State of the Art in Reuse Mechanisms of Object-Oriented Programming

PhD Research Report #1

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

PhD Supervisor: prof. dr. ing. Ioan Jurca

Faculty of Automation and Computer Science

University Politehnica of Timişoara

8th March 2006

Contents

Abstract

Inheritance is one way to achieve class reusability in object-oriented systems. Reverse inheritance can be considered a special kind of inheritance, where the subclasses exist first and then the superclass is created. It is more natural to define the subclasses first, to notify the commonalities and then to factor them in a common superclass. Using reverse inheritance it is possible to achieve class reusability in several ways: to capture common functionalities of classes, to insert a new class in an already existing class hierarchy, to extend an existing class hierarchy, to reuse partial behavior of a class and to create a new type. Reverse inheritance class relationship equipped with semantics equivalent to direct inheritance is not straightforward. When integrating reverse inheritance in several modern object-oriented programming languages, a lot of conceptual problems arrise, which have to be analyzed carefully.

Acknowledgements

This research report belongs to a PhD programme developed in the collaboration of University "Politehnica" of Timisoara, Romania and the University of Nice from Sophia-Antipolis, France. Until now, I attended two preparation stages at the I3S research laboratories in Sophia-Antipolis where I worked in an international research team. I want to thank **professor Ioan Jurca** for his efforts in supervising my PhD programme and in elaborating the reviews for this report. I would like to thank **M.C. Philippe Lahire** and **M.C. Pierre Crescenzo** for their intellectual and financial efforts invested in the development of the ideas related to the theme of the thesis and sustaining the research. I would like also to thank the team members for their determination in making me advancing on the subject. I would like to thank also **professor Markku Sakkinen** from University of Jyvaskyla, Finland, for the valuable ideas that he gave me during one of my PhD preparation stages. 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** for her commitment and determination in developing the prototypes implied by this research. I would like to thank also the dean of our faculty, professor **Octavian Proştean** and to the chief of our department professor **Vladimir Creţu** for the approval of the financial support in one of my research internships.

Chapter 1

Introduction

1.1 Motivation

One of the most important factors on which the software quality depends is reusability. The benefits of reusability are increased speed of executing projects, decreased maintenance effort, reliability [27]. In object-oriented technology, one way to achieve reusability is by organizing the classes in hierarchies. Currently, class organization is done by inheritance, which is considered one of the basic concepts in the object-oriented paradigm. Inheritance is an incremental modification mechanism which allows the transformation of the ancestor class into a descendant class by augmentation [13]. In practice it has several uses, it can be used for **subtyping** as well as for **subclassing**. From the modeling point of view inheritance can be used either for **classification** or for **implementation**. A very close concept to the concept of inheritance is the reverse relationship, namely **reverse inheritance**.

The idea of reverse inheritance seems to have appeared in the world of objectual database [35], where the main goal is object reuse. Then it was integrated in the context of object-oriented programming languages as **generalization** [30], in order to reuse classes. After that, some ideas of integrating it in Eiffel language can be found in [25], which we admit to be the most advanced approach at the moment. Finally the same concept is discussed in several aspects related to multiple programming languages in the work of [32].

The works of [35, 30, 25, 32] argue about the idea that the reverse inheritance concept favors software reusability in the case of object-oriented systems. The creation of a generalized class which plays the role of supertype and contains all the common features of subclasses is a way of achieving class homogeneity and a better reuse[35]. The interest for such a class relationship can grow when we are dealing with subclasses which belong to a library and have read-only source code or even worse, the source code is not available [32]. The reason for which a library is read-only may vary: copyright reasons, maintenance reasons. We can mention also that this class relationship was neither completely developed in the literature, nor integrated in a programming language[32]. Next, several ways in which reverse inheritance can be usefull in class hierarchy reorganization are presented in more details.

1.1.1 Designing in a More Natural Way

In [30] it is stated that reverse inheritance is a more natural way for designing class hierarchies. When modeling classes, it is considerred 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.

