The Model of the Generic Mechanism for the Extension of Object-Oriented Programming Languages Reverse Inheritance for Eiffel

PhD Research Report #2

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

PhD Supervisor: prof. dr. ing. Ioan Jurca
PhD Co-supervisor: prof. Philippe Lahire (University of Nice, France)

Faculty of Automation and Computer Science

University Politehnica of Timişoara

January 28, 2008

Contents

Abstract

Reverse inheritance class relationship is a different kind of inheritance where the subclasses exist first and the superclass is created afterwards. When designing an object-oriented system, it is more natural to create first the specialized classes and then to notice commonalities and to factor them into superclasses using reverse inheritance. Class libraries do not always fit into the context of the developed software, the adaptation mechanisms of reverse inheritance can help in this direction. Such a class relationship was never fully defined in the literature nor implemented in a programming language. Because of concepts like adaptations, lack of overloading, equivalency between attributes and methods, the most appropriate programming language to integrate reverse inheritance is Eiffel.

Acknowledgements

This research report is part of the PhD programme developed in the collaboration between University Politehnica of Timisoara, Romania and University of Nice from Sophia-Antipolis, France, where I had two internships of three month each, one in 2003 and the other in 2005.

I would like to thank professor Ioan Jurca for his dedicated efforts in supervising my PhD research activity, sustaining my research projects and making the reviews for this report.

I would like to thank professor Philippe Lahire and M.C. Pierre Crescenzo for their financial and intellectual resources invested in the research around the subject of the thesis. I would also like to express my gratitude to the french team who was motivating me thus making me advancing on the subject.

I would also like to thank professor Markku Sakkinen from University of Jyvaskyla, Finland, for showing points of interest for the developed problematics during one of my PhD internships, especially for this research report.

Also I would like to thank my colleagues lecturer Dan Pescaru and teaching assistant Emanuel Ţundrea for the realistic feedback and for the technical coaching. I want also to express my gratitude to Ms. Monica Ruzsilla and Ms. Smaranda-Claudia Chirila for their implication in the development of some prototypes based on ideas deriving from this research.

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

Chapter 1

Introduction to the Approach

In this report 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 details 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 [CCL$^+$05b]. We keep the same restrictions of not affecting the behavior of exherited classes when exheriting from them. The choice of Eiffel was taken because its philosophy is closer to the concepts required by reverse inheritance e.g. the renaming technique, the presence of conforming and non-conforming inheritance, the lack of overloading for features. There are some attempts to implement reverse inheritance for Java programming language [CPT05, CRC$^+$06a] and even to ajust its semantics in order to solve a restricted set of problems [CRC06b], but the resulted semantics is a great deviation from the philosophy of Java.

A secondary objective of the report 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, UML04].

Next, we will present some motivating examples to show why both reverse inheritance and like-type class relationships are needed. The examples are not analyzed thoroughly, only a flavour of what these class relationships are capable to do, is emphasized next.

1.1 Motivative Reverse Inheritance Use Cases

Reverse inheritance has several capabilities in the direction of adaptation and evolution of aplications [CCL05a, CCL07a, CCL07b].

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.

1.1.2 Capturing Common Functionalities

In [OJ93] is mentioned that it is more economical to refactor an existing application by extracting the framework that otherwise should be created from scratch. In some applications classes belonging to different contexts need to be used together. Sometimes they have even common

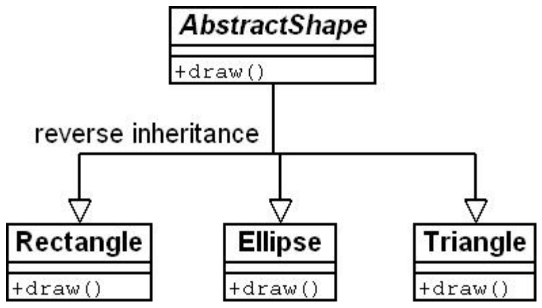

Figure 1.1: Capturing Common Functionalities

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 from figure 1.2 we present a class hierarchy which at design time had only two classes *Shape* and *Rectangle* in a subtype relationship. Later was decided that a new class *Parallelogram* had to be added to the hierarchy. 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.

1.1.4 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

[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.

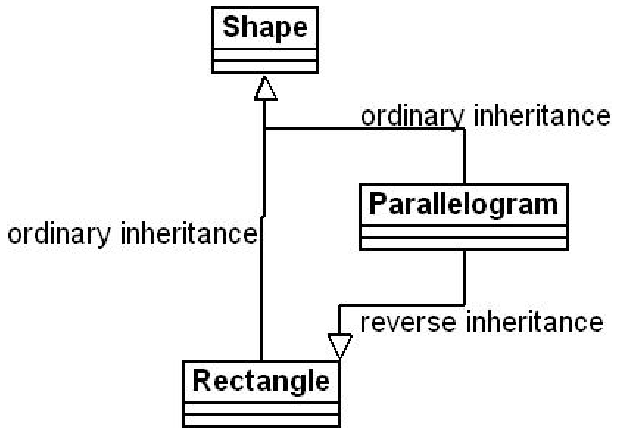

Figure 1.2: Inserting a Class Into an Existing Hierarchy

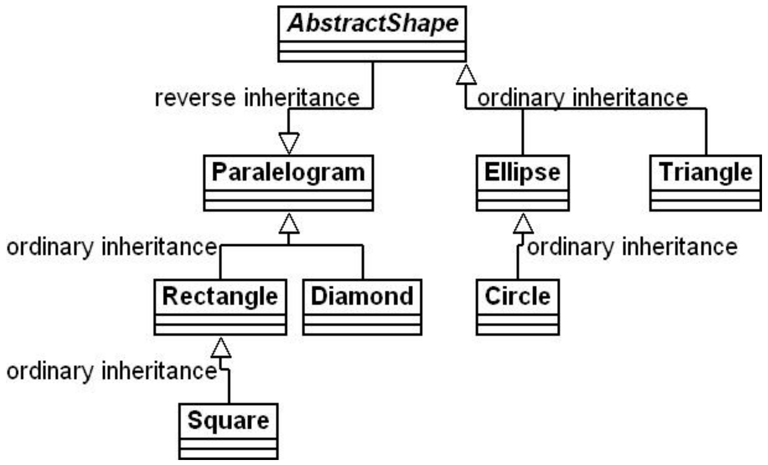

Figure 1.3: Extending a Class Hierarchy

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 precess, an abstract superclass named *AbstractShape* is created using reverse inheritance. Then 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 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 sample 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

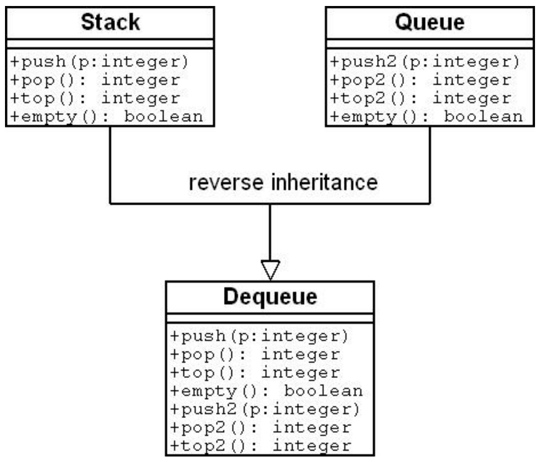

Figure 1.4: Reusing Partial Behavior of a Class

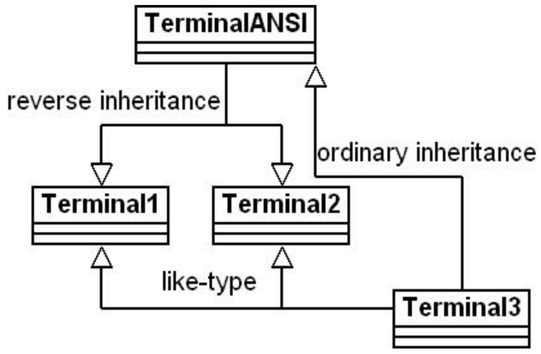

Figure 1.5: Creating a New Type

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 4.6) 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, and 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

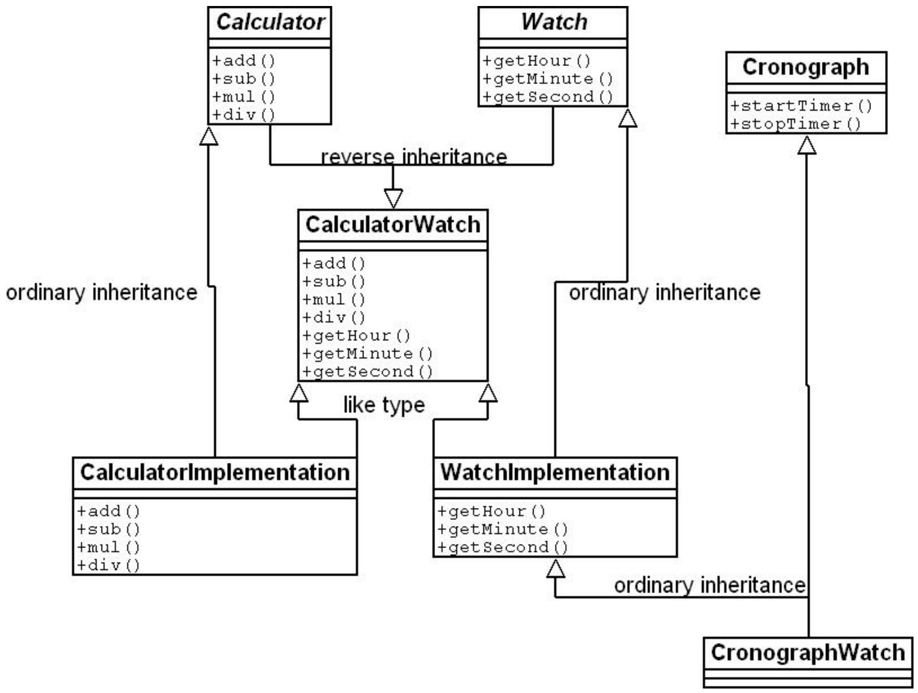

Figure 1.6: Decomposing and Recomposing Classes

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 can not be added new functionalities. It is more natural to extract the behavior using the like-type class relationship into classes *CalculatorImplementation* 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 Outline of the Approach

The report has the following outline. Chapter 2 presents the basic elements of our approach of reverse inheritance. There are discussed aspects like cardinality, feature factorization, type conformance. In chapter 3 are presented main mechanisms through which feature adaptations can be performed. In chapter 4 are discussed dynamic binding aspects in the context of reverse inheritance and constraints which must be imposed on foster classes. In chapter 4.6 the semantics of like type class relationship is discussed. In the final chapter 5 the approach is evaluated and conclusions are drawn.

Chapter 2

Creating a Class by Reverse Inheritance

2.1 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. 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] do 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 2.4 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 defaults in the semantics of reverse inheritance 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 of reverse inheritance may be called **factored**, **reverse inherited** or **exherited**.

In the next subsections we set the main principles which stand for the reverse inheritance class relationship and we propose a notation.

2.1.1 Definitions

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

[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.

[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.

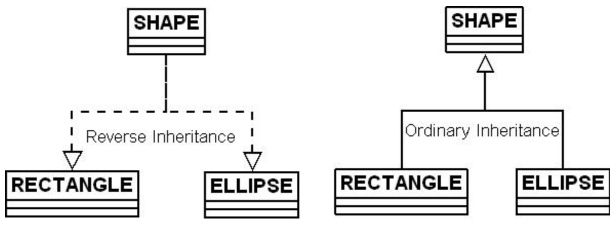

Figure 2.1: Reverse Inheritance

Example 1 Reverse Inheritance Example

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

conforming cases.

From this 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.

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 with ordinary inheritance. Some rules will give an intuitive definition of reverse inheritance:

Rule Equivalence with Ordinary Inheritance. Declaring a reverse inheritance relationship from a class A to a class B is equivalent to declare an ordinary inheritance relationship from class B to class A.

Rule Invariant Behaviour. Introducing an ancestor C to one or several classes $C_1, \ldots, C_n$ using reverse inheritance does not modify the behavior of $C_1, \ldots, C_n$.

2.1.2 Notations

In figure 2.1 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 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 1 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 that aspect. This code is semantically equivalent with the context in which class *SHAPE* is created first, and then subclasses *RECTANGLE* and *ELLIPSE* (see the code which is described in the next example 2).

Example 2 Reverse Inheritance Equivalent Sample

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

Reverse inheritance[4] allows the programmer to create a superclass out of one or several subclasses and to select the **common features** 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 are illustrated in the rules of example 3:

In the grammar from example 3 we provide the definition of the foster class which is the source class 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 who have either the same signature or those for which the signature may be adapted using the clauses presented in chapter 3. In the selection clause, the keyword **exherit** is then used in different combinations to select the exherited features:

- using the **all** keyword denoting 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 denoting 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 there are excluded non-eligible features 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 $\mathrm{C}_1 .. \mathrm{C}_\mathrm{n}$ is exheritable if one of the following assumption is satisfied:

- The signatures of *f* in all C_i ($1 \leq \mathrm{i} \leq \mathrm{n}$) are identical;
- The signatures of *f* in all C_i ($1 \leq \mathrm{i} \leq \mathrm{n}$) may be adapted in order to conform:
  - either to the signature of *f* in one C_j ($1 \leq \mathrm{j} \leq \mathrm{n}$) ;

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

Example 3 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]
    [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]
    [redefine feature_list]
```

  – 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 **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.
- 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 clauses **redefine** and **undefine**. These issues are developed in chapter 3.
- The clause **moveup** 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 clause **select** 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 4.2.

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 **redefine** clause has the same semantics as in ordinary inheritance of Eiffel and it is used for signature redefinitions or for implementation redefinitions.

The compatible combinations of these clauses will be studied in a later chapter 4, 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 3.

2.2 Single/Multiple Reverse Inheritance

Eiffel supports multiple inheritance (so that it can be single and multiple); 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 2.2 and 2.3 we will analyze the case of the *Dequeue* example taken from [Ped89].

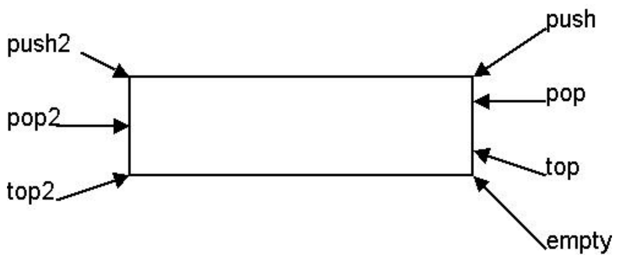

Figure 2.2: Dequeue Sample

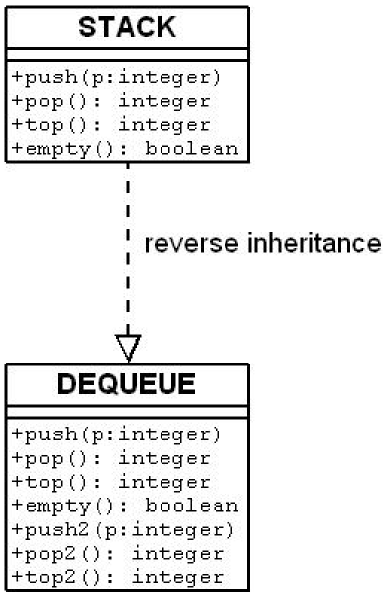

Figure 2.3: Dequeue Class Diagram

2.2.1 Single Reverse Inheritance

The sample proposed in figure 2.2, 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 2.3, 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 sample 4.

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 this is needed).

Example 4 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
```

