine).
```

- In Eiffel routines can be defined in other programming languages, so they need to have an external name in that language which can be different from the name of the feature in Eiffel. The external name of the feature is modelled as a String manifest constant.

```
external(#id,#routine,#manifestConstant).
```

- *#id* is the primary key;
- *#routine* is the routine that has the external implementation;
- *#manifestConstant* is the identifier of the String containing external name of the feature.
- The external alias of a feature can be modelled using the followings:

```
externalAlias(#external,#manifestConstant).
```

- #external is the id of the external declaration that is referred;
- #manifestConstant is the identifier of the String which is the name of the alias.

Local declarations in a routine may be declared using the following rule:

```
localDecl(#id,#routine,'LocalName',#type).
```

- *#id* is the primary key;
- *#routine* is the routine identifier the local declaration belongs to;
- *LocalName* the name of the local variable;
- *#type* points to the type identifier of the local variable.

There can be noticed that the order in which locals are declared is not important.

The next six clauses refer to all the possible assertions that can be added to a feature. The *require* fact is used for attaching the precondition of a feature within a class which has no superclass, except *ANY*. The *requireElse* is used for declaring the precondition of a feature in a class having one or more superclasses, completing the preconditions from the superclasses. We note that the precondition of the subclass must be weaker or equal than the precondition of the superclass. The *ensure* and *ensureThen* fact attaches a postcondition to a routine in the case of a class not having or having superclasses. The *requireOtherwise* and *ensureOtherwise* are used for declaring the precondition and postcondition of a feature in the foster class.

```
require(#id,#routine,#assertion).
requireElse(#id,#routine,#assertion).
requireOtherwise(#id,#routine,#assertion).
ensure(#id,#routine,#assertion).
ensureThen(#id,#routine,#assertion).
ensureOtherwise(#id,#routine,#assertion).
```

- *#id* is the primary key;
- *#routine* points to the routine the assertion belongs to;
- *#assertion* points to a fact modelling the logical expression to be verified.

The assertion fact is modelled as follows:

```
assertion(#id,#expression).
```

- #id is the primary key;
- *#expression* is the identifier of an expression which must return boolean value.

An assertion may have a tag as an attribute, this is modelled using the following fact:

```
assertionTagAttribute(#assertion,'TagName').
```

- *#assertion* is the identifier of the assertion;
- *TagName* is the name of the assertion tag.

The class type present in an adaptation item is modelled using the following clause:

```
adaptedClassType(#id,#attributeAdaptation or #methodAdaptation,#classType).
```

- *#id* is the primary key;
- *#attributeAdaptation* or *#methodAdaptation* are parent facts;
- *#classType* is the identifier of the class type used in the adaptation process.

The facts is used to adapt attributes or methods to a different signature or behavior.
The attribute adaptation is modelled using the following fact:

```
attributeAdaptation(#id,#featureDecl,#featureDecl,#type or
#identifier (like precursor),#expression,#expression).
```

- *#id* is the primary key;
- *#featureDecl* is the identifier of the parent feature which defines the adaptation;
- *#featureDecl* is the identifier of the feature from the subclass which is adapted;
- *#type* is the identifier of the result type;
- *#expression* is the identifier of the expression passed to precursor;
- *#expression* is the identifier of the expression passed representing the adapted result.

B.6 Reification of Types

The types are represented on two levels: at one level we have a fact modelling an abstract type and on the second level we have the concrete types:

```
type(#id,#classType or #expandedType or #separateType or #likeType or #bitType).
```

- *#id* is the primary key;
- the second argument (*#classType*, *#expandedType*, *#separateType*, *#likeType*, *#bitType*) is the identifier of the concrete type that the current fact refers to.

Class types are modelled by the following fact:

```
classType(#id,#classDecl or #formalGeneric).
```

- *#id* is the primary key;
- *#classDecl* is the identifier of the class name;
- *#formalGeneric* is the identifier of the formal generic parameter.

One can notice that the class type may refer a class declaration or a formal generic.

A class type may have actual generics if the class referred is generic. If the class type refers a formal generic it can never have actual generic types.