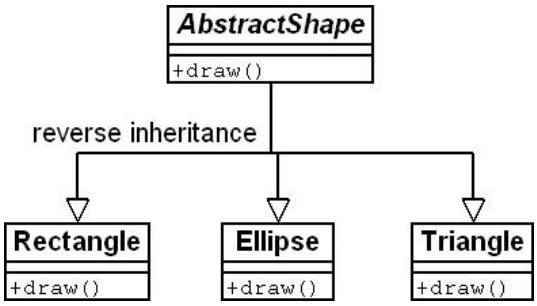

Figure 1.1: Capturing Common Functionalities

1.1.2 Capturing Common Functionalities

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

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

1.1.3 Inserting a Class Into an Existing Hierarchy

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

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

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

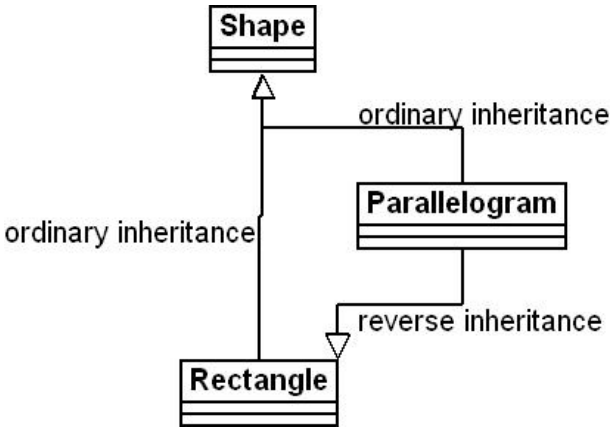

Figure 1.2: Inserting a Class Into an Existing Hierarchy

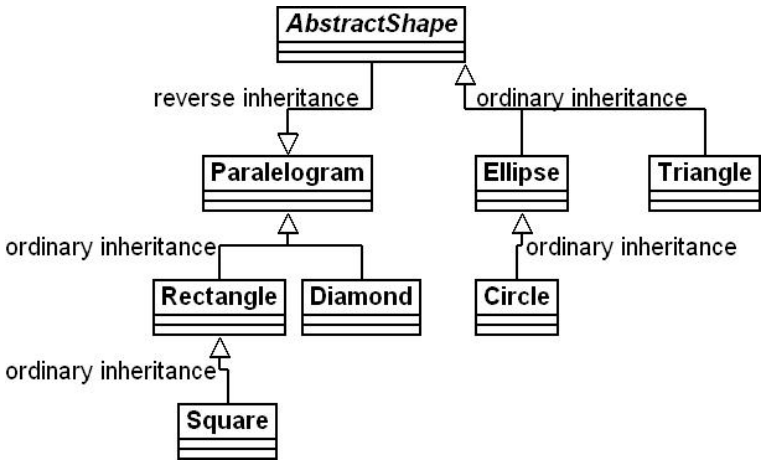

Figure 1.3: Extending a Class Hierarchy

1.1.4 Extending a Class Hierarchy

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

In the use case depicted in figure 1.3 we have a situation of class hierarchy modeling shapes. Initially only the hierarchy rooted by class *Parallelogram* existed and it could not be modified. As a first step of the redesign 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

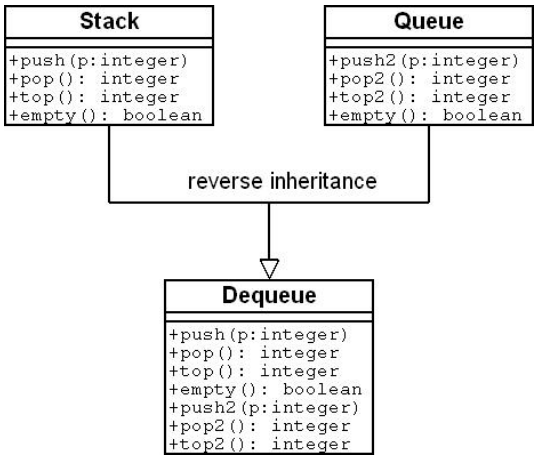

Figure 1.4: Reusing Partial Behavior of a Class

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 related to one end of the *Dequeue* class. A new class *Queue* is then created to get the operations related to queue abstract data type. In conclusion the programmer has the choice of reusing several parts of the code written in a class.