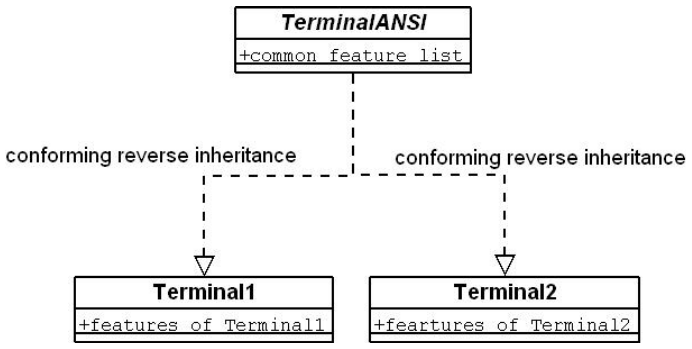

Figure 2.4: Multiple Reverse Inheritance

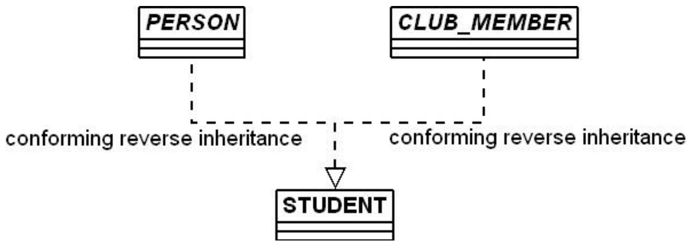
PERSON
CLUB_MEMBER
conforming reverse inheritance
conforming reverse inheritance
STUDENT

Figure 2.5: Two Independent Reverse Inheritance Relationships

2.2.2 Multiple Reverse Inheritance

Multiple reverse inheritance is a special case of reverse inheritance where are involved multiple source classes. 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 2.4 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.

2.2.3 Several Independent Reverse Inheritance Relationships

The case in which several superclasses exherit from a class, like in figure 2.5 are not multiple reverse inheritance, it is just the fact that there are several reverse inheritance relationships which happen to have the same target class. In figure 2.5, *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 point of views 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. One ambiguity related to feature exheritance may arise if some

[5]This will be studied in details in chapters 3 and 4.

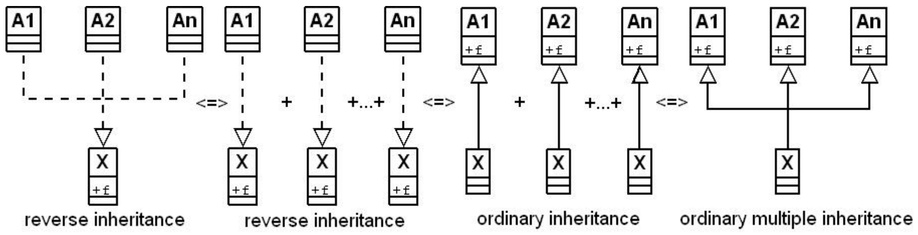

Figure 2.6: Several Independent Reverse Inheritance Relationships

features from a subclass are multiply exherited into two or several superclasses. As we specified 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 prove 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 2.6.

In the general case of figure 2.6 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.

2.3 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 [CCL+04c, CCL04a, CCL04b]. 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 3.

2.3.1 Implicit Rules Regarding Feature Exheritance

In this subsection we will learn 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.

2.3.1.1 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 it will be a concrete feature in the foster class.

In example 5, 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 6, we have the same

Example 5 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
```

class configuration as in example 5, but the feature a is redefined in the foster class. Redefinition in Eiffel serves for 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 former purpose.

In example 7 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.

2.3.1.2 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 8. 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 details in chapter 4.

Example 6 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 7 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 8 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 9. 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 is left only one exheritance branch 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.

2.3.2 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.

2.3.2.1 Implicit All Common Feature Selection

The **exherit ... all** keyword combination will denote that there are selected for factorization all exheritable features. In example 10 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 9 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 10 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 11 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
```

2.3.2.2 Explicit Common Feature Selection

We state in example 11, 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 there are a lot of candidate features to 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.

2.3.2.3 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 12 we consider that common features are automatically exherited, it would be the case for the attribute *area* from both *RECTANGLE* and *ELLIPSE* classes. Since the attribute *color* 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 12 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
```

2.3.2.4 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 13 the use of **nothing** keyword is more intuitive, for creating a new type *SHAPE*.

2.3.3 Influence of the Nature of Common Features

In this subsection are 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.

2.3.3.1 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 of by redefinition) features in the superclass. As it is mentioned in [Sak02] only some type and name conflicts may occur. Types will be discussed in later sections 3.3.1 and 3.3.2, while name conflicts are analyzed in section 3.1.3. A basic situation of exheriting attributes having the same signatures is presented in example 14.

The two original existing classes *RECTANGLE* and *ELLIPSE* have both one feature *perimeter* with the same name. The case where features have different names is discussed in section 3.1.3. About constant attributes, they can not 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 can not 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 and exists one having a common supertype for all the types in subclasses, that supertype will be used as type for the implicitly exherited deferred feature in the foster class, in this case some redefinition statements must be used.

Example 13 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 14 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 15 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 there is no common supertype for the exherited attributes in the exherited classes then redefinitions must be applied using a new common supertype.

2.3.3.2 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. Conform 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 15. 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.

2.3.3.3 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 whatever they are deferred or effective. 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 16 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 exist initially. 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 even 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 one of the following chapters 3.

Example 16 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, is illustrated the polymorphic behavior of all *SHAPE* instances, 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.

2.3.3.4 Factoring Implementation

Another case is where both common signature and the implementation are factored as well [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 17 will illustrate such an implementation exheritance situation.

Example 17 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
    halfperimeter
 end
 ELLIPSE
 all
 feature
  -- perimeter:REAL; (is implicit)
  -- halfperimeter:REAL is
  --  (RECTANGLE implementation)
  -- end
end
```

In example 17 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 can not 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 can be 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.

Example 18 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
...
```

2.4 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]. A class implicitly is considered to be a reference type.

Type conformance definition can be found in [Ped89] where 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. It is referred also as instances of type T conforming to S.

As in ordinary inheritance there is 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.

2.4.1 Conforming Reverse Inheritance

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

Example 18 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.

Example 19 Non-conforming Reverse Inheritance

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

Example 20 Non-conforming Reverse Inheritance (2)

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

2.4.2 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 its type to be a supertype[8] of the subclasses. We have to state regarding the semantics of this class relationship that all rules apply from the conforming reverse inheritance except those related to type conformance. Reverse inheritance conforming and non-conforming 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:

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 Conformance. 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 20 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.

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

Example 21 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;
...
```

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

2.4.3 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 21 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 22 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
```

2.4.4 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[9]. The signature of a feature consists of name, parameter number and type, return type. In the redeclaration of a feature, the original arguments and results types can be replaced in the subclass with conformant 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 22 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.

2.4.5 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

[9]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 23 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 objects.

In example 23 any instance of class C will be an object 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.

2.5 Type Exheritance

In this chapter 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 its 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.

2.5.1 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 can not have. The actual generic types can be any type of Eiffel type system. Each one of these situations is analysed in the next subsections.

2.5.1.1 Exheriting Class Types Referring Class Declarations

In example 24 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 Class Declarations 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 24 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 25 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 26 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
```

2.5.1.2 Exheriting Class Types Referring Formal Generics

In example 25 we deal with class types referring formal generics. In such cases because of genericity problems discussed in section 3.4 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.

2.5.1.3 Exheriting Class Types Referring Class Declarations and Having Actual Generics

In example 26 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 that such a type to be exherited 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.

2.5.2 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, but still some type rules will be issued in the context of reverse inheritance.

In example 27 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 27 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.

2.5.3 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.

2.5.3.1 Exheriting Features Anchored to "current"

In the example 28 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 to 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.

2.5.3.2 Exheriting Features Anchored to Other Features

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

In example 28, the anchors are attributes. We can set the following rule:

Example 28 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 29 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 30 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
 C
 all
 redefine f
 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).

2.5.3.3 Exheriting Features Having Arguments Anchored to Other Arguments

In this case we consider not just method arguments but also return types. In example 30, 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 3.3.1. 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.

2.5.3.4 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 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

Example 31 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
 C
 all
 redefine att
 feature
  att: T
  -- f(p: like att): like att is deferred end -- is implicit
end
```

by rules. One possibility is to use the anchored feature target type, which might be a recursive process.

In example 31 it is presented 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 exherited also.

2.5.4 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 32 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;
- either 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 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

Example 32 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
```

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 Bit types in order to be exherited, they must refer the same value and they will be exherited in the foster class as referring an integer manifest constant.

2.5.5 Exheriting Various Types

In example 33 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 separate keyword to the class type having an actual generic.

2.6 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 were specified in the superclass, it would be inherited by all the subclasses changing thus the semantics of the original ones.

Example 33 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
```

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, so that the content of the superclass must be restricted. It can contain only a subset of the exherited features preserving their signatures (but 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 bases for determining when a foster class can have superclasses:

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

2.7 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]. In the context of reverse inheritance the redefinition clause is placed after all the exheritance branches and the feature selection section. 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 can not 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 can not 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 3 are presented details on these issues.

To have a first idea of possible combination 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 clause, 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 can not be used in combination with the two. 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 an eventual **renaming** clause the new name of the feature has to be used in the eventual 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 redefinition or other 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 can not be undefined. The redefinition of a moved feature is necessary if its signature is changed in the foster class.

2.8 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 relationship. 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.

There were presented the implicit and natural semantics of factoring features. 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 is the same rule, only the signature is exherited, being deferred too 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. There are analyzed cases where attributes and methods with the same signature are 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 in this point is the same as the one used for ordinary inheritance. Another supposition for keeping the semantics of the language intact is to allow the behavior in the foster class influence in any case the behavior in the subclasses.

Type exheritance refers to the rules showing how types can be exherited. Types having 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 from class types have to be applied along with the appearance of **expanded** respectively **separate** keyword. 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 can not 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 rises 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 made involves only two exherited classes, but the same reasoning can be applied when working with multiple ones.

Chapter 3

Adaptation of Exherited Features

In [LHQ94] is presented a set of adaptations which are valid in the definition of the reverse inheritance semantics. We still 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 some features of classes which are exherited in order to conform to a common signature. These feature adaptations are performed before their factorization, in order that a feature with different signatures to satisfy the constraints for being exherited.

In this chapter we will see 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.

3.1 Adaptation for Ordinary Inheritance Applied to Reverse Inheritance

3.1.1 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. About the changes that can be done there are some limitations.

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 3.5.

Implementation redefinition will take into account the rules in subsections 17 and 2.6, 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 34. 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 following conditions:

Example 34 Feature Redefinition

```
class T
end
class T1 inherit T
end
class A
feature
  f(x: T1) is do -- implementation of class A end
end
class B
feature
  f(x:T) is do -- implementation of class B end
end
class C exherit
 A
 B
 all
 redefine f
feature
  f(x: T) is do -- implementation of foster class C end
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 3.5;
- implementation adaptation will conform to the rules related to implementation exheritance (section 2.3.3.4) and to the rules related to behavior in the newly created class (section 2.6);
- 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.

3.1.2 Feature Undefinition

In ordinary inheritance if an affective feature is undefined then it becomes deferred in the subclass (please note 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 2.3.1.

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

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

Example 35 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
```

3.1.3 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 35 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.

3.1.4 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 3.1.

3.2 Special Signature and Value Adaptations

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

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 3)
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 details is section 4.2
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 4)

Table 3.1: Semantics of Inheritance and Exheritance Clauses

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 if one adaptation had been defined for one of its ancestors, if any. The keyword **adapted** 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 36.

3.2.1 Scale Adaptation

The idea of providing scale adaptation can be found in the works of [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 37 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 shape, but using different scales for that. The method in class *RECTANGLE* returns a value in

Example 36 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
```

Example 37 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
```

```
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
```

Figure 3.1: Parameter Position Adaptation

cm^2 while the corresponding one in class *ELLIPSE* returns a value in m^2. Following the same homogeneity principle announced in section 34 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 that finally when it is returned must be adapted.

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 38. In order to exherit the attribute *surface* 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 achieve 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 adaptations 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 adaptations 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.

3.2.2 Parameter Order Adaptation

The issue of parameter order adaptation is discussed in [LHQ94]. They even proposed a syntax extension to solve this problem[1]:

[1] 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.

Example 38 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 39 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)
```

The syntax extension which is proposed rely on the clause *adapt* like for scale adaptation (see section 3.2.1). 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 perform a call to the method *zoom* with the parameters in the reverse order.

Example 39 illustrates a situation where the adaptation of the parameter order is useful. When the method scale is called then everything work as if it was the *zoom* method which is called relying on both adaptation mechanism and dynamic binding semantics[2].

3.2.3 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:

- 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 his 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 to obtain a particular behavior, some parameters can be duplicated. It is intended that the behavior of the superclass to 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 yield a result which will be used as a parameter.

In example 40 all the cases are experimented and a syntax is proposed to describe them.

In example 40 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

[2]Issues dealing with the dynamic binding will be addressed with much more details in chapter 4.