```
actualGenericType(#id,#classType,#formalGeneric,#type).
```

- *#id* is the primary key;
- *#classType* is the parent identifier the type the actual generic belongs to;
- *#formalGeneric* is the identifier of the formal generic parameter the actual generic type corresponds to;
- #type is the identifier of the actual generic type.

Expanded and separate types are a special kind of class types which have special properties. The expanded type instances are objects not references, while separate types are used in the concurrent mechanisms of Eiffel.

```
expandedType(#id,#classType).
separateType(#id,#classType).
```

- *#id* is the primary key
- *#classType* is the identifier of the class type that is expanded/separate.

Like types or anchored types are types which refer to the types of other features, formal arguments or local declarations. There are allowed also links to **current** keyword which actually represents a contextual reference to the current object. These types were introduced mainly for helping covariant redeclaration of features:

```
likeType(#id,#identifier(current) or #featureDecl or #formalArgument or #localDecl).
```

- *#id* is the primary key;
- *#identifier* is the identifier of the current keyword;

- *#featureDecl* is the the identifier of the feature referred;
- *#formalArgument* is the identifier of the formal argument referred;
- *#localDecl* is the identifier of the local variable used.

Finally bit types are represented by the following fact. To be noted that they reference either a integer constant either a integer constant feature.

```
bitType(#id,#integerManifestConstant or #featureDecl).
```

- *#id* is the primary key;
- *#integerManifestConstant* refers to an integer constant;
- *#featureDecl* refers to an integer constant attribute.

B.7 Reification of Instructions

A compound instruction is modelled as a ordered set of instructions by the following fact:

```
compound(#id,
 #routine or #conditional or #elseIf or #conditionalElse or
 #when or #multiBranchElse or #loop or #debug,
 [#creation,#assign,#reverse,#call,#conditional,#multiBranch,#loop,
  #debug,#check,#retry]).
```

- *#id* is the primary key;
- the parent entity may be *#routine*, *#conditional*, *#elseIf*, *#when*, *#multiBranchElse*, *#loop*, *#debug*;
- refers in order instruction clauses like *#creation*, *#call*, *#multiBranch*, *#loop*, *#debug*, *#check*, *#retry*.

A rescue compound is modelled like this:

```
rescueCompound(#id,#routine,
 [#creation,#assign,#reverse,#call,#conditional,#multiBranch,#loop,
  #debug,#check,#retry]).
```

- *#id* is the primary key;
- the parent entity may be routine only;
- refers in order instruction clauses like *#creation*, *#call*, *#multiBranch*, *#loop*, *#debug*, *#check*, *#retry*.

The creation instruction is modelled next:

```
creation(#id,#compound,#featureDecl or #localDecl or #identifier (result)).
```

- *#id* is the primary key;
- *#compound* is the identifier of the parent the creation instruction belongs to;
- *#featureDecl* is the identifier of the feature referencing the newly created object;
- *#localDecl* is the identifier of the local referencing the newly created object;

- *#identifier* is the identifier of the result keyword referencing the newly created object.

The type of the creation instruction can be expressed optionally using the next fact:

```
creationType(#id,#creation,#type).
```

- *#id* is the primary key;
- *#creation* is the creation instruction the type is used for;
- *#type* is the type used in the creation.

The creation instruction may imply also a call which is modeled next:

```
creationCall(#id,#creation,#call).
```

- *#id* is the primary key;
- *#creation* is the creation instruction the call is used in;
- *#call* is the identifier of the call needed in the creation process.

The assignment instructions are represented by the following facts:

```
assign(#id,#compound,#featureDecl or #localDecl or #identifier (result),
#expression).
reverse(#id,#compound,#featureDecl or #localDecl or #identifier (result),
#expression).
```

- *#id* is the primary key;
- *#compound* is the identifier of the parent the assignment belongs to;
- *#featureDecl* is the identifier of the assigned feature, which must denote an attribute;
- *#localDecl* is the identifier of the assigned local;
- *#identifier* is the identifier of the result keyword;
- *#expression* is the identifier of the expression assigned.

A conditional instruction has a parent, an expression that must be evaluated and a compound. Optionally there might be present some ordered else-if instructions which deal with other alternatives:

```
conditional(#id,#compound,#expression,#compound,[#elseif,...]).
```

- *#id* is the primary key;
- *#compound* is the identifier of the parent the conditional belongs to;
- *#expression* is the identifier of the evaluated expression;
- *#compound* is the identifier of the compound instruction executed on the then branch;
- *#elseif* is the identifier of the ordered else-if instructions.

The else-if instruction contains a condition, an expression and a compound.