1.1.6 Creating a New Type

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

1.1.7 Decomposing and Recomposing Classes

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

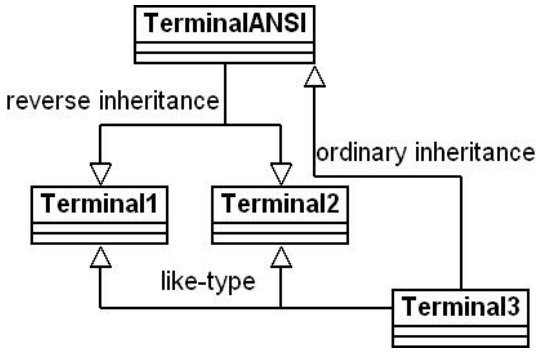

Figure 1.5: Creating a New Type

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

1.2 Overview on the Inheritance Class Relationship

The origin of inheritance dates from 1960 and was introduced in the Simula language where it was known under the name of **concatenation** [43]. The inheritance concept can not exist without the concept of class through which the objects are defined. The class is considered to be the building brick of every object-oriented system having the role of both type and module [27]. Classes are organized in hierarchies representing the backbone of almost every object-oriented system. They contain the state and the behavior of objects. There is no final definition for inheritance and its implementing mechanisms. Next, several informal definitions of inheritance are provided from the literature.

Inheritance is a class relationship where one class shares the state and behavior defined in one or more classes, so classes can be defined in terms of other classes. A subclass redefines or restricts the existing structure and behavior of the superclass [9]. Inheritance in [27, 2] is defined as module extension mechanism because it makes possible to define new classes from existing ones by adding or adapting features, and as a type refinement mechanism which allows the definition of new types as specializations of already existing ones. Inheritance is commonly regarded as the feature that distinguishes object-oriented programming from object based programming or other modern programming paradigms [43]. It supports the construction of reusable and flexible software. In the sense of object-oriented programming, inheritance is an incremental modification mechanism that transforms an **ancestor** class into a **descendant** class by augmenting it in various ways [13]. The ancestor class is known also as **base** class, **parent** class or **superclass** and the descendant class as **derived** class, **child** class, **heir** class or **subclass**. A class is **abstract** if it has a partial implementation and as a consequence it can have no instances. Such a class may contain abstract and concrete members. In Eiffel language [28], an abstract class is known also as a **deferred class**, an abstract feature as a **deferred feature** and a concrete feature as an **effective feature**.

Inheritance brings several benefits like code and data reuse, class hierarchy conceptual organization, rapid prototyping. There are also some drawbacks of inheritance class relationship. Execution speed is affected because object-oriented programs must include code implied by several supporting mechanisms like: constructors, method calling mechanism (polymorphism and parameter transmission), garbage collectors, run-time type checkers. Another consequence of the supporting mechanisms in object-oriented systems is program size. All of the mechanisms men-