Example 40 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
```

at the head of method declaration (after possible preconditions), before the keyword **deferred** or respectively **do**.

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, freezed, replicated or computed. The syntax used for this adaptation is the one presented at the beginning of this section (3.2).

3.3 Classic Signature Adaptations

Rule Classic Method Signature Adaptation. The redefine and undefine clauses allow to specify the adaptation of parameter type and the type of feature (attribute type or return function type). These adaptations rely on the conformance rules based on the polymorphism, exactly as the clause with the same name applied to ordinary inheritance in Eiffel. A feature is considered to be redefined in a subclass in the context of ordinary inheritance if signature, specification or implementation are changed. We keep the same rule in the context of reverse inheritance: signatures must be covariant, specification should be implied by all the specifications of subclasses for a certain feature.

3.3.1 Parameter Type Adaptation

Parameters are used to exchange information between methods and procedures. Sometimes because the subclasses were developed in different contexts some parameter type adaptations are necessary for methods. This case deals with all type adaptations which can be performed when undefining or redefining a feature. For that it is necessary to use the clauses **redefine** or **undefine** depending on the relevance of exheriting the concrete version or only the signature of the feature.

Redefining the Signature and the Behavior

In example 41 the feature f declared in class A is of type $SQUARE$ whereas it is of type $CIRCLE$ in class B. Since in class C the feature f which is exherited from both subclasses, needs to be of type $FIGURE$ (a supertype of $SQUARE$ and $CIRCLE$). We assume that the body has to be redefined along with the signature redefinition. In this case we must use the clause redefine in the foster class and the definition of a new body is needed. A full discussion about the possible combinations of exheritance clauses had been presented in section 2.7 and this adaptation conforms to it.

Let us discuss about whether class $FIGURE$ exists already or not. If it exists then there is nothing to do, otherwise two alternatives are possibles : i) to create it *a posteriori* using a reverse inheritance relationships to the parameter type classes, ii) to use a common existing ancestor of the subclasses (there is at least one: the special class ANY which is the superclass of any defined classes in Eiffel).

Redefining the Signature Only

Example 42 relies on the classes declared in example 41 but in this situation, it is only necessary to modify the signature and not the body, so therefor we use the implicit clause **undefine** instead of **redefine**. The feature f is the exherited in C with the parameter p of type $FIGURE$ from subclasses A and B where the parameter p of f is respectively of type $SQUARE$ and $CIRCLE$ (the class $FIGURE$ is a superclass of $SQUARE$ and $CIRCLE$). In this situation, the declaration in C of f as a deferred feature will be sufficient[3].

[3]All what had been said in previous paragraph about the fact that class *FIGURE* exists or not and, about the possible combination of exheritance clauses, remains true.

Example 41 Parameter Type Adaptation

```
class FIGURE inherit
 SQUARE
 CIRCLE
end
class A
feature
 f(p: SQUARE) is do ... end
end
class B
feature
 f(p: CIRCLE) is do ... end
end
foster class C exherit
 A
 B
 all
 redefine f
feature
 f(p: FIGURE) is do ... end
end
```

Example 42 Class Type Parameter Adaptation (1)

```
deferred foster class C exherit
 A
 B
 all
 redefine f
feature
 f(p: FIGURE) is deferred end
end
```

Example 43 Class Type Parameter Adaptation (2)

```
class FIGURE
 convert
  to_square: SQUARE is do ... end
  to_circle: CIRCLE is do ... end
feature
end
class SQUARE
end
class CIRCLE
end
class A
feature
 f(p: SQUARE) is do ... end
end
class B
feature
 f(p: CIRCLE) is do ... end
end
deferred foster class C exherit
 A
 B
 all
 redefine f
feature
 f(p: FIGURE) is deferred end
end
```

Considering the Case Where There is No Type Conformance

Let us assume that *FIGURE* is not a common ancestor of *SQUARE* and *CIRCLE* (and that we do not want to use the class *ANY*). In this context we may use the conversion mechanism available in Eiffel: when a certain type object is needed in a context and some other type object is passed, conversion methods have to be provided. Please note that in figure 43 we redefine only the signature (use of the clause **undefine**) but it would work exactly in the same conditions if the behavior should be redefined also (use of the clause **redefine**).

In example 43 in class *C* we have the same situation as in the previous example. The difference is that class *FIGURE* is not related to *SQUARE* and *CIRCLE* with an ordinary inheritance relationship. In order to maintain the type conformance between *SQUARE* (resp. *CIRCLE*) and *FIGURE* it is necessary that *FIGURE* is equipped with a conversion method for *SQUARE* (resp. *CIRCLE*) instances. An alternative would be to define conversion methods dedicated to *FIGURE* in both classes *SQUARE* and *CIRCLE*.

Rule Adaptation in the Context of No Type Conformance. The class on which relies the parameter type of a method (let us name this class X) which is exherited in a foster class must, either be a superclass of all classes corresponding to the type of that parameter (let us name them $X_1 \ldots X_n$) in the subclasses mentioned in the foster class or, declare conversion routines between class X and all classes $X_1 \ldots X_n$.

Considering Primitive Types

In Eiffel a primitive type is considered as a regular expanded class and it is equipped with some conversion routines (even if they are treated in a special way by the compiler). Then, when

Example 44 Primitive Type Parameter Adaptation

```
class A
feature
 f(p:FLOAT) is do ... end
end
class B
feature
 f(p:DOUBLE) is do ... end
end
deferred foster class C exherit
 A
  adapt f
 end
 B
  adapt f
 end
 all
feature
 f(p:INTEGER) is
  adapted
   {A} (p:FLOAT) is f(p) -- FLOAT
   {B} (p:DOUBLE) is f(p) -- DOUBLE
  deferred
 end
end
```

primitive types are involved in the parameter list of a method some additional adaptations are possible. The principle of those adaptations is the following: when an effective parameter has a smaller type representation than the actual parameter and if their type are compatible, then the statement is correct. For example, if the actual parameter of a feature is of type *FLOAT* or *DOUBLE* then an effective parameter of type *INTEGER* may be used[4] (but not if it is a *BOOLEAN* because its type is not compatible). This situation is illustrated in example 44. Feature *f* in class *C* has a parameter of type *INTEGER* while in the subclasses *A* and *B* the corresponding parameter is of type *FLOAT* and *DOUBLE*.

Rule Adaptation for Primitive Types. The primitive type parameter in the foster class must be compatible with all the corresponding parameters in the subclasses and must have the same or a smaller representation[5] than those in the subclasses.

3.3.2 Return Type Adaptation

Return types may influence the signature of exherited features. Let us analyze what is the impact of return types in two cases when they are primitives and when they are classes. Some adaptations are possible but with certain restrictions. Let us have a look what happens in the situation of type classes.

In example 45, feature *f* is exherited from classes *A* and *B* into class *C*. In each class (*A*, *B* or *C*), the feature *f* returns an object of a certain type. Let's analyze the possible restrictions between these types. If an instance of *A* or *B* is pointed out by a feature of type *C* then a call to *f* will return an instance of type *Y* or *Z*. It is mandatory for types *Y* and *Z* to conform to *X*. Type

[4]The set of *INTEGER* values are included in the set of *FLOAT* or *DOUBLE* values.

[5]By smaller representation we mean that all values of the foster class parameter type must be included in the set of values of each subclass parameter type.

Example 45 Class Attribute and Return Type Adaptation

```
class X
feature
end
class Y inherit X
feature
end
class Z inherit X
feature
end
class A
feature
 f: Y is do ... end
end
class B
feature
 f: Z is do ... end
end
foster class C exherit
 A
 B
 all
 redefine f
feature
  f: X is do ... end
end
```

X being a supertype of *Y* and *Z*, its interface could be a restriction of *Y* and *Z* interfaces. This represents a limitation of the adaptation. Of course, when such a type *X* does not exist initially it could be created by reverse inheritance. This way the covariance principle of the returned type is preserved in reverse inheritance.

If we assume that there is no inheritance relation between classes *X*, *Y* and *Z* than the only way exheritance is possible is when conversion methods are provided to them.

In example 46, a polymorphic call to the feature *f* on an instance of type *A* or *B* through a feature of type *C* will return an instance of type *X*. Since the calls are executed by the feature *f* declared in classes *A* or *B* they will in fact return an instance of type *Y* or *Z*. Conversion methods have to be specified from *Y*, *Z* to *X*. In the proposed example, the conversion methods are located in classes *Y* and *Z*. Another possibility would be that class *X* provides the conversion methods, like it is the case in example 43. Obviously, they cannot be declared in both places since the rules of Eiffel disallow such conversion ambiguity.

Rule Return Type Adaptation. If a feature which returns a result is exherited in a foster class, then one of the following assumptions must be satisfied:

- the return type defined in the foster class must be a supertype of the corresponding feature type in all subclasses;
- conversion routines are defined between the feature return type in all subclasses and the feature type defined in the foster class.

Considering Primitive Types

When dealing with primitive types we can analyze a sample general enough to draw decent conclusions.

Example 46 Class Return Type Adaptation (2)

```
class X
feature
end
class Y
convert
  toX: X is do ... end
feature
end
class Z
convert
  toX: X is do ... end
feature
end
class A
feature
 f: Y is do ... end
end
class B
feature
 f: Z is do ... end
end
foster class C exherit
 A
 B
 all
feature
 f: X is do ... end
end
```

Example 47 Primitive Return Type Adaptation

```
class A
feature
  f:INTEGER is do ... end
end
class B
feature
  f:FLOAT is do ... end
end
foster class C exherit
 A
 B
 all
feature
 f:DOUBLE is
  adapted
   {A} DOUBLE is Result
   {B} DOUBLE is Result
  do
   ...
  end
end
```

Let's examine example 47. The feature *f* from class *A* returns an integer, the feature *f* from class *B* returns a float value and the exherited feature *f* of class *C* returns a double value. If the feature *f* is called through a query whose static type is *C* whereas its dynamic type is *A* or *B* then it is straightforward because a float value and an integer value can be represented by the type *DOUBLE*. If the type of feature *f* in class *C* is *FLOAT* then still there is no precision loss. But if the type of feature *f* in class *C* is *INTEGER* or *BYTE*[6] then the float values returned by the feature *f* in class *B* will be truncated, leading to some precision loss.

Rule Return Type Adaption. When in the foster class, the type of an attribute or the return type of a function belongs to the set of primitive types, then it must be compatible (to satisfy the compatibility rules between primitive types of Eiffel) with the type of the corresponding features in all the subclasses. This implies that the type of the feature in the foster class must be larger[7] than the corresponding one in all subclasses.

It can be noticed that the rules regarding class parameter type and class return type are the same. The idea exploited in these cases is the substitution principle of polymorphism. The parameter and the return type in the foster class have to be chosen in such manner that the instances of the parameter types or return types from the subclasses to be able to be referred by the corresponding ones in the foster class. For primitive types polymorphism no longer holds, so another idea is used. For primitive type parameters, which represent usually input data, the type in the foster class should be tighter than those in the subclasses. In this way, only the subset of common values of all types from subclasses will be used. When dealing with primitive return types the things are opposite. The return type in the foster class must be able to represent all the values which can be returned by any method from the subclasses.

3.3.3 Attribute Type Adaptation

We will analyze attribute type adaptations in this subsection in several cases. In Eiffel an attribute access implies a query which returns a copy of the attribute instance. We will have a discussion regarding types *T1* and *T2* relative to type *T*. First we will consider the case of *T1*, *T2* and *T* being class types.

In case both *T1* and *T2* conform to type *T* we have to deal with a covariant redefinition like in example 48.

In case *T1* and *T2* do not conform to *T* and classes *T1* and *T2* have conversion methods to type *T* and vice-versa class *T* has conversion methods to *T1* and *T2* then we can get a valid attribute exheritance. This is emphasized in example 49.

In example 49 classes *T1* and *T2* are equipped with conversion methods to type *T*, and class *T* is equipped with conversion methods to types *T1* and *T2*. Since such conversions are provided in both ways attribute a of type *T1* respectively *T2*, from classes *A* and *B* can be exherited in class *C* having a new type *T*.

Rule Attribute Type Adaptation. If an attribute is exherited in a foster class, then one of the following assumption must be satisfied:

- the attribute type defined in the foster class must be a supertype of the corresponding feature type in all subclasses;
- conversion routines are defined between the attribute type in all subclasses and the feature type defined in the foster class.

[6]By BYTE we denote an integer type with a limitation of range from 0 to 255.

[7]By larger representation we mean that all values of the foster class return type must include the set of values of each subclass return type.

Example 48 Attribute Type Adaptation (1)

```
class T
end
class T1 inherit T
end
class T2 inherit T
end
class A
 feature a:T1
end
class B
 feature a:T2
end
foster class C exherit
 A
 B
 all
 redefine a
 feature a:T
end
```

Example 49 Attribute Type Adaptation (2)

```
class T
convert
  toT1: T1 is do ... end
  toT2: T2 is do ... end
feature
end
class T1
convert
  toT: T is do ... end
end
class T2
convert
  toT: T is do ... end
end
class A
 feature a:T1
end
class B
 feature a:T2
end
foster class C exherit
 A
 B
 all
 feature a:T
end
```

Example 50 Attribute Type Adaptation (3)

```
class A
 feature a:FLOAT
end
class B
 feature a:INTEGER
end
foster class C exherit
 A
 B
 all
feature
 a:DOUBLE is
 adapted
 {A} DOUBLE is (precursor) Result
 {B} DOUBLE is (precursor) Result
end
```

Considering Primitive Types

When dealing with primitive types in the context of attribute exheritance there can be type adaptation in a restricted way. A favorable situation is when in the foster class there is an exherited attribute having a larger type and in the subclasses there are smaller types for the corespondent exherited features like in sample 50.

When reading attribute *a* using a *C* reference on a *A* or *B* instance there is no problem since the *FLOAT* or *INTEGER* values provided by the subclasses can be represented by the corresponding *DOUBLE* type of the foster class.

When setting the value of an attribute in Eiffel it is used an assigner method, which has a parameter compatible with the type of the attribute. The adaptation of parameters was studied in subsection 3.3.1. The exheritance of an attribute together with its assigner method will be analyzed in subsection 4.3.5.

Rule Attribute Type Adaption. When in the foster class, the type of an attribute belongs to the set of primitive types, then it must be compatible (to satisfy the compatibility rules between primitive types of Eiffel) with the type of the corresponding features in all the subclasses. This implies that the type of the feature in the foster class must be larger than the corresponding one in all subclasses.

3.4 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 an arbitrary type. If the type is constrained to a specific one then it is possible to do more with it in the class. From this dual point of view we start the genericity analysis. We have to specify also that a class can depend on more than one generic parameter, actually it is possible to have a list of formal generic parameters. Our attention is focused on features using generics potentially to be exherited and also on the relationship between foster class and exherited class. It is known that in ordinary inheritance a subclass must instantiate a superclass in the process of inheritance. The same thing happens in the context of reverse inheritance: the foster class will instantiate the generic exherited classes in the process of exheritance.

Example 51 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[DOUBLE]
 B[DOUBLE]
 all
feature
  -- e: INTEGER is deferred end --implicit exheritance
end
```

3.4.1 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 three next 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, can be eventually exherited.

In example 51, the feature e from the subclasses A and B will be automatically exherited while feature f can not be exherited even if in the exheritance branches the same concrete type is used when instantiating the exherited classes. The reason is that such a configuration can not exist in ordinary inheritance where the generic subclass inherits from the foster class and no instantiation is possible between the two classes. The reason why the exheritance of feature f seemed possible is the presence of the subclass instantiation by the foster class. In our example if we parameterized classes A and B with type *DOUBLE* we will actually exherit from two subclasses having *INTEGER* and *DOUBLE* attributes from which we will exherit only attribute e.