```
elseIf(#id,#conditional,#expression,#compound).
```

- *#id* is the primary key;
- *#conditional* is the instruction the elseif belongs to;
- *#expression* is the identifier of the evaluated expression;
- *#compound* is the identifier of the compound instruction executed on the else-if branch.

The conditional else branch of a conditional instruction is modelled next:

```
conditionalElse(#id,#conditional,#compound).
```

- *#id* is the primary key;
- *#conditional* is the instruction the else belongs to;
- *#compound* is the identifier of the compound instruction executed on the else branch.

The multi branch instruction is similar to the switch instruction of C language and it is modelled as follows:

```
multiBranch(#id,#compound,#expression,[#when,...]).
```

- *#id* is the primary key;
- *# compound* is the identifier of the parent the creation belongs to;
- *#expression* is the identifier of the evaluated expression;
- *#when* is the identifier of the when clause in the ordered list.

The branches are modelled by the following clause:

```
when(#id,#multiBranch,#compound).
```

- *#id* is the primary key;
- *#multiBranch* is the identifier of the branch the when clause belongs to;
- *#compound* is the identifier of the compound instruction to be executed if one of the when choices match.

The choice element is modelled next:

```
choice(#id,#when).
```

- *#id* is the primary key;
- *#when* is the identifier of the parent branch.

A choice may be represented either by constants either by constant ranges:

```
choiceConstant(#id,#choice,#featureDecl or #manifestConstant).
choiceConstantRange(#id,#choice,#featureDecl or #manifestConstant,
 #featureDecl or #manifestConstant).
```

- *#id* is the primary key;
- *#choice* is the identifier of the parent choice;
- third and fourth arguments are identifiers of constant attributes or manifest constants (character or integer).

The multi branch instruction is equipped with an optional else alternative, as we can see next:

```
multiBranchElse(#id,#multiBranch,#compound).
```

- *#id* is the primary key;
- *#multiBranch* is the identifier of the parent multiple branch instruction;
- *#compound* is the identifier of the compound instruction to be executed in case no branch is taken.

The only instruction of Eiffel dealing with iteration is modelled next:

```
loop(#id,#compound,#compound,#expression,#compound).
```

- *#id* is the primary key;
- second argument *#compound* the is the identifier of the parent the creation belongs to;
- third argument *#compound* is the identifier of the compound instruction to be executed for initialization purposes;
- *#expression* is the identifier of the expression to be evaluated at each step;
- #compound is the identifier of the compound instruction to be executed at each step.

The invariant of a looping instruction follows:

```
loopInvariant(#id,#loop,#assertion).
```

- *#id* is the primary key;
- *#loop* is the identifier of the parent loop instruction;
- *#assertion* is the identifier of the assertion representing the loop invariant.

A loop instruction may have variant expressions also:

```
loopVariant(#id,#loop,#expression).
```

- *#id* is the primary key;
- *#loop* is the identifier of the parent loop instruction;
- *#expression* is the identifier of the expression representing the loop variant.

The variant may have an name attached optionally:

```
loopVariantIdentifierAttribute(#loopVariant,'VariantName').
```

- *#loop* is the id of the parent loop variant;
- '*VariantName*' is the name of the variant.

The debug instruction is the next one modelled:

```
debug(#id,#compound,#compound).
```

- *#id* is the primary key;
- the second argument *#compound* is the identifier of the parent the creation belongs to;

- the third argument *#compound* is the id of the compound instruction executed in case of debug situations.

A debug instruction may have attached zero or more keys:

```
debugKey(#id,#debug,#manifestConstant).
```

- *#debug* is the identifier of the parent debug instruction;
- *#manifestConstant* is the identifier of the string value of the debug key.

The check instruction verifies an assertion related to a compound instruction:

```
check(#id,#compound,[#checkAssertion,...]).
```

- *#id* is the primary key;
- *#compound* is the identifier of the parent the creation belongs to;
- *#checkAssertion* refers to an assertion that will be checked at runtime. They are kept in a list in order to preserve their execution order.

The assertion of a check instruction is handled by the following fact:

```
checkAssertion(#id,#check,#assertion).
```

- *#id* is the primary key;
- *#check* is the identifier of the parent check instruction fact;
- *#assertion* is the identifier of the assertion to be checked at runtime.

The retry instruction is used in case of a constraint failure in a compound. As the retry instruction has no parameters it can be unique in the model and it can be referred as many times as necessary.