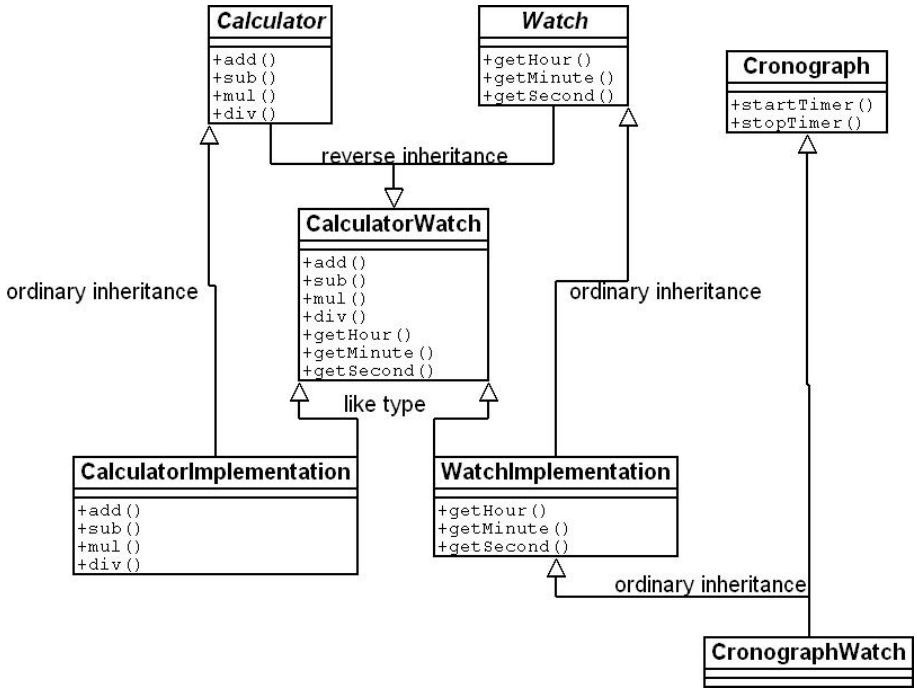

Figure 1.6: Decomposing and Recomposing Classes

tioned earlier imply routines which will be executed in runtime. So their code is added to the object-oriented system. The complexity of the object-oriented systems is higher compared to other systems designed using non object-oriented based paradigms. The complexity of the software systems is managed by the complexity of the object-oriented paradigm concepts.

In [27] a taxonomy of inheritance uses is presented. Each possible purpose of inheritance is individually analysed.

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

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

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

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

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

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

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

Structure inheritance applies between a deferred superclass defining a property and a sub-

class which models an object having that property. For instance a class COMPARABLE will be the superclass of all classes which support the comparing functionality.

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

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

In [7] it is considered that inheritance has mainly two different viewpoints: extension and specialization inheritance. The class relation is used by programmers in modeling, to express conceptual relations between classes and to share code between classes. It is presented a new abstraction mechanism named component, a solution that integrates both views of inheritance in an object-oriented language. The component is a non-instantiable collection of data and related operations. Classes can be composed of such components. Inheritance works at two levels: at component level which has the purpose of code-reuse and class level inheritance for subtyping.

Inheritance represents an important motive for divergence in the community of researchers because of its different uses and implementations in the programming languages. There are a lot of works showing positive and negative examples of how inheritance must be used [26, 43].

1.3 Inheritance in Object-Oriented Programming Languages

In this section we will discuss about how the features of the inheritance mechanisms are implemented in several programming languages. In C++ [41, 34] there are mainly two kinds of inheritance: public and private (protected). The public inheritance is allowing the inheritance of superclass members with their default visibilities. Private inheritance hides the public and protected inherited members making them private in the subclass. Protected inheritance affects only the visibility of the public members, being protected in the subclass. On the other hand private inheritance involves that the subclass will be no longer a subtype of the superclass. That means that the type conformance relationship between the classes is disabled. In the case of multiple inheritance there is a special kind of inheritance called virtual inheritance. This is due to the virtual declarations of the base class which imply a special sharing behavior of the inherited features when multiple inheritance paths are available. These aspects will be detailed in a later section.

In Eiffel [28, 2] there are two kinds of inheritance: conforming and non-conforming. The feature inheritance mechanism is the same in both kinds of inheritance. The difference appears related to the subtyping relationship between the superclass and subclass. With conforming inheritance the subtyping relationship holds, while with non-conforming it doesn't. A special capability in the context of inheritance in Eiffel is the feature redeclaration. This may imply feature renaming, since it is considered that in the subclass, a new name may increase clarity, or redefinition, meaning change of signature or implementation. Of course, signatures may be changed in conformance with the rules of covariance. For this, a set of keywords are used: *rename*, *undefine*, *redefine*, *select*.

In Java [8] the inheritance mechanism always involves subtyping. The inheritance mechanism involves single subclassing and multiple subtyping. In other words, classes may have only one superclass while interfaces can have multiple superinterfaces. On the other hand, classes may implement multiple interfaces. By interface we refer to a special concept, which behaves like a pure abstract class and has only abstract methods. This approach avoids all problems encountered in some of the complex cases of multiple inheritance. In C# [11] we will find the same behavior as in Java, but some concepts may be named differently.