Rule Exheriting Generic Classes in a Non-generic One. When subclasses are generic and not constrained but the foster class is not generic, then it is possible to exherit normally features which are not generic, but not generic features even if their corresponding actual parameter types are compatible.

Generic Foster Class and Generic Subclasses

Classes *A[G1]*, *B[G2]*, *C[G3]* are all generic classes. In this case a class hierarchy would look like in example 52. Like in example 51, it is possible to exherit non-generic features as far as they satisfy the constraints mentioned in earlier sections. Because there is no constraint on the generic parameters it is possible to exherit any feature involving a formal generic parameter. As far as the signature of the feature is the same except for the name of the formal generic parameter that may be different, no adaptation is needed. If there is more than one generic parameter, the exheritable feature signatures must have the same ordered generic type set from the foster class. The number

Example 52 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
```

of generic parameters of the foster class or of the exherited classes are not important as far as the exherited feature types point to the same formal generics in the foster class. The instantiation between the foster class and the exherited classes can be reversed and be used in the equivalent ordinary inheritance.

Rule Exheriting Generic Classes in a Generic One. When subclasses are generic and the foster is generic too, then it is possible to normally exherit features which are not generic and also generic features using the same generic type set from the foster class.

Generic Foster Class and Non-generic Subclasses

Superclass *C[G]* has generic parameters while the subclasses do not have.

In some cases (see example 53), it may be interesting to build a foster class which not only is more abstract but also can implement, using genericity, a part of the behavior independently from the data involved (this is particularly useful for data structures like in example 54). 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 the multiple reverse inheritance requires that 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 ordinary inheritance class relationship such an instantiation is very necessary since the foster class must be instantiated in order to be inherited.

Rule Exheriting Non-generic Classes in a Generic One. When subclasses are non-generic but the foster class is generic, then such a class combination is not valid since instantiation

Example 53 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
```

Example 54 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
```

Example 55 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[X]
 B[Y]
 all
 redefine a, m
feature
  a:T
  m(p: T)is do ... end
end
```

information in inheritance does not exist and can not be inferred.

3.4.2 Constrained Genericity

In the case of constrained genericity the generic parameter has a type that conforms to. So there is a minimal set of messages an object can receive through the interface of that type. We take the same structural example like for 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 possibilities for the parameters and types.

Non-generic Foster Class and Generic Subclasses

If *C* has non-generic features, then the non-generic features can be normally exherited. Depending on some 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* conforms to *T*. A special case may be the case where *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 the superclass *C* the 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. In the exheritance branches, concrete types *X* and *Y* must respect the corresponding type constraints specified in the exherited classes and also the requirement related to the existence of a common supertype *T*.

The equivalent class relationship in this configuration is valid, but the instantiation information is lost.

Generic Foster Class and Generic Subclasses

In 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 56,

Example 56 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[G3]
      B[G3]
      all
      redefine m
    feature
      m(p:G3) is do ... end --to write a new code
    end
```

attribute *a* and method *m* can have a generic parameter *G3* in the superclass which conforms to *T3* because *T1* and *T2* conform to *T3* also.

The class configuration is valid in equivalent inheritance because the instantiation information can be inferred.

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 can not be inferred.

3.5 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 first 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.

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[8]:

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

```
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.

3.5.1 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:

```
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 less 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 state their neutral elements. We consider that *E* is a logical expression.

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

```
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 3.2: Exheritance and Assertion Redefinition

The operators *not* and *implies* can not 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 3.2).

In the most 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 57.

In example 57 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 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

Example 57 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
```

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 can not 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 58) then just the exherited features from each subclass **only** list are kept. In example 58 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 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.

3.5.2 Combined Precondition and Combined Postcondition

A different approach on assertions has its origin in [Int05] reference section 8.10.5. There are defined the combined precondition and postcondition of a feature in a subclass having multiple superclasses:

```
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 [Int05] 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.

Example 58 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
```

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* there can be proposed:

```
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 strenghten 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 3.5.1), 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.

3.6 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 some mathematical formula 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 formula (idea taken from scale adaptations) were used. From this point we can generalize and to allow not just mathematical operations but any expression build with the language constructs. The two adaptations related to scale and parameter number use the same syntax build around **adapt** and **adapted** keywords.

Classic signature adaptations take into account parameter and return type modifications. The cases of types created by classes are taken into account and there are provided adaptation solutions based on polymorphism or on the class conversion mechanism existing in Eiffel. With primitive types the implicit conversion of numerical expressions is used for the adaptation. 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.

Feature redefinition in Eiffel allows the use of anchors, so there are some cases of anchored typed adaptations. There was analyzed the exheritance of features which have anchored types referring **current**, argument types and feature types.

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. For postconditions and invariants, in the worst case it can be used the *true* postcondition. Regarding the features which are not exherited and are parts of the assertions there was presented an algorithm for eliminating those features from the logical expression without affecting their semantics. 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 4

Dynamic Binding and Constraints on Exherited Features

4.1 Dynamic Binding of Common Features

Sections 4.1.1 and 4.1.2 deal with the two situations that may occur when exheriting a feature f from a class A to a class C: the exherited version of f and the inherited version of f may have a common seed or not (diagrams *1a* and *1b* from figure 4.1). But the following rules are very important to define the semantics and they are independent of the seed of a feature. Reverse inheritance does not create new inheritance relationships between already existing classes.

Rule Copying Feature. When a method f is exherited from a class A into a class C without its body or if the body is provided either by the class C itself or an ancestor of class C, then the feature f is not moved in class C but **copied** and an implicit redefinition (or undefinition if f is defined in an ancestor of A and not redefined in A) is handled in class A.

Rule Moving Feature. When a method f is exherited from a class A into a class C with its body (it may be declared in an ancestor of A and redefined or not in A), then the feature f may be **moved** effectively in class C and an implicit undefinition may be applied (if f is defined in an ancestor of A) when inherited in class A from those classes.

Sections 4.1.1 and 4.1.2 describe all combinations in the case of single inheritance and single exheritance. Having more ancestors (inheritance) or more subclasses (exheritance) does not change the possible combinations but it may be necessary to undefine, redefine or move up implementation (for exheritance only) in order to remove possible ambiguities.

Rule Handling Possible Conflicts. If a foster class C inherits a feature f from several ancestors then normal rules of ordinary inheritance of Eiffel apply and it may be necessary to use

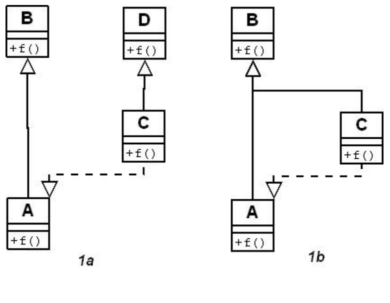

Figure 4.1: Copying or Moving Features - Main Diagrams

Feature implementation in C is / Clause to be added when	D : *e*[4]	A is *e*	D : *e*	A : d[5]	D : d	A : e	D : d	A : d
inherited	none	undefine	none	none	NA		NA	
new	redefine	undefine / redefine	redefine	none	none	undefine / redefine	none	none
exherited	undefine / redefine	moveup	NA		none	moveup	NA	
deferred	undefine	undefine	undefine	none	none	undefine	none	none

Table 4.1: Combinations for Getting the Implementation of a Method in a Foster Class (1)

inheritance clauses to remove the ambiguity. If the same class exherits also a feature f from several subclasses then it must apply also the different clauses (redefine, undefine, moveup) in order to get only one feature in class C. Finally, the rules defined in figures 4.2 and 4.4 are applied.

4.1.1 Multiple Inheritance of Features with No Common Seed

It is necessary to study how to specify where to take the implementation of a feature f when it is both inherited from D and exherited from A in a class C (that is to say when the exheriting class C does not inherit from a direct parent of A). More precisely, the feature f of class C and A do not have the same seed.

Rule Default Handling. For a feature which is exherited, the default is that its implementation is not exherited (equivalent to use undefine when feature is not deferred).

As it has been said earlier, table 4.1 describes possible cases for single inheritance and single exheritance. In the columns of this table the letters specify whether the feature is effective (e) or deferred (d), first in the parent (class D), second in the heir (class A). The rows describe the different alternatives wanted in the new class (class C). The inheritance of class B in class A does not have any impact on the table (rules of ordinary inheritance are followed).

The cells of the table tell what clause is required first for the parent (none, **redefine** or **undefine**), second for the heir (none or **moveup**[1]). Clause **undefine** may be omitted when needed because it is the implicit default, but we decided to keep it for a better readability of the table. In figure 4.2 we keep it also, but it is put between parenthesis. *NA* means that the case does not make sense (you cannot inherit or exherit implementation if the feature is deferred). The different combinations are summarized in figure 4.2[2]. The case numbered with the alphabetical letter number i[3] at line j in figure 4.2 corresponds to the cell row j and column i in table 4.1. For instance the case *3.c* means that implementation is the one found in the exherited class C.

It is important to mention that for attributes it doesn't work exactly in the same way because it is not possible to undefine an attribute in Eiffel when inheriting. The consequence is that all cases which imply to use a clause **undefine** when an attribute is involved in an ordinary inheritance relationship (like *"C inherits from D"*) should not be applicable (*NA*)[6]. In case *2.a* and *2.c* of figure 4.2, we may use the clause **redefine** or **undefine**. The **undefine** clause is particularly useful in

[1]**moveup** means getting the implementation which is pointed out and not only the signature of the feature.

[2]In figure 4.2 we may have the choice of the clause when inheriting from B in A because it depends from the status of f in B. Depending the case it could be either to make f effective (i.e. nothing), **undefine** f or **redefine** f.

[3]For example, *1* (resp. *4*) corresponds to a (resp. d).

[6]For more details about the possibility to undefine an attribute when exheriting, see section 2.3.3.

the framework of multiple exheritance in order to join implementations, in some cases to choose clause **redefine** or clause **undefine** leads to exactly the same semantics.

In the case of exherited implementation, depending on the situation, the routine body is either "moved" to the foster class, or is "copied" so that there will effectively be a redefinition in the subclass - although the code is the same. Namely, there is a difference if the keyword **precursor** is used in the method body, probably also with anchors (***like Current***) - see sections 4.3.2 and 2.5.3. There are some issues to be noted if the feature is an attribute:

- There is no separate implementation (body) in the same sense as for a method (routine).
- If the feature is an attribute already in the existing superclass D, it must be an attribute also in the existing subclass A and in the foster class C.
- If the feature is deferred in the superclass D and is an attribute in the subclass A, it would appear very natural to make it deferred in the foster class by default.

In pure exheritance (without inheritance) it would appear still more natural to exherit an attribute as an attribute by default. However, there are still also good arguments for keeping deferred as the default.

When there are multiple superclasses, things get only a bit more complicated: if the implementation is inherited from one superclass, the feature must be redefined (or undefined) in the inheritance from all other superclasses in which it is effective. When there are multiple subclasses, things get hardly more complicated at all: the implementation can be exherited from only one them, of course.

Rule Undefining Attributes. When a foster class declares an ordinary inheritance relationship then the rules of Eiffel related to attributes fully apply. In particular an attribute may not be undefined. But if it is a reverse inheritance relationship, then any feature may be undefined and thus become deferred.

Influence of Renaming Some discussion about the use of renaming can be initiated according to case presented in figure 4.2. Let us try to remove a possible conflict between the feature f which is inherited from D and the feature f redefined in A which is exherited. Let us suppose that we may use renaming. if f is renamed when inherited from D then it would change the behavior of A because a feature would be added. It is impossible then to merge again in A f_d and f coming from C. If f is renamed as f_a when exheriting from A the problem would be the same. We should use **undefine** or **redefine** instead of **rename**. The only solution for using renaming would be to rename f when inheriting from D and when exheriting from A (and using **redefine** or **undefine** either when exheriting from A or inheriting from D).

Rule Renaming When Exheriting. If a foster class C inherits a feature f from a class D and exherits a feature f from a class A, then renaming (if applied) must be performed for both. If class C has no ancestor containing a feature f, then renaming may be performed freely when exheriting as far as no name conflict is introduced (the reverse is not possible because D may not contain more features than A).

The Case Where There Are No Ancestors Let us revisit the different cases of table 4.1 taking the assumption that class D does not exist. We show that all what had been said in this general framework still apply. How to take the implementation of a feature f when it is exherited from A in a class C (which does not inherit from a class containing f) is quite straightforward, it is described in table 4.2.

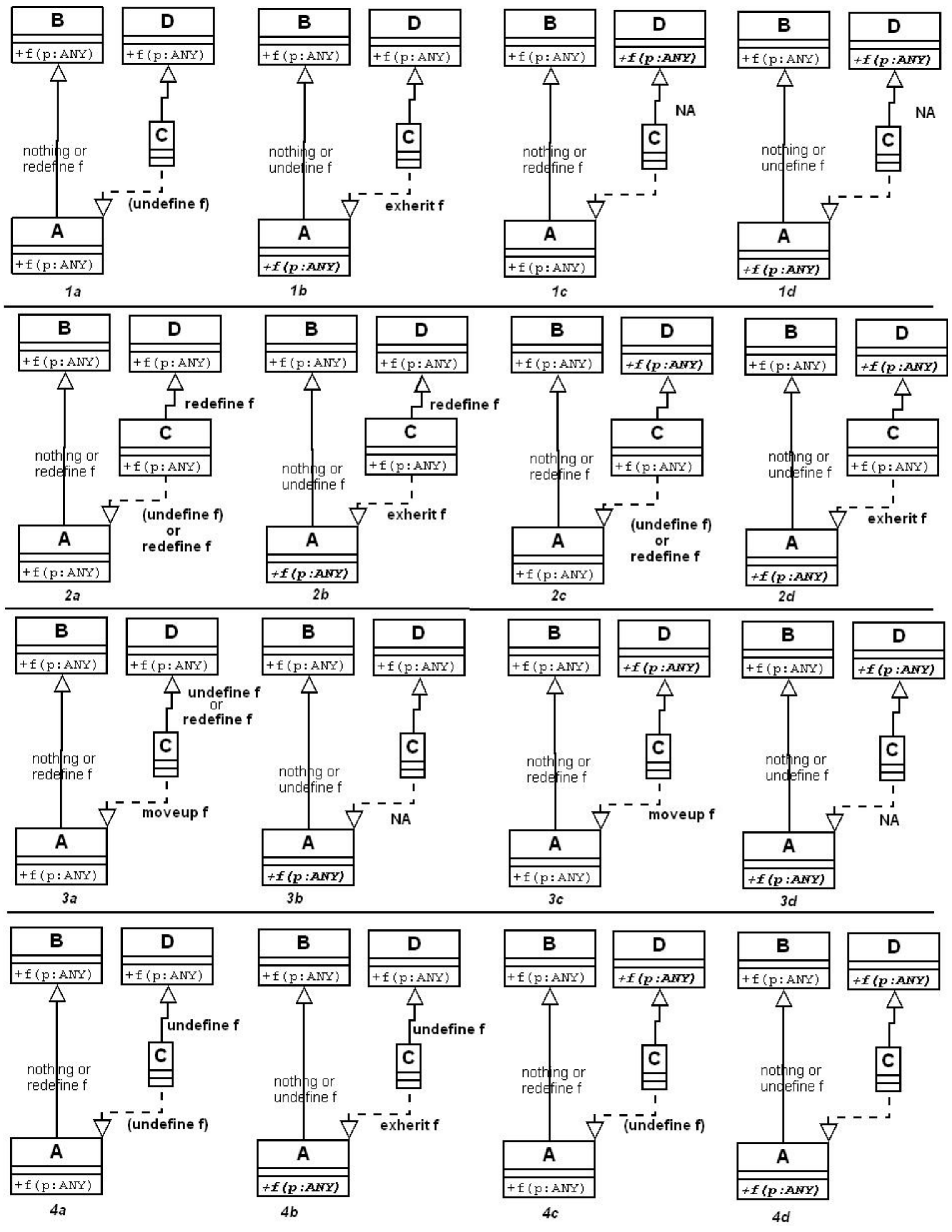

Figure 4.2: Combinations Dealing with Getting the Implementation of a Method in a Foster Class (1)