```
retry(#id,#compound).
```

- *#id* is the primary key;
- *#compound* is the identifier of the parent the creation belongs to.

Calls are a special kind of instructions. They are modelled in a quite sophisticated way because of the composed receivers:

```
call(#id,#expression or #compound, #featureDecl or #localDecl).
```

- *#id* is the primary key;
- the second argument *#expression* or *#compound* is the identifier of the parent the creation belongs to;
- *#featureDecl* is the identifier of the feature declaration that is called, in case of precursor call, it is the id of the feature from the superclass or an identifier representing a local.

The call receivers represent the constructions before the called feature and they are linked one to another in case there are multiple ones:

```
callReceiver(#id,#call or #callReceiver,
 #identifier (result, current, precursor) or
 #expression or #call or #featureDecl or #formalArgument or #localDecl).
```

- #id is the primary key;
- the second argument models the parent of the receiver:
  - *#call* is the identifier of the call the receiver belongs to;
  - *#callReceiver* is the identifier of the parent call receiver of the current receiver;
- the third argument models the receiver entity, and it can be:
  - *#identifier* points to **result**, **current** or **precursor** keywords;
  - these identifiers will be always the last call receivers;
- *#expression* points to an expression identifier;
- *#call* is the identifier of another parent call;
- *#featureDecl* is the identifier of a feature referencing an object;
- *#formalArgument* is the identifier of a formal argument referencing an object;
- *#localDecl* is the identifier of a local declaration referencing an object.

For example the call *current.f.g.h* will be represented like this:

```
call(200,100,id of feature h).
callReceiver(300,200,id of feature g).
callReceiver(301,300,id of feature f).
callReceiver(302,301,id of "current").
```

The precursor receiver of a call may be parameterized by a certain type. This is useful in case of multiple inheritance to select a certain implementation to be executed from one parent.

```
callReceiverPrecursorTypeAttribute(#id,#callReceiver,#type).
```

- *#id* is the primary key;
- *#callReceiver* is the parent receiver which can be only **precursor**;
- *#type* is the identifier of the type used for casting.

The actuals of a call are modelled next. They are not ordered since they refer the formal argument of the feature where the order is kept:

```
actual(#id,
 #creationCall or #call or #actuals,
 #formalArgument,
 #expression or #featureName or #identifier (current, result)).
```

- *#id* is the primary key;
- *#creationCall* or *#call* identifies the call using the current call actual;
- *#actuals* identifies a fact modelling a set of actuals;
- *#formalArgument* the identifier of the corresponding formal argument;
- *#expression* is the identifier of the expression used as actual parameter;
- *#featureName* is the identifier of the feature representing the address mark;
- *#identifier* may be identifier of **current** or **result** representing address marks.

B.8 Reification of Expressions

Expressions are modelled through a fact which may pointing at several facts representing expressions:

```
expression(#id,#call or #identifier (current, result, precursor) or
 #subexpression or #manifestConstant or #manifestArray or
 #unaryOperator or #binaryOperator or #strip).
```

- *#id* is the primary key;
- the second argument is the identifier of another subexpression element:
  - *#call* is the id of a call;
  - *#identifier* which may point to **current** or **result** keywords;
  - *#subexpression* points to another expression;
  - *#manifestConstant* points to a constant;
  - *#manifestArray* is the identifier of an array operator;
  - *#unaryOperator* is the identifier of an unary operator;
  - *#binaryOperator* is the identifier of a binary operator;
  - *#strip* is the identifier of a strip expression.

Subexpressions model an expression which is enclosed between parenthesis:

```
subexpression(#id,#expression).
```

- *#id* is the primary key;
- the second argument is the identifier of the expression element.

As component of an expression is the unary operator. It may be: +, -, not, free operator, old.