In the context of inheritance we can discuss also about inherited features or members. In an Eiffel class a feature has an unique name, which can not be overloaded. In C++ and Java it is possible to define several methods with the same name but they are obliged to have different

[2]Constant attributes are those attributes that hold a readonly value.

[3]Once functions differ from the ordinary functions in the sense that their body is executed only once on an instance, no matter how many times it it called.

signatures. By signature it is meant parameter number and types. Return types do not belong to the signature.

1.4 Document Outline

We propose to analyze the features of reverse inheritance class relationship from the conceptual point of view. The report is structured as follows. Chapter 2 presents the most important reuse mechanisms of object-oriented technology. In chapter 3 we discuss generalities regarding reverse inheritance, like basic principles, notations in different approaches. Chapter 4 deals with exheritance at class interface level. There are analyzed which features from the class interface can be exherited and what major problems are encountered. In chapter 5 implementation exheritance issues are discussed. In chapter 6, some interesting combinations of ordinary and reverse inheritance are studied. Chapter 7 points out the conclusions of our analysis and the future work.

Chapter 2

Reuse Mechanisms in Object Technology

In this chapter the principles behind the most significant reuse mechanisms in object-oriented programming will be discussed. The interest goes in the direction of mechanism design, encountered problems, possible compromising solutions and its supporting motivation. Implementation issues are also a goal for this section, as they could be reused or similar ideas could be developed starting from them. The proposed mechanism for analysis refer to central concepts of object-oriented technology like inheritance, mixins, traits, roles, separation of concerns with its object paradigms (aspect oriented programming, composition filters).

2.1 Multiple Inheritance

In this subsection we will discuss several issues about multiple inheritance since it is a special form of inheritance, which is the base concept of the object-oriented paradigm. Multiple inheritance is the one of the most powerful facilities offered for the software development allowing to combine several concepts in one abstraction. Disallowing inheritance to accept multiple parents would limit the potential of inheritance in general [28]. Like single inheritance, multiple inheritance is used to extend the module (class) and to create a powerful type system in applications. Inheritance is **single** if the subclass has one parent or **multiple** if the subclass has multiple parents. Because a class can inherit parents in more than one way, it is encountered the case of **repeated inheritance**. We will focus on the implementations of Eiffel, C++, Java and C# statically typed programming languages, analyzing the problems and the existing solutions. The interesting points of discussion are name clashes, duplicating and sharing features, dynamic binding issues.

In Eiffel, multiple inheritance occurs even when the same parent class is inherited twice by the same subclass. This is also known as the repeated inheritance case [28]. If a class has no parents declared, it implicitly is considered that inherits from class *ANY*, which is the base class of any user defined class. In practice multiple inheritance is used in describing the basic data structures implemented in the base library of the Eiffel language. It has been circulated the idea that multiple inheritance is a dangerous and destructive concept [28]. This is not a justified opinion, but in practice it results from imperfect implementations and its improper uses. When using it properly, it permits combining abstractions, being a key technique in object-oriented development. As graphical convention, multiple inheritance can be represented like in figure 2.1. We used the UML notation to show that class C has two parents A and B.

In C++ this kind of multiple inheritance is named also as **independent multiple inheritance** [42], because there is no dependency between the superclasses. This name allows separating from the case of repeated inheritance.

In multiple inheritance one technical problem is the **name clash**. This happens when several features with the same name are inherited from different parents. In the Eiffel philosophy, features

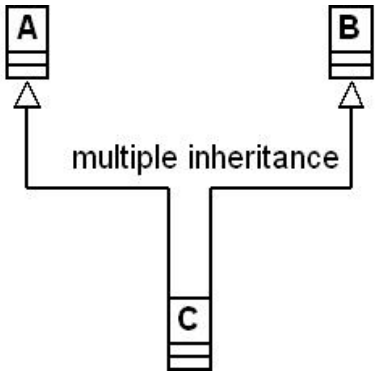

Figure 2.1: Multiple Inheritance

Example 1 Multiple Inheritance Name Clashes

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

can not be overloaded within a class [28], each feature has a unique name, but even in languages which support overloading (like C++ or Java) the conflict persists for features with identical signatures. Such a name clash situation is represented in example 1, taken from [27].