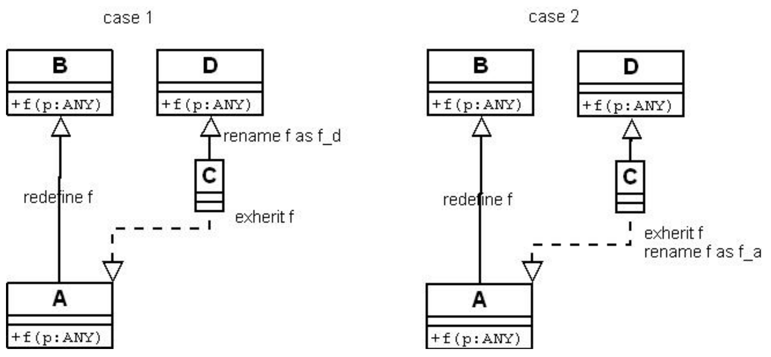

Figure 4.3: Renaming When Exheriting

Feature implementation in C is / Clause to be added when	B : e[7]	B : d[8]
new	undefine/redefine	none
exherited	moveup	NA
deferred	undefine	none

Table 4.2: Combinations for Getting the Implementation of a Method in a Foster Class (1.1)

4.1.2 Multiple Inheritance of Features with Common Seed (Repeated Inheritance)

We think here about where do we take the implementation of a feature f when it is both inherited from B and exherited from A in a class C (that is to say when the exheriting class is "put into the middle"). In table 4.3 .... The default for methods and attributes is identical to the rule defined at the beginning of section 4.1.1. If feature f is defined only in B (and not in A), then it is only a particular case already handled more generally by the rules defined in this table.

In the same way as table 4.1, the letters of the columns of table 4.3 specify whether the feature is effective (e) or deferred (d), first in the parent (class B), second in the heir (class A). The rows set the different alternatives desired in the new foster class (class C).

The cells of the table tell what clause is required (**undefine** is omitted when needed because it is the default), first for the parent (none, redefine or undefine), second for the heir (none or select[9]). NA means that the case does not make sense (you can not inherit or exherit implementation if the feature is deferred). The different combinations are summarized in figure 4.4.

We can see that the content of the cells of table 4.3 are the same as in table 4.1 even if it addresses repeated inheritance instead of just ordinary inheritance. Of course, the clauses used within figure 4.4 are also the same as in figure 4.2. It sounds quite normal for methods as well as for attributes because repeated inheritance of a features has an impact only on the sharing/duplication of objects. Moreover, as far as it concerns attributes regardless if we are in the framework of repeated inheritance or not, the use of the clause **undefine** is not allowed so that the join mechanism does not apply to attributes. Sharing and replication of features is discussed in section 4.2 but let us have a first look on provided facilities:

- A feature f which is declared in a class B and which is inherited into two classes C and A without being adapted (redefined or renamed) is considered implicitly as only one feature in a class X which inherits from A and C. If it is renamed at least in A, C or X, then these

[9]Here **select** means getting the implementation (usual meaning in Eiffel) but does not mean that we choose the implementation in the set of possible implementations induced by possible replications in the framework of repeated inheritance.

Feature implementation in C is / Clause to be added when	B : *e*[10]	A is *e*	B : *e*	A : d[11]	B : d	A : *e*	B : d	A : d
inherited	none	undefine	none	none	NA		NA	
new	redefine	undefine/redefine	redefine	none	none	undefine/redefine	none	none
exherited	undefine/redefine	moveup	NA		none	moveup	NA	
deferred	undefine	undefine	undefine	none	none	undefine	none	none

Table 4.3: Combinations for Getting the Implementation of a Method in a Foster Class (2)

features are considered as different features (duplicated if not redefined or with another semantics when redefined).

- Considering inheritance of feature with or without repeated inheritance, two attributes which have the same name (possibly after renaming) and which are inherited from two different classes into a third one, may become only one attribute if the attribute is redefined in each branch through which the feature is inherited.

- In the same context, two methods which have the same name (possibly after renaming) and which are inherited from two different classes into a third one, may become only one method if the method is redefined in each branch through which the feature is inherited or, if it is undefined in all branches except one.

In figures 4.2 and 4.4, when a case describing a combination of reverse inheritance and ordinary inheritance leads to one of these situations, then the effect will be the same as described above.

A possible code transformation involves creating a class hierarchy in which classes B and A should have the semantics desired as in the original design. The valid cases analyzed in figure 4.2 can be easily transformed into cases of multiple inheritance at the implementation level[12]. Depending on each case in particular, some actions have to be performed in order to get the desired semantics: moving the implementation from one class to another, changing the inheritance clause, adding a new inheritance clause. The valid cases presented in figure 4.4 should be replaced by an equivalent hierarchy in which class C is put between B and A using only ordinary inheritance, and the direct inheritance link between B and A should be removed. Of course, the previously enumerated actions are necessary in this case also.

4.1.3 "select" Like Approach Does Not Solve All Ambiguities

The clause of **select** in Eiffel is not consistent in all cases. A sample of such a case can be found in figure 4.5. 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 59:

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 versions using such a combination. Using another combination the selection could be ambiguous, only a certain suitable combination makes the selection unique as it should.

[12]By implementation level we refer to the stage in which class hierarchies containing reverse inheritance links must be compiled. In our approach, we decided to generate pure equivalent Eiffel code and then to compile it with an ordinary Eiffel compiler. Details regarding the implementation of the reverse inheritance semantics will be provided in the third research report of the PhD program.

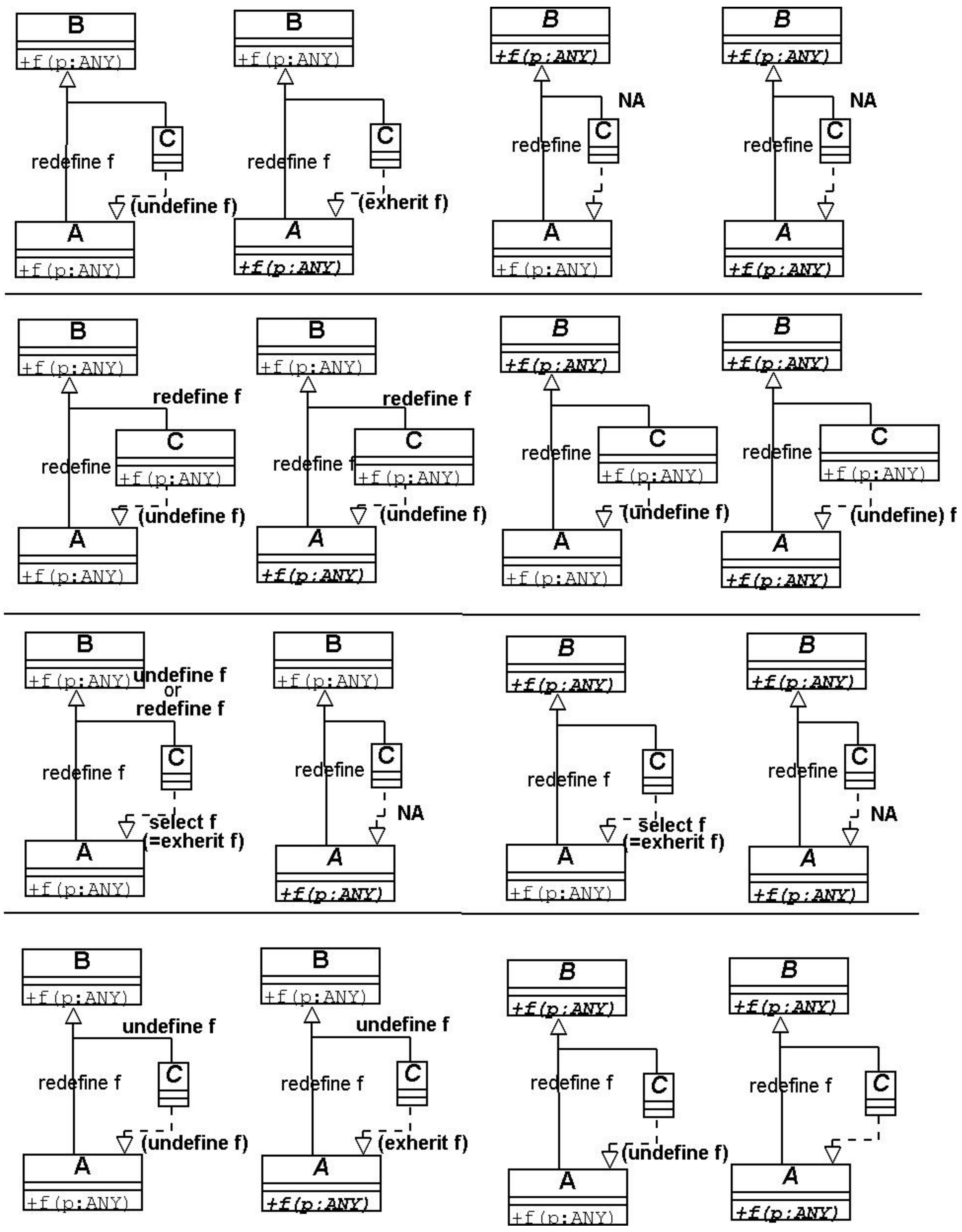

Figure 4.4: Combinations Dealing with Getting the Implementation of a Method in a Foster Class (2)

Example 59 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
```

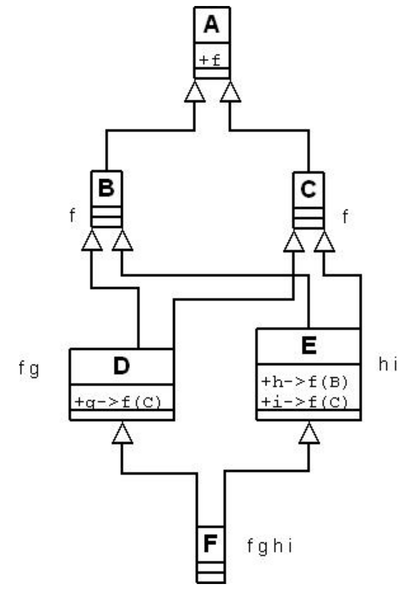

Figure 4.5: Select Problem

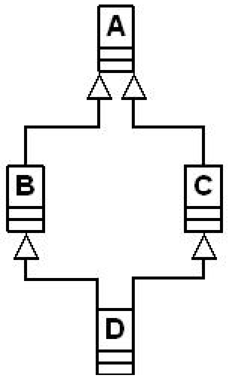

Figure 4.6: Fork-Join Inheritance Sample

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

4.2 Considering the Time Stamp When Defining a Class

In figure 4.6 we study the different orders in which the class hierarchy is built. In this sample class A is the highest ancestor, B and C are then next and finally class D is the lowest descendant.

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 (this corresponds to the same situation as in figure 4.4). 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.

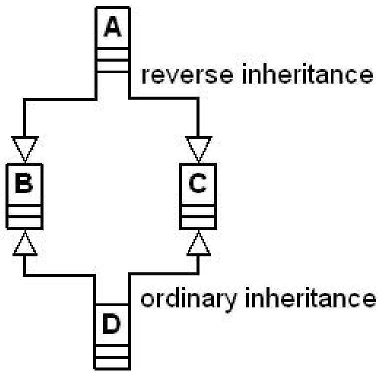

Figure 4.7: Sharing Features (case 1)

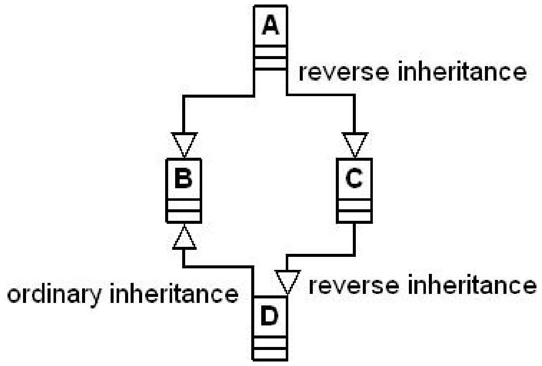

Figure 4.8: Sharing Features (case 2)

4.2.1 Sharing Features

The several 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 only when *B* and *C* have a common ancestor. This restriction really looks like an unnecessary non-orthogonality in Eiffel. There can be imagined some different orders in which the sharing of features from *B* and *C* is prevented by the rules of Eiffel [13]:

- Case *1*: ***BC+D*A***. This means classes *B* and *C* exist initially, 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 4.7).

- 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 4.8).

- Case *3*: ***D*B*C*A***. This means that class *D* exist initially, 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 4.9).

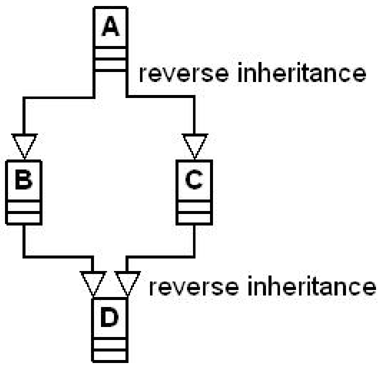

Figure 4.9: Sharing Features (case 3)

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.

4.2.2 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 order where B and C are swapped is 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 4.2 and 4.4) .

- 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 60).

- 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.

[13]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.

Example 60 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
```

4.3 Constraints on Factored Features and Foster Classes

4.3.1 Using the Keyword *frozen* for Features

Since in Eiffel by default all feature calls are dynamically linked, there is no need to call a method using the reference of the same type as the object it points to. It is unlike C++ with its implicit non-virtual methods. The impact of the keyword **frozen** has to be taken into account in several situations dealing with reverse inheritance.