```
unaryOperator(#id,'OperatorSymbol',#expression).
```

- *#id* is the primary key;
- *'OperatorSymbol'* is the symbol of the operator;
- *#expression* is the identifier of the expression representing the operators argument.

Another expression component is the binary operator (free operator symbol, +, -, *, /, ^, div, mod, =, /=, <, >, <=, >=, and, or, xor, and then, or else, implies):

```
binaryOperator(#id,#expression,'OperatorSymbol',#expression).
```

- *#id* is the primary key;
- #expression is the identifier of the expression representing the operators left argument;
- *'OperatorSymbol'* is the symbol of the operator;
- *#expression* is the identifier of the expression representing the operators right argument.

The manifest array expression is modelled by the next fact:

```
manifestArray(#id,[#manifestArrayExpression,...]).
```

- *#id* is the primary key;

- the third argument is a table for maintaining the order of the expression identifiers.

The manifest array expression is modelled by the next fact:

```
manifestArrayExpression(#id,#manifestArray,#expression).
```

- *#id* is the primary key;
- *#arrayOperator* is the identifier of the array operator referred;
- *#expression* is the identifier of the expression used by the operator.

A strip expression returns an array of all attributes of an object. It is modelled by the following fact:

```
strip(#id).
```

- *#id* is the primary key of the strip expression.

A strip attribute is modelled next:

```
stripAttribute(#id,#strip,#featureDecl).
```

- *#id* is the primary key;
- *#strip* is the parent strip instruction the attribute belongs to;
- *#featureDecl* is the identifier of the referred attribute.

Identifiers are modelled by the following facts:

```
identifier(#id,'IdentifierName').
```

- *#id* is the primary key;
- *'IdentifierName'* is the name of the identifier.

Manifest constants of type boolean, character, integer, real, string, bit are modelled next:

```
manifestConstant(#id,'ManifestConstantValue', 'boolean' or 'character' or
'integer' or 'real' or 'string' or 'bit').
```

- *#id* is the primary key;
- *'ConstantValue'* is the value of the manifest constant;
- the third argument models the type of the constant.

Invariants are boolean expressions which must hold in the context of a class:

```
invariant(#id,#classDecl,#assertion).
```

- *#id* is the key of the node;
- *#classDecl* is the identifier of the class that the invariant refers to;
- *#assertion* is the identifier of the invariant assertion.

B.9 Reification of Exheritance

The exheritance class relationship is expressed between the foster class and the exherited classes as types and instantiated in case of generic classes:

```
exheritance(#id,#classDecl,#classType).
```

- *#id* is the primary key;
- *#classDecl* is the identifier of the current, source class;
- *#classType* is the identifier of the exherited class type.

The adapt and moveup clauses are linked to the exheritance class relationship and to a feature:

```
adapt(#id,#exheritance,#featureDecl).
moveup(#id,#exheritance,#featureDecl).
```

- *#id* is the primary key;
- *#exheritance* is the identifier of the exheritance branch;
- *#featureDecl* is the identifier of the feature to be adapted.

The exheritance selection mechanism is presented next:

```
selectExherit(#id,#exheritance,#descendantQualification).
```

- *#id* is the primary key;
- *#exheritance* is the identifier of the exheritance branch;
- *#descendentQualification* is the identifier of the class chain that the feature is selected in.

The feature which may be selected in the exheritance selection mechanism is next. One exheritance clause may have multiple features to be selected.

```
selectExheritFeature(#id,#selectExherit,#featureDecl).
```

- *#id* is the primary key;
- *#selectExherit* is the identifier of the parent fact;
- *#featureDecl* is the identifier of the feature to be selected.

Descendant class chains can be constructed using the following clause:

```
descendantQualification(#id,#descendantQualification,#classDecl).
```

- *#id* is the primary key;
- *#descendantQualification* is the element before the current one, the first entity will have no predecessor, so this value will be zero;
- *#classDecl* is the identifier of the class in the chain.

The export declarations in the context of reverse inheritance are attached to the class and not to the inheritance clause, this is why they are modelled separately. Between a feature and export client class there is a many to many relationship (multiple features can be exported to multiple classes):

```
exportExherit(#id,#classDecl).
```

- models an export statement;
- *#id* is the primary key;
- *#classDecl* is the foster class identifier the export belongs to.

The classes which may be linked to an export statement are modelled next:

```
exportExheritClass(#id,#exportExherit,#classDecl).
```

- *#id* is the primary key;
- *#exportExherit* is the identifier of the export statement;
- *#classDecl* is the identifier of the class participating in the export statement.

Exported features are attached to the export statement by the following fact:

```
exportExheritFeature(#id,#exportExherit,#featureDecl).
```

- *#id* is the primary key;
- *#exportExherit* is the identifier of the export statement;
- *#featureDecl* is the identifier of the feature that is exported.

If one desires to export all features of a class the following clause must be attached to the export clause:

```
exportExheritFeatureAll(#id,#exportExherit).
```

- #id is the primary key;
- #exportExherit is the identifier of the export statement.

The redefinition in the context of reverse inheritance is global, it does not belong to an exheritance branch like in ordinary inheritance:

```
redefineExherit(#id,#classDecl,#featureDecl).
```

- *#id* is the primary key;
- *#classDecl* is the identifier of the class containing the feature;
- *#featureDecl* is the identifier of the redefined feature.

The selection mechanism of exheritance allows to select or deny a set of specific features through the following facts:

```
onlyFeature(#id,#classDecl,#featureDecl).
exceptFeature(#id,#classDecl,#featureDecl).
```

- *#id* is the primary key;
- *#classDecl* is the identifier of the class hosting the feature selection clauses;
- *#featureDecl* is the feature in the exherited class.

The selection mechanism can be set to select all exheritable features or no features at all, in order to create a new type:

```
allFeature(#classDecl).
nothingFeature(#classDecl).
```

- *#classDecl* is the identifier of the class hosting the feature selection clause.

B.10 Reification of Feature Adaptation

A rather new concept, but very necessary in the process of feature exheritance is the concept of adaptation, which changes dramatically the structure of an ordinary feature. The main change consists, as we know, in having a feature adaptation clause for each exherited class when necessary.

```
adaptedClassType(#id,#attributeAdaptation or #methodAdaptation,#classType).
```

- models the class type present in an adaptation item;
- multiple adaptedClassType facts refer to one #attributeAdaptation or #methodAdaptation fact;
- *#id* is the primary key;
- *#attributeAdaptation* or *#methodAdaptation* are parent facts;
- *#classType* is the identifier of the class type used in the adaptation process;

```
attributeAdaptation(#id,#featureDecl,#featureDecl,#type or
 #identifier (like precursor),#expression,#expression).
```

- *#id* is the primary key;
- *#featureDecl* is the identifier of the parent feature which defines the adaptation;
- *#featureDecl* is the identifier of the feature from the subclass which is adapted;
- *#type* is the identifier of the result type;
- *#expression* is the identifier of the expression passed to precursor;
- *#expression* is the identifier of the expression passed representing the adapted result;

```
routineAdaptation(#id,#featureDecl,#featureDecl).
```

- *#id* is the primary key
- *#featureDecl* is the identifier of the feature which defines the adaptation;
- *#featureDecl* is the identifier of the feature from the subclass which is adapted.

The adapted formals are modelled next:

```
adaptedFormals(#routineAdaptation,#formalArguments or #identifier (like precursor)).
```

- *#routineAdaptation* is the identifier of the parent fact;
- *#formalArguments* is the identifier of the formal arguments fact;
- *#identifier* is the identifier of **like precursor** keyword construction.

The adapted type mark is modelled next:

```
adaptedTypeMark(#routineAdaptation,#type or #identifier (like precursor)).
```

- *#routineAdaptation* is the identifier of the parent fact;
- *#type* is the identifier of the return type of the method;

- *#identifier* is the identifier of the **like precursor** keyword construction;

The next fact models the adapted actuals of an adapted feature, they can be regular actuals or the **precursor** keyword.

```
adaptedActuals(#routineAdaptation,#actuals or #identifer (precursor)).
```

- *#routineAdaptation* is the identifier of the parent fact;
- *#actuals* is the identifier of the set of actuals;
- *#identifier* (**precursor**) is the identifier of the keyword used when a simple call has to be made.

The fact refers to the adapted actuals:

```
actuals(#id,#adaptedActuals).
```

- *#id* is the global identifier for the actuals;
- *#adaptedActuals* is the parent fact.

The adapted result consists in an expression which may contain the **result** keyword and is modelled by the following fact:

```
adaptedResult(#routineAdaptation,#expression).
```

- *#routineAdaptation* is the identifier of the parent fact;
- *#expression* is the identifier of the expression fact.