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

In C++, multiple inheritance presents the same problem of name clashes, but a different solution is used, the one of explicit designation [42, 41]. The individual selection of one or another inherited feature with the same name, is made with the help of the full qualification. So, the resolution operator "::" is used in this sense. We will revisit the same example in the context of C++ (example 2):

It can be noticed that in the subclass, using the name of the superclass for the inherited features, they can be distinguished without any problems. In case of a more complex hierarchy the resolution operator can be used repeatedly.

In Java[8] multiple inheritance is possible for types but not for classes. In other words this means that interfaces can be multiply inherited while classes can not. For classes there can be used only single inheritance, this being a way of avoiding the multiple inheritance problems. In

Example 2 Multiple Inheritance Conflict Resolution in C++

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

Example 3 Multiple Inheritance Conflict Resolution in Java

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

the case of interfaces may appear some conflicts. If two superinterfaces declare a field[1] with the same name in each of them, then in the common subinterface any reference to that field causes ambiguity errors [8]. Multiply inherited interface fields through different inheritance paths are unified into a single feature [8].

We took a sample from Java Language Specification book [8] in order to exemplify the two possible conflict situations that may arise. Fields *RED*, *GREEN*, *BLUE* are multiply inherited by both interfaces *RainbowColors* and *PrintColors* from *BaseColors* interface. Then they are inherited into *LotsOfColors* interface through multiple paths. The access "*CHARTREUSE = RED+90*" will not arise ambiguities since *RED* member is unified at the subinterface level. It is not the same case for *YELLOW* member defined in *RainbowCollors* and in *PrintColors* with different values. A potential reference to *YELLOW* member in the subinterface will determine a name conflict. It is not clear which features should be taken into account the one with value 3 or the one with value 8.

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

Figure 2.2: Direct Repeated Inheritance

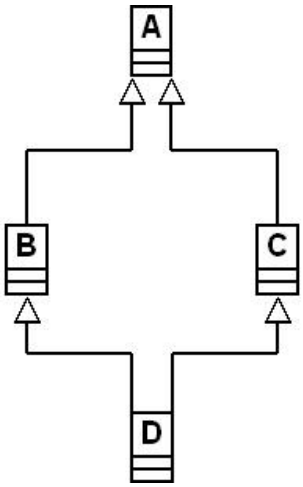

Figure 2.3: Indirect Repeated Inheritance

2.1.1 Repeated Inheritance

In the context of multiple inheritance we encounter the case of repeated inheritance, because a class can be the descendant of another in several ways [28]. There are two cases of repeated inheritance: **direct repeated inheritance** and **indirect repeated inheritance**, like in figures 2.2 and 2.3:

Repeated inheritance arises when two or more parents of D have a common parent A. We have to state that direct repeated inheritance is not allowed in all languages. For example in Eiffel it is possible but in C++ it is not. Class D is the repeated descendant of A and A is the repeated ancestor of D. There are two problems about repeated inheritance which must be solved: the fate of the repeatedly inherited features and the solutions in the resolution of dynamic binding ambiguities [28]. The repeatedly inherited features could be **replicated** meaning that there will be one copy for each inheritance path or **shared** meaning that one unique copy will be inherited. The solution proposed in Eiffel [28] is to be able to decide for each feature independently how to deal with it. This can be done using the renaming mechanism in the following way: shared features will have the same name in the subclass and replicated features will have different names. If we want to replicate repeatedly inherited features we have to change their names using the renaming mechanism. The implicit behavior is the sharing of repeatedly inherited features. In [28] there is an example about a situation where one feature should be replicated and one should be shared (see figure 2.4). Class *HOUSE* has two members *street_address* and *insured_value*. Then two subclasses are created *BUSINESS* and *RESIDENCE*. Later a class *HOMEBUSINESS* is built as