If we deal with frozen features in the exherited classes, then in the superclass the corresponding exherited ones should be deferred. In single reverse inheritance a frozen attribute can be frozen in the superclass if necessary. In multiple reverse inheritance, an attribute can remain frozen only if it has the same type and is frozen in all source classes. A method can remain frozen only if it has both the same signature and the same body in all source classes (and is frozen), which is obviously an extremely rare event.

Let us suppose that a class C exherits a feature f from a class A, if the feature f is not frozen in C, performing it (or reading it) on an object of type A will rely on the dynamic binding rules. According to the rules of Eiffel, the version of the feature which is effective in the exact class of the object will always be chosen. It is not possible to specify in a method invocation that the version belonging to some ancestor class should be called (as one can do in C++).

It is possible to define in a descendant class a new feature with the same name as a feature (frozen or not) inherited from the ancestor, if the inherited one is renamed. According to reverse inheritance if some feature f is not exherited then some other feature can be exherited and renamed to f in the target class. Similar possibilities of "non-virtual redefinition" (or hiding) exist also in some other languages, and they tend to cause confusion. However, that does not affect dynamic binding.

Rule Frozen Features in Foster Classes. A foster class C cannot declare a feature as frozen except if the features exherited from all source classes are:

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. A feature which is frozen in a source class may always be exherited as a non-frozen feature in the foster class.

4.3.2 Impact of "Precursor" Keyword

A subtle point about reverse inheritance and dynamic binding is the behavior of the keyword **Precursor** in methods. Let us suppose that the method f was inherited into class A from a superclass B, where it is effective (not deferred) and was redefined in A. In that case, it is possible to call the inherited version of f from the new version by using the keyword **Precursor**. The two main situations pointed out in section 4.1 are summarized in figure 4.10. The text associated to "*…?…*" depends on the cases presented in corresponding hierarchies.

C is Inserted Between A and B

First let us suppose that C is inserted "*in the middle*", i.e. inherits from B and exherit from A. There are four different cases:

1. **The implementation of *f* is the one of *B***. This means f is inherited from B and not redefined in C and f is exherited from A but (implicitly) undefined. Thus, the precursor version is the same as without reverse inheritance, and there is no problem.

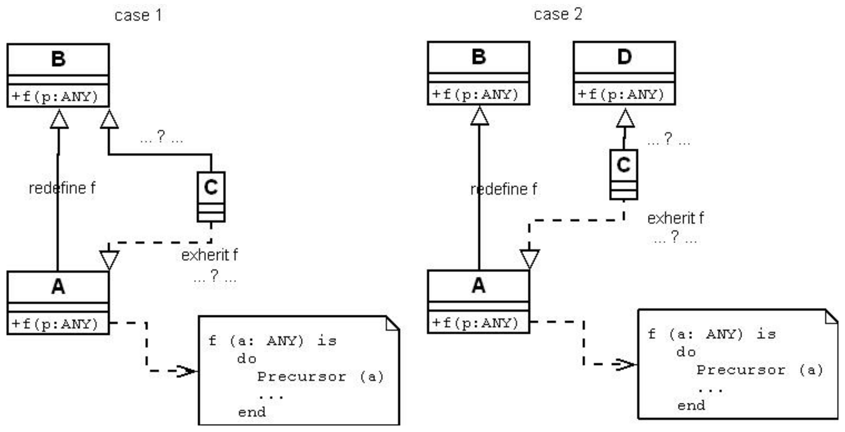

Figure 4.10: Main Configuration When Using the Precursor Keyword

2. **The implementation of *f* is the one of *A*.** Feature *f* is exherited from *A* into *C* and is undefined when inherited from *B*. This means that the redefinition of *f* is effectively moved from *A* to *C*, and again there is no problem.

3. **The implementation of *f* is the one of *C*.** Feature *f* is redefined in *C* either when it is exherited from *A* (and undefined when inherited from *B*) or when it is inherited from *B* (and undefined - implicitly - when exherited from *A*) or both. **Precursor** in the version of *A* will then call this version and not the version of *B*. This means that the behavior of objects of type *A* will change; thus this case must not be allowed.

4. **Feature *f* is undefined in *C*.** This happens if *f* is undefined when inherited from *B* and when it is exherited from *A*. This must not be allowed because **Precursor** would refer to a non-existing method.

C is Inserted Only as a Brother of B

Now let us suppose that *C* is inserted "*as a brother of B*", i.e. does not inherit from *B* in addition of exheriting from *A*. *C* may inherit from *D* but there is no common seed between the features exherited from *A* and those inherited from *D*.

If *D* does not contain any feature which conforms to *f* and has the same name as *f* (possibly after renaming in *C*), then exheritance must not be allowed because **precursor** would not refer to anything. It is the same if *D* contains a feature which does not conform to *f* and has the same name as *f* (possibly after renaming in *C*).

Otherwise, if *f* from *D* has exactly the same body as the feature *f* declared in *B* (this is unlikely), then the four cases which apply when *C* is inserted "*into the middle*" apply here also. In all other cases the behavior of those two features are different, so that the **precursor** of *f* in *A* cannot apply to the feature from *D* and the exheritance must not be allowed.

However, when the behavior of those two features are different, it affects only the objects of type *C* if this class is not deferred, or more generally if it may have occurrences and for the valid combinations described in figure 4.2, the behavior of *A* may not be changed[14].

Interference with Anchors

There is an additional subtlety in both legal cases (i.e. the first and the second) if the expression "like Current" describes a type either in the signature or in the body of *f*. Namely, that causes an implicit redefinition in every class, and its meaning is thus different in *A*, *B* and *C*. In *case*

[14]See rules at the beginning of section 4.1.

Example 61 Exportation and Exheritance

```
class B1
 feature f {C1, C2, C3} is do ... end
end
class B2
 feature f {C1, C2, C4} is do ... end
end
foster class A
 exherit
  B1
  B2
  all
  export {C1, C2} f
end
```

2 that causes no problems, because the meaning stays the same for objects of type *A*. In *case 1*, the meaning changes from *B* to *C*. Therefore, to achieve the effect of no redefinition from the viewpoint of class *A*, it is actually necessary to redefine *f* (automatically) in *C* so that it simply calls **Precursor**. Considering the above complications, it seems that only *case 4* should be allowed if *C* is not defined to inherit *B*.

4.3.3 Exportation and Exheritance

With ordinary inheritance each feature has specified a list of client classes which can access it through an object. With ordinary inheritance, it is possible to change the set of classes to which a feature is exported freely (it may be extended or reduced without constraints). So we propose to provide the same facilities for reverse inheritance. However we have to be careful because to remove classes from the set of exported classes may have an impact on the assertions in the foster class if those assertions refer to the feature.

Moreover, we should remember that in some Eiffel specifications (like ECMA-367 [Int05]), to export a feature to a client class *C* means also to export this feature to all descendants. This means that the export list for a feature can not be shortened in the subclass, the feature may add to the list of clients but it can not remove them. Other specifications, like ETL2 [Mey02] allows shortening the list of clients, thus it is possible to hide features in the subclasses. This approach is sensible to the polymorphic catcalls[15] [Mey97].

The ETL2 rule implies that if *A* exherits *f* from *B* where *f* is exported to classes $C_1 \ldots C_n$ and if the class *A* exports *f* to classes $C_1 \ldots C_{n+1}$ then implicitly *B* will export *f* to C_{n+1} if *f* is **moved** in A. It will be necessary to ensure that the client list to which *f* is exported will be reduced to $C_1 \ldots C_n$ in *B*. But this last rule will not guarantee that *f* may not be called by C_{n+1} on an object of type *B* through an attribute of type *A*.

In example 61 class *A* exherits feature *f* which originally in class *B1* has three classes in the export list *{C1, C2, C3}* and in *B2* has *{C1, C2, C4}*. In foster class *A* the list is shortened to a subset of the original common values *{C1, C2}*.

Rule Exportation List of the Exherited Features. In a foster class the exportation list for an exherited feature must be kept the same or reduced to a 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 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

[15]cat = Changing Availability of Type

Example 62 Exheriting Creation Procedures

```
class B1
 create make, default_create, build
 ...
end
class B2
 create make, build, construct
 ...
end
foster class C
 exherit
  B1
  B2
  all
 create make, build
end
```

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 the catcalls.

4.3.4 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 too. 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[16], 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. Than, we have to list them in the create section of the foster class if the corresponding features in subclasses were creation procedures too. In example 62 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 *C*, too.

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.

4.3.5 Exheritance of an Attribute with "Assign" Clause

There are attributes which have an assign clause 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: 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 be exherited. The syntax must allow to specify an assign method in the foster class through an attribute redefinition.

[16]By moving a feature all related conditions must be respected: resolving dependencies, behavioral consistency of the new hierarchy.

Example 63 Exheritance of an Attribute with Assign Clause

```
class A
feature
  x:INTEGER assign put_x
  put_x(p:INTEGER) is do x:=p end
end
class B
feature
  y: INTEGER assign put_y
  put_y(p:INTEGER) is do y:=p end
end
foster class C exherit
 B
  rename
   x as z,
   put_x as put_z
 end
 C
  rename
   y as z,
   put_y as put_z
 end
 all
end
```

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 63, both attribute *x* of class *A* and attribute *y* of class *B* 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.

4.3.6 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 4.11). 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.

In figure 4.11 class *C* is inserted in the middle of the hierarchy created by *A* and *B*. Feature f from class *A* is exherited into class *C* where *f* gets a new alias. If the aliases for feature *f* in class

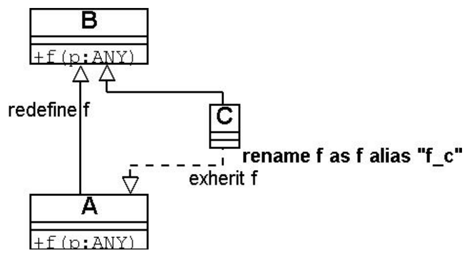

Figure 4.11: Adding an Alias When Exheriting

C are newly added and different from the ones exherited from *A* 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 *C*, in class *A* are implicitly removed. The new aliases of feature *f* from class *C* can be used in any new subclass of *C*, for instance.

Rule Aliases and Reverse Inheritance. The renaming mechanism (which includes the aliases mechanism) does not affect the semantics of reverse inheritance.

4.3.7 Exheriting Obsolete Features

Obsolete features are those old features in a class which are meant not to be used and they may 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[17].

4.3.8 Exheriting Features of Type *once*

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 64 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 that feature *init* is not using any once key, so implicitly is set on *PROCESS*, feature *setup's* **once** key 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.

[17]One of the main concepts from the philosophy of reverse inheritance is to favour feature reuse at the highest level possible. This is why obsolete features are allowed to be exherited without any restrictions, but with a warning.

Example 64 Exheriting Features of Type once

```
class A
 feature init once
  ...
 end
end
class B
 feature setup once(''OBJECT'')
  ...
 end
end
class C
 feature initialize is
 do
  ...
 end
end
foster class X
 exherit
  A
   rename init as start
   undefine start
  end
  B
   rename setup as start
   undefine start
  C
   rename initialize as start
   undefine start
  end
  all
 feature start is deferred end
end
```

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 4.4: Possible Combinations of Ordinary Inheritance and Reverse Inheritance

Rule Exheritance of *once* Features. The **once** mechanism does not affect the semantics of reverse inheritance.

4.4 Constraints on Foster Classes

4.4.1 Using the *frozen* Keyword

In Eiffel the **frozen** keyword applied to a class will restrict that class from being inherited and having descendants. So, the downward evolution of the class is stopped. 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. Relative to reverse inheritance, two problems must be studied: if a foster class can be declared as **frozen** and if it is possible to exherit a subclass which is declared as **frozen**.

If a foster class is declared as **frozen**, in the context of ordinary inheritance, it means that the foster class can not be extended by inheritance, and it can not have new subclasses. On the other hand the foster class may exherit subclasses, otherwise it does not make any sense to declare the class as foster. There are still two reasons why a foster class should not be allowed to be declared as **frozen**: reverse inheritance was designed for class reuse and declaring a class as frozen will restrict the reusability (the foster class may have descendants created by ordinary and reverse inheritance), the second reason is related to the philosophy of the language which does not combine the class modifiers (in order to keep simplicity of the language there is allowed only one class modifier to be stated when defining a class).

The second problem determining whether a frozen class can be exherited or not must take into account that the class was born with the frozen modifier and for some reasons the designer intended to prevent the inheritance of that class. If there are some useful features in that class there is no reason why the class should not be exherited.

4.4.2 Using the *obsolete* Keyword

An **absolete** class in Eiffel is a class which does not meet the current standards and which will 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 (these features should not be marked as **obsolete**). The exheritance of an obsolete class should be allowed only if the hierarchy, from which the **obsolete** class belongs, will no longer evolve in the future.

4.5 Combining Ordinary and Reverse Inheritance

Let us have a look at the possible combination of reverse and ordinary inheritance (see table 4.4):

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

Example 65 Inheriting 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
```

4.5.1 Inheriting From a Foster Class

If we discussed about the exheritance of inherited features, we have to discuss about the inheritance of exherited features. This is one reason to keep the new class relationship as symmetrical as possible.

In order to be more explicit we will start from the example given in subsection 2.3.3.4 and we will develop further within a natural hierarchy development scenario.

In example 65, we created a new class *TRIANGLE* which is derived from *SHAPE*. So 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 of 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 inherited and exherited method *perimeter* and a method implementation *halfperimeter*.

Rule Inheriting from a Foster Class. Features exherited by a foster class will be inherited by any subclass of the foster class.

4.5.2 Inheriting in a Foster Class

A class C which declares a reverse inheritance relationship to a class A may also inherit from a class D if D contains at most the same features as C (this constraint is necessary in order not to change the behavior of descendants of A). It is interesting to get this opportunity when we want to keep and reuse some features that are already defined (see figure 4.12):

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

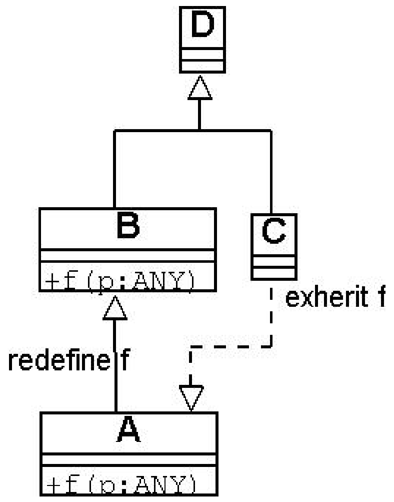

Figure 4.12: Interest for Allowing (Restricted) Inheritance in Foster Classes

case when features are renamed.

Rule Inheriting in a Foster Class. In a foster class C if one method f belongs to an ancestor A of C, then the feature must be also exherited and both of them must have the same name. Then if f is renamed when inheriting from A then it must be also renamed when it is exherited (with the same name, of course).

4.5.3 Exheriting From a Foster Class

In this subsection we discuss the idea that a foster class should or could have another foster class on top of it. From this point of view, comparing to ordinary inheritance, it is possible to create subclasses to already existing subclasses, thus making the class hierarchy to evolve downward. Reverse inheritance would facilitate the upward evolution of class hierarchies like in example 66.

The only delicate issue that can be noted in this situation is the visibility of the implicitly exherited features in cascade. In example 66 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 in a long chain of foster classes, it would be very difficult that in the top of the hierarchy to realize exactly which features succeeded to be exherited finally along the whole path.

On the other hand, the reverse inheritance relationship is meant for using it 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 can be used like in example 67.

The main idea in example 67 is the recommendation 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.

4.5.4 Allowing to Exherit Features from an Ancestor

Another possible lack of clarity may happen when one wants to exherit inherited features from a class, because those features are not listed in the text of that descendant class. The reverse inheritance class relationship allows the exheritance of any feature of the target class: defined directly in the text or inherited.

Example 66 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
```

Example 67 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
```

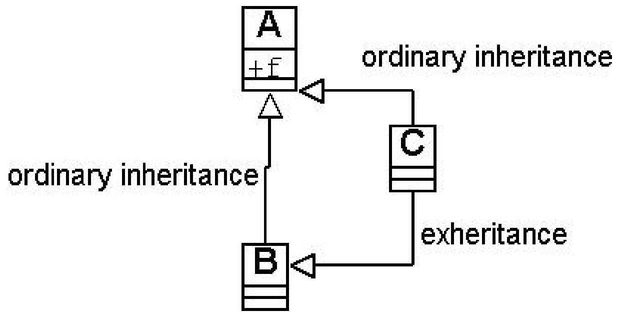

Figure 4.13: Exheriting Inherited Features (1)

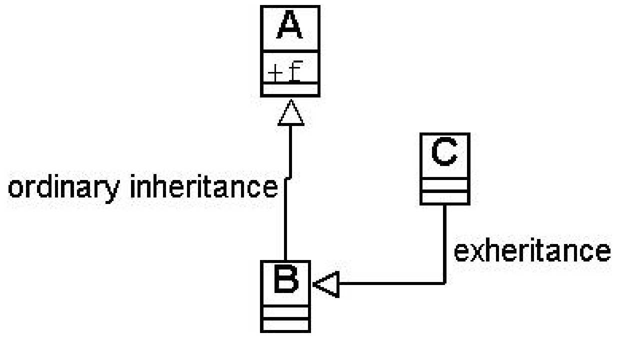

Figure 4.14: Exheriting Inherited Features (2)

Let us get deeper into this subject analyzing the case presented in figure 4.13.

In this sample we analyze the features of class A inherited in class B which are then exherited into class C. We mention also that class C in this situation is a descendant of A. Of course that feature f is inherited in both B and C classes. On the other hand feature f from class B is a potential candidate to be exherited in class C too. In this point a conflict occurs between the inherited feature and exherited one in class C. A priority rule can suppress this ambiguity by setting inheritance to have preference over exheritance by default. If the feature f has a covariant redefinition in class B and if the exheritance of f is desired in class C, it has to be made explicitly. It can happen that in both classes B and C feature f has a covariant redefinition. In such case the effect is implicit feature duplication and a name conflict. This problem can be fixed using the renaming technique.

In the sample described in figure 4.14, we take into consideration the same sample but class C is not a subclass of A anymore.

Such a class construction is equivalent to a multiple inheritance construction where class B is obtained by multiple inheritance out of classes A and C. Because we do not want to affect class B by the features of class C factored by reverse inheritance, we have to disallow replication of feature f in class B. 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 A and B classes. The only restriction is that if f is an attribute in A it has to be an attribute in B as well. It is not possible in Eiffel to undefine an attribute in a subclass, so the decision for an attribute can not be changed further. Unlike attributes, when choosing methods one can change his mind later on. The problem of feature f unification in classes A and C must be analyzed. If f is an effective method in C and not deferred in A, the "implicit ordinary inheritance" from C to B should undefine f, by making it deferred.

Problems arises when the exherited feature f in class C is an attribute and also an attribute

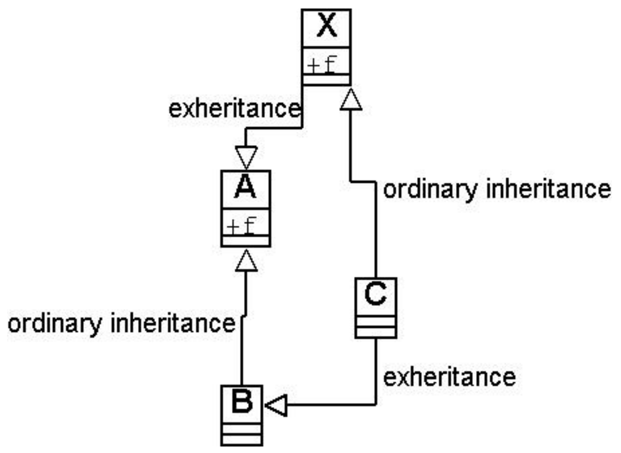

Figure 4.15: Exheriting Inherited Features (3)

or effective method in A. The equivalent multiple inheritance scheme implies that f should be duplicated in class B. The solution in this case comes from using reverse inheritance again like in figure 4.15.

In Eiffel a multiple inherited feature with a common origin will not be replicated. In this direction we created class X as a superclass of A and C by reverse inheritance with A and linked by ordinary inheritance with C.

4.6 Like-Type Relationship in Eiffel

The second type of reverse inheritance is called non-conformance because it has the goal of selective common feature import from exherited subclasses. This mechanism is very similar to roles [VN96, Kri96, TUI05] or views [AB91, KR93, Hil99, UML04] from the database world.

4.6.1 Motivation

In section 1.1.7 we presented an example where like-type class relationship can assist in designing the reverse inheritance class relationship. The main idea of that example is that reverse inheritance helps in splitting the interface of a class while, the like-type class relationship helps in partitioning the implementation of the same class. Sometimes not all the behavior of the class is desired to be reused but only a fragment of it, so the resulted class code will be smaller.

We explain also why this class relationship is still needed in the context of already having non-conforming reverse inheritance. As we already learned with reverse inheritance we can exherit only common features from several exherited classes. With like-type we can exherit also features which are not common in all the target classes. In particular the cases of single non-conform reverse inheritance and single like-type are the same. So in the cases of single like-type class relationship the single non-conforming reverse inheritance class relationship can be used.

4.6.2 Definitions and Notations

The syntax extension for the feature implementation importing class relationship will be the **like-type** keyword. The feature implementations may come from one or multiple classes. The selection of imported features is done using the same keyword **moveup** as for exheritance.

In example 68 we show that class C is built importing features from classes A and B using the like-type class relationship. We suppose that feature *m1* depends only on attribute *f1* and that feature *m3* depends only on attribute *f3*.

Example 68 Like Type Definition

```
class A
feature
 f1:REAL;
 f2:REAL;
 m1 is do ... end
end
class B
feature
 f3:REAL;
 f4:REAL;
 m3 is do ... end
end
class C liketype
 A
  moveup f1,m1
 end
 B
  moveup f3,m3
 end
end
```

4.6.3 Cardinality

Like conforming reverse inheritance, the non-conforming one can be either single or multiple. When like-type class relationship has a single target class from where the features are imported we say it is single like-type. When there are multiple classes targeted then we discuss about multiple like-type. When like-type is single the things are simpler, but when it is multiple then a few problems may occur. These problems will be discussed further in several sections.

4.6.4 Imported Feature Selection

As we discussed in the context of reverse inheritance there are several possibilities of selecting the features to import. One possibility is to select all features from one or more target classes. In case of single like-type such selection would make no sense, thus the target class is cloned in the source class. Ordinary inheritance could be used for such situations. The explicit selection of the imported set is another choice. Selecting all implementations except a certain set is another way of performing the import. Selecting no feature makes no sense, so such selection strategy is excluded.

4.6.5 No Type Conformance

Between the source and the target classes there is no type conformance link, just as in non-conformance ordinary or reverse inheritance. The source class of like-type relationship may have features which are not present in all target classes and also target classes may have features which are not present in the source class.

In example 68 we can see that there is no type conformance between class *C* obtained by like-type class relationship and target classes *A* and *B*. Obviously, class *C* will be equipped with features *f1*, *m1* from *A* and *f3*, *m3* from *B* so there will be no type conformance between *A* and *C* or respectively *B* and *C*.

If no type conformance is provided then it is clear that there will be no dynamic binding issues. Having no type conformance means also that we can not have a like-type class relationship between classes related by direct or indirect ordinary or reverse inheritance conform or non-conform.

4.6.6 Exherited Features

The main idea of this relationship is to import feature implementations in the source class from one or more target classes in order for the source class to reuse them. Imported features act just like they would be defined originally in the source class. This assumption will cause the same family of problems that we had with the **move up** facility of reverse inheritance.

Regarding the target class it can be deferred or effective. If deferred features are exherited, then the source class becomes deferred as well. Such an import of deferred features is not very useful in practice. Taking into account the nature of the features we can see that attributes are very easy to import, but they can be easily rewritten as well. The only interest stands up only for method implementations.

4.6.7 Dependency Problems

When certain behavior is imported into the source class it can be affected by the local calls. These calls link the imported features to behavior that they depend on. So in the problem of resolving dependencies we have several possibilities:

- to import them recursively from the same class, this will lead to importing almost all the behavior from the target class. This can be done automatically or manually by letting the programmer select the import of dependencies recursively.
- to import them from other target classes, the imported features must be selected manually.
- to declare the dependencies in the source class as deferred. This choice would make the class deferred also.
- to let the programmer redefine their implementation.

The most natural approach seems to let the programmer decide between the presented choices by manually selecting the necessary imports.

4.6.8 Possible Name Conflicts

When different features are exherited from different classes then the possibility of name collisions arises. The import of features having the same names is forbidden. Renaming is the mechanism that could solve such conflicts if both implementation are needed to be imported. The same mechanism should be used also in case of dependency conflict.

4.6.9 Exheriting the Inherited Behavior

Another interesting situation arises when we want to import inherited behavior from a class. This behavior is not listed in the target class but it exists there implicitly. The most natural solution would be to restrict like-type class relationship to import only locally declared methods. In case someone needs to import inherited behavior he should import it directly from the superclass that defines it.

4.6.10 Adding New Features

In the implementation of the source class can be added new attributes and methods that may use the imported features further. The same effect can be obtained only by creating a new descendant class using ordinary inheritance and adding in the newly created class the desired behavior. We think that by using this approach, instead of mixing imported features from multiple classes and new features in the same class, we increase the clarity in the design process. In one step we create the class having multiple selected behaviors and in the second step we create a new subclass declaring just the new features.

4.6.11 Conclusions

We can compare this class relation with non-conforming inheritance in both cases: single and multiple. Non-conforming reverse inheritance allows exheriting only the common features, while like-type class relationship non-common ones. Unlike reverse inheritance which implicitly exherits features as deferred, like-type class relationship is interested only in method implementation. At one point the two mechanisms are equivalent: single non-conforming reverse inheritance and single like-type.

4.7 Summary

In the current chapter are presented several ideas about how to handle the special cases in which a class is the target of both ordinary and reverse inheritance. We analyzed two cases of features having or not common seed. In both cases 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 feature, the exherited feature. Also the effect of the **select** keyword is analyzed. The conclusion drawn is that the mechanism does not allow in more complicated class configuration the selection of the desired feature when dynamic linking problems occur. This problem is not addressed by ordinary inheritance, so reverse inheritance will not try to solve it. 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 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 selection mechanism from ordinary inheritance is used in an "a posteriori" manner, meaning that the decision for the selection of a feature is made in the latest built foster class representing the base class of the hierarchy. The difference between the original selection mechanism and the one adapted to the foster class is the specification of the class to which the dynamic binding problem refers. The like-type class relationship semantical elements are issued: definition, notation, cardinality, feature selection, type conformance, dependency problems, name conflicts. Mainly, it has the same class of problems like reverse inheritance.

Chapter 5

Related Works. Conclusions. Perspectives

5.1 Related Works

In this section we will analyze several works related to class hierarchy reorganization, class reuse mechanisms, software adaptation and evolution. There will be highlighted 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 implementation of reverse inheritance may imply a 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.

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 (conform and non-conform).

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).

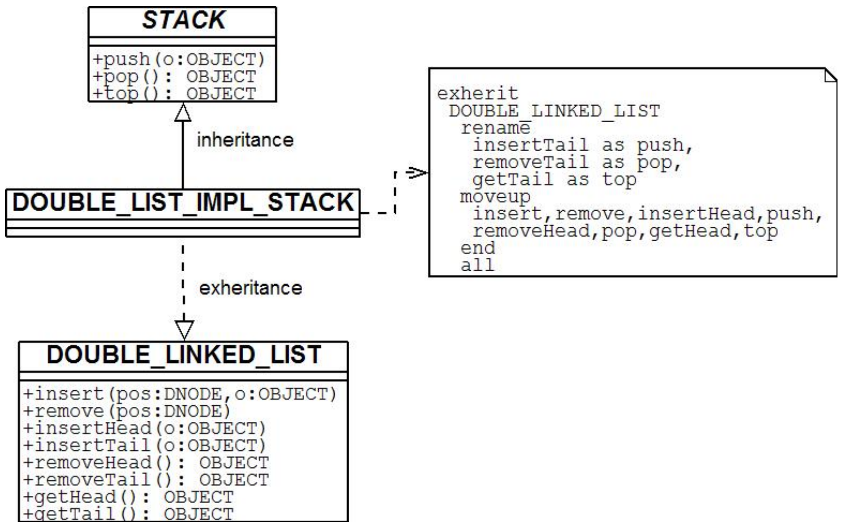

Figure 5.1: Adapter Using Reverse Inheritance

5.1.1 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 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 5.1 and example 69 we show how reverse inheritance can help in implementing the Adapter design pattern from [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 5.2 and example 70).

Example 69 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
```

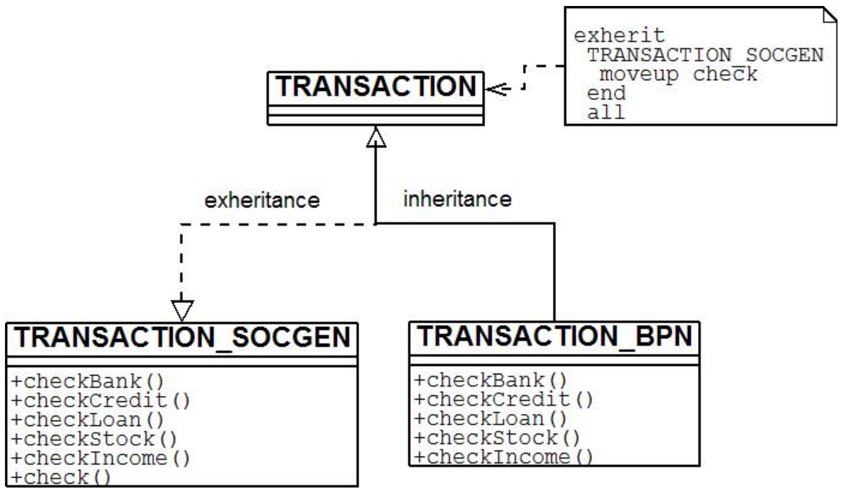

Figure 5.2: Template Method Using Reverse Inheritance

Example 70 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
```

In figure 5.2 and example 70 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 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.

5.1.2 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 71 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 72 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 73. 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

[1]Changing Availability of Type

Example 71 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 72 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 73 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
```

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 73 and it must contain attributes and accessor for storing the value and its coordinates.