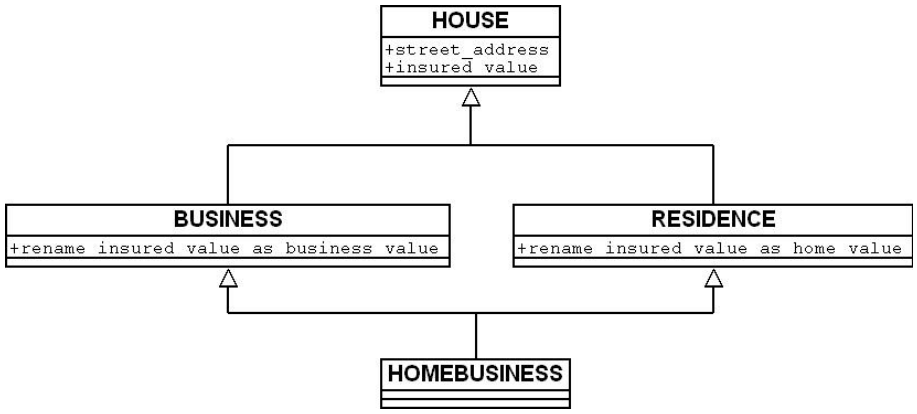

Figure 2.4: Replicated and Shared Features in Repeated Inheritance

Example 4 Repeated Inheritance in C++

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

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

In C++ there is one global selection possibility for the multiply inherited features [34, 41]. Implicitly multiple sub-objects are created when such a subclass is instantiated. Example 4 presents such a situation where class L is the superclass of A and B. Later class C is created as a subclass of A and B. Under these circumstances an instance of C will contain a sub-object L corresponding to A and another sub-object L corresponding to B.

C++ offers also the other possibility of sharing features. This can be done by declaring the base classes as virtual, so any virtual base class will generate a single sub-object [42]. In example 5 we studied the behavior of the virtual base classes mechanism. In this case we decided to obtain in class C only one copy of the features from the base class L and in class D two copies: one inherited from C and the other directly inherited from L.

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

Example 5 Virtual Base Classes in C++

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

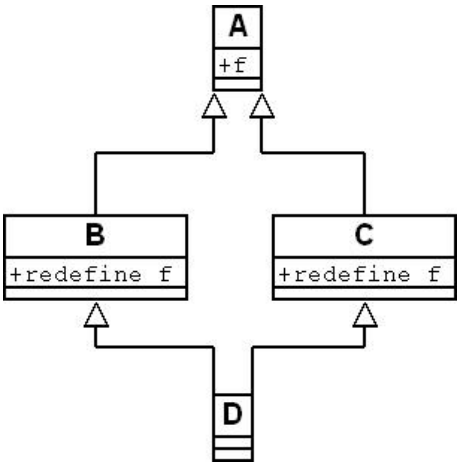

Figure 2.5: Redefined Features in Repeated Inheritance

Example 6 Deffering Multiple Inherited Features

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

Dynamic binding problems occur in situations like the one presented in figure 2.5 where the redefinitions of a repeatedly inherited features are made. In Eiffel features can be redeclared [28], this can imply redefinition or effecting. Redefinition means that a feature which lives in the superclass, gets a new implementation or a new covariant signature or a new set of assertions in the subclass. Redeclaration happens even if a deferred feature in the superclass is effected in the subclass. The problem of dynamic binding appears when on an instance of class D, referenced through a variable of type A, the feature f is called. Because feature f is redefined in B and C classes it means that there are two implementations available and name conflict arises. There can be analyzed two cases: one in which features are intended to be **shared** and one in which features are intended to be **duplicated**. When sharing features the only possibility to eliminate the name conflict is to deffer all conflicting features except maybe one. In this way only one implementation will be available for feature f in class D.