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.

5.1.3 Reverse Inheritance and Other Related Works

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. Some implementation related issues will be presented briefly in section 5.3.

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 are already existing. 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.

5.2 Conclusions

Defining the semantics of reverse inheritance and like-type class relationship, several conclusions can be drawn. The choice for the syntactical elements was made taking into account the expressiveness of the resulted class hierarchy. In order to avoid confusion there was pointed out the difference between single, multiple reverse inheritance and several independent reverse inheritance class relationships.

Feature factorization is the key in obtaining an uniform interface for all the different subclasses. The choices for factorization are dual, 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. The nature of the feature attribute or method is important when setting the implicit rules of factorization. However, for practical reasons, to be of more use to the hierarchy designer, we decided that it is better that implicitly attributes are factored as concrete features, while methods or combination of attributes and methods as deferred features, in the foster class. When implementation is subject to factorization, there are quite strong restrictions imposed in order to obtain a valid class hierarchy. Dependencies are the first problems to be solved either by exheriting them or by reimplementing them at the foster class level, but without affecting the behavior of the original classes.

Another key element of reverse inheritance pointed out is the type conformance between subclasses and the foster class. The substitution principle of polymorphism works exactly like in ordinary inheritance. Dynamic binding of common features still holds in the context of reverse inheritance. Because of symmetry reasons the concept of non-conformance reverse inheritance was introduced. In some versions of the language, non-conforming inheritance is used as solution in solving dynamic binding problems. The 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 ordinary and reverse 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. The approach is strongly based on the fact that in ordinary and reverse inheritance features can be redefined. Exheritance clause combinations in context of attributes, methods and mixes of attributes and methods are restricted. Moving a feature and redefining another, or moving at the same time several features are not valid combinations.

Since adaptations are the core of factorization mechanisms permitting the use of several different classes under a common interface, a special chapter presents the related problematics. First the classic adaptation mechanisms like redefinition, undefinition and renaming, are presented how they work in the context of reverse inheritance.

A special set of adaptations are the ones related to signatures, where the scale of the return type, the parameter type and parameter order can be changed. In the study primitive type were considered and also user defined types. For primitive types the idea that one type can include the values of another type was used for determining the representative primitive type that can be used in the foster class. For primitive parameter types they must be smaller in the foster class than all the corresponding types in subclasses, while primitive return types must be larger in the same context. However it is natural that a common interface restricts the primitive type range of the input and to enlarge the primitive type range of the output. For class derived types the things go in two directions either there are type conformance relations between the types and polymorphism substitution is used or there are provided conversion routines. The presented adaptations are not a severe deviation from the philosophy of the language.

Regarding genericity there are a lot of situations for unconstrained and constrained generic subclasses. It is allowed for a foster class to instantiate and exherit several generic foster classes. If the foster class gets also generic then there can be exherited non-generic features and also generic features since there is no constraint about the generic parameters, they can be instantiated with any type. A special useful case 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, it can be always provided by reverse inheritance. The exheritance of features having anchored type must be analysed in two cases: when it is anchored to a type or to **current**.

The redefinition of assertions (preconditions, postconditions and assertions) 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 is used. This approach is not always applicable when preconditions from the subclasses are contradictory. **False** is the strongest precondition, but it will forbid executing the code in the method, 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 factored in the foster class. For this we proposed 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. To the already existing solutions in literature were brought some improvements, making reverse inheritance work in practice, although the solution can be improved.

Dynamic binding problems arise in class configurations where a foster class can get the implementation of a feature from at least two sources: a subclass and a superclass. Such a problem can be controlled by using appropriate inheritance and exheritance clauses in a valid combination. Two big sets of class configurations have been studied: when having or not common seed. The influence of renaming in some cases may cause inconsistent class hierarchies. 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.

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.

Impact of some keywords related to the status of a feature and class were analyzed in the context of reverse inheritance. The most interesting one is related to the use of **precursor**. The exheritance of implementations having precursor limits the capabilities of reverse inheritance, some cases in very strict conditions are still possible. Related to exportation of features, the 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.

5.3 Perspectives

The next step of this research is to prove that reverse inheritance is a feasible class relationship by implementing its semantics in Eiffel programming language. There is no programming language

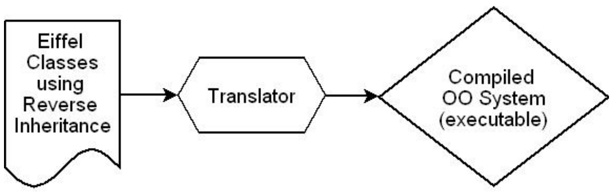
Eiffel Classes using Reverse Inheritance
Translator
Compiled OO System (executable)

Figure 5.3: Implementation Main Idea

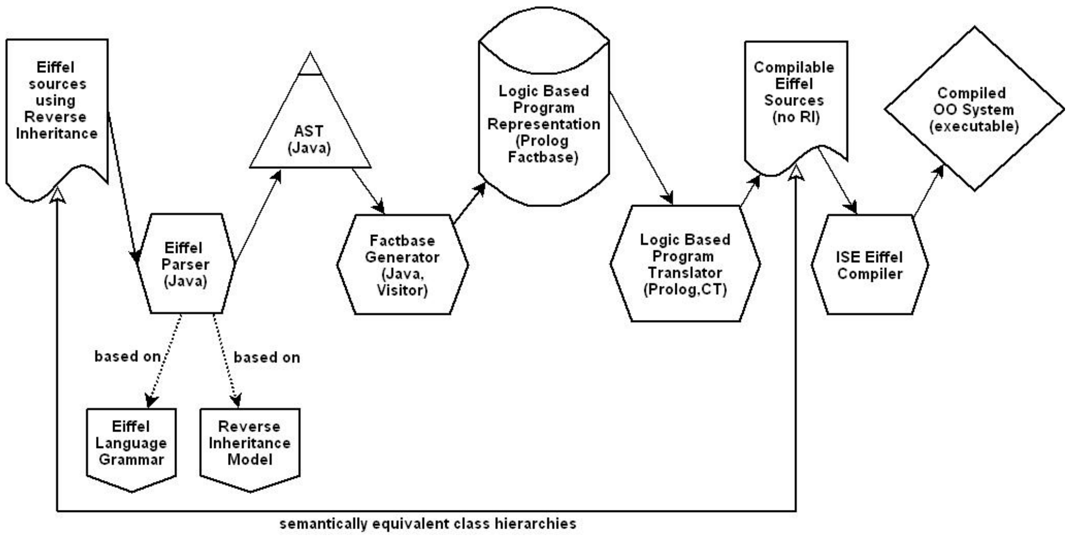
Eiffel sources using Reverse Inheritance
Eiffel Parser (Java)
AST (Java)
Factbase Generator (Java, Visitor)
Logic Based Program Representation (Prolog Factbase)
Logic Based Program Translator (Prolog,CT)
Compilable Eiffel Sources (no RI)
ISE Eiffel Compiler
Compiled OO System (executable)
based on
based on
Eiffel Language Grammar
Reverse Inheritance Model
semantically equivalent class hierarchies

Figure 5.4: Logic Based Transformation Implementation

yet implementing such a concept [Sak02], but this can be done by designing a prototype. The main idea of such a prototype is to be able to translate object-oriented systems based on class hierarchies involving ordinary and reverse inheritance into executable code. In figure 5.3 are presented: the starting point which consists in the classes of a redesigned Eiffel project by reverse inheritance, the translator which is a generic entity for the moment and the target represented by the compiled object-oriented system.

An important issue is related to the nature of the reused classes, which usually are written in pure Eiffel or they can be parts of precompiled libraries. For simplicity, we will consider that the source code of the reused class hierarchies is available. If it is not, there must be performed adaptations at binary code level.

There are multiple solutions to achieve the translation presented in figure 5.3, but we are focusing on an implementation which is designed using logic based program transformation as depicted in figure 5.4.

Next, we will present the steps of an implementation which assumes that class source code is available. The first entity of the diagram contained in figure 5.4 is the collection of source classes which are reengineered by reverse inheritance. These classes organized in hierarchies may belong to different projects, and the need for adaptation is resolved with the help of reverse inheritance.

The source classes using reverse inheritance are the input for the Eiffel parser. The Eiffel parser is based on the Eiffel language grammar [Mey02, Int05] and also on the reverse inheritance model. The model of the foster class is expressed by special rules in the grammar of the extended language[2].

[2]Reverse inheritance is considered to be an extension of the Eiffel programming language.

For creating the Eiffel parser we could use an already existing Eiffel library like **Gobo** [Bez07], which has tools like gelex and geyacc (the Eiffel equivalent of lex and yacc used for compiler generation) and to augment the existing Eiffel grammar with the exheritance related rules. For the fact-base (logic based program representation) generation part we could use the visitor mechanism offered by the generated AST. It will provide normalized Prolog facts about the analysed program. All the information from the AST (Abstract Syntax Tree) are translated into Prolog facts linked together in a relational way, like in relational databases. For example there will be facts representing the classes of the system, the features of those classes, the modifiers used in those classes, etc. The relations between these program element facts can be made using integer keys.

The same job can be performed by a set of tools that can be found in the universe of Java technology like **JavaCC$^{\mathbf{TM}}$[SM]** which can generate a parser from the grammar of the extended Eiffel language, and can facilitate the automatic generation of the AST. The **JavaCC$^{\mathbf{TM}}$** parser generator in combination with **JTree [SM]** or **JTB [JTB]** tree generators will help in obtaining automatically the desired AST. For each grammar rule the tree generators will generate a new class representing a node in the tree. The generated parser is written in Java and also the AST representation is based on Java objects.

The logic based program translator transforms the model in the fact-base using conditional transformations **CT** [KK02]. A conditional transformation is a composable pair of a precondition and a transformation. If the precondition is true then the transformation is executed. For example before adding a new class in the object-oriented system, there must be tested if a class with the same name does not already exist. The transformations refer mainly to the creation of an equivalent class hierarchy through the elimination of the reverse inheritance semantical elements from the model [CKLS07]. For example all foster classes will be transformed into effective or deferred ordinary classes and all exheritance links between classes will be transformed into ordinary inheritance links. This part of the prototype is implemented in Prolog, thus the fact-base generated in the earlier stages can be easily manipulated.

The final step is to generate the pure Eiffel source code from the transformed Prolog facts. These sources would be then compiled by a regular Eiffel compiler like **ISEEiffel$^{\mathbf{TM}}$** or **SmartEiffel**, which will produce the executable binary of the reengineered object-oriented system. This processing stage involves visiting all facts from the fact-base and printing them out conform to the Eiffel syntax.

There are also other possible implementations, like expressing reverse inheritance semantics using other programming languages like C, C++, Java. Another possibility is to design a compiler and to compile directly the object-oriented system, containing ordinary and reverse inheritance, as well, into an executable binary.

List of Algorithms

List of Figures

List of Tables

Bibliography

[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.

[Bez07] Eric Bezault. GOBO Eiffel Project. http://www.gobosoft.com, November 2007.

[CCL04a] 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.

[CCL04b] 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, OCL Project, Laboratoire Informatique, Signaux et Systmes de Sophia-Antipolis (UNSA / CNRS), France, September 2004.

[CCL$^+$04c] Ciprian-Bogdan Chirila, Pierre Crescenzo, Philippe Lahire, Dan Pescaru, and Emanuel Tundrea. 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, OCL Project, Laboratoire Informatique, Signaux et Systmes de Sophia-Antipolis (UNSA / CNRS), France, February 2005.

[CCL$^+$05b] Ciprian-Bogdan Chirila, Pierre Crescenzo, Philippe Lahire, Dan Pescaru, and Emanuel Tundrea. 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, OCL Project, Laboratoire Informatique, Signaux et Systmes de Sophia-Antipolis (UNSA / CNRS), France, 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.

[CKLS07] Ciprian-Bogdan Chirila, Gunter Kniesel, Philippe Lahire, and Markku Sakkinen. Eiffel/RI Project Website. http://nyx.unice.fr:9000/trac, December 2007.

[CPT05] Ciprian-Bogdan Chirila, Dan Pescaru, and Emanuel Tundrea. 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.

[CRC$^+$06a] Ciprian-Bogdan Chirila, Monica-Naomi Ruzsilla, Pierre Crescenzo, Philippe Lahire, Dan Pescaru, and Emanuel Tundrea. 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.

[CRC06b] 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.

[Fow99] Martin Fowler. *Refactoring Second Edition.* Addison-Wesley, 1999.

[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.

[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.

[Int05] ECMA International. Standard ECMA-367 Eiffel: Analysis, design and programming language. www.ecma-international.org, June 2005.

[JTB] *Java Tree Builder.* http://compilers.cs.ucla.edu/jtb/.

[KK02] Günter Kniesel and Helge Koch. ConTraCT - Conditional transformations for incremental compilation of aspects. http://javalab.cs.uni-bonn.de/research/contract/aosdSlides/index.htm, June 2002.

[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.

[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.

[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.

[OJ93] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring, 1993.

[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.

[Sak02] Markku Sakkinen. Exheritance - Class generalization revived. In *Proceedings of the Inheritance Workshop at ECOOP*, Malaga, Spain, June 2002.

[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.

[SM] Inc. Sun Microsystems. *Java Compiler Compiler [tm] (JavaCC [tm])*. http://javacc.dev.java.net.

[SN88] Michael Schrefl and Erich J. Neuhold. Object class definition by generalization using upward inheritance. In *IEEE Transactions*, 1988.

[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.

[UML04] UML Superstructure version 2.0. www.omg.org/uml, October 2004.

[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.