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

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

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

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

Example 7 Replicating Multiple Inherited Features

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

Example 8 Multiple Inheritance Dynamic Binding Case (1)

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

Example 9 Multiple Inheritance Dynamic Binding Case (2)

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

Example 10 Multiple Inheritance Dynamic Binding Case (3)

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

Example 11 Disabling Polymorphism

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

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

The dynamic binding problem is more severe in this case since there are two versions of the feature in the subclass. A call to the f feature on a D typed object will remain ambiguous unless some criteria is used in favouring one of the implementations. In [28] there are discussed several solutions. The first solution is to use the implementation of the class which is the first listed in the inheritance clause. This approach would change the semantics of a class when changing the order of the inheritance clauses. On the other hand when dealing with more complicated class hierarchies having several features it will lead to impossible situations of selecting different implementations from different superclasses. The second solution is to use a special **select** keyword in the language, which allows to declare directly the choice in the favor of one implementation. The third approach comes with the idea of disabling polymorphism on the several inheritance paths, except one from where the implementation will be achieved. In example 11 on the inheritance path corresponding to class C the polymorphism is disabled using the **expanded** keyword. This means also that the subtyping class relationship between instances of D and C are cut.

2.1.2 Implementations of Multiple Inheritance

In [10] there are presented several implementation techniques of the multiple inheritance concept in different type of programming languages. These techniques involve class hierarchy transformations in order to integrate this concept in languages with single inheritance and even with no inheritance. These transformations intend to maintain as much as possible the model of the original class hierarchy, to respect the polymorphic behavior of strongly typed languages, avoid excessive code repetition.

There are several basic transformations available for different kind of inheritance models. For languages having no inheritance there can be performed translations like emancipation, variant types or simulation using flags, composition. When dealing with a language having single inheritance there are possible techniques of expansion or mixed techniques. In case of single subclassing and multiple subtyping (which is the case of Java [8] and C# [11]) a mixed strategy has to be used. In the case of languages with multiple inheritance the only concern is the resolution conflict mechanism. Of course, the techniques presented in the context of languages with no inheritance can be applied also to languages with single inheritance, single subclassing and multiple subtyping, multiple inheritance. These basic transformations will be exercised on a demonstrative multiple inheritance class hierarchy.

In figure 2.6 is presented a representative hierarchy which will be transformed using each of the transformations previously enumerated. It can be noticed that there is a complex case of repeated

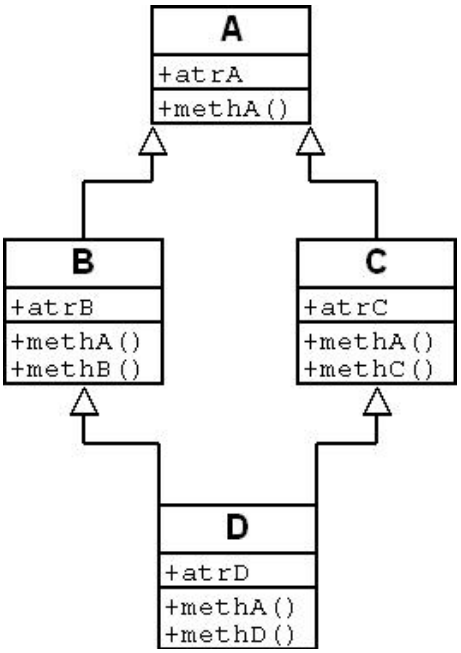

Figure 2.6: Multiple Inheritance Class Hierarchy

inheritance with multiply inherited features through several inheritance paths. Attribute *atrA* is inherited from A to D through B and C classes. On the other hand method *methA* is overridden and has different implementations in each subclass of the hierarchy. We have to mention also that from the typing system point of view there are some subtyping class relationship in this class hierarchy: B and C are subtypes of A, and class D is a subtype of both classes B and C.

2.1.2.1 Emancipation

The strategy of emancipation [10] involves cutting all inheritance links between classes and including all exhibited features as own resources. This strategy is also known as flattening [27]. A special attention has to be given to the several versions of a method and its "super" like calls. All these have to be renamed in order to keep the behavioral consistency of the methods. In figure 2.7 we can see the effects of the transformation. Each class is independent, it has no inheritance links, inherited attributes are duplicated for each class. Inherited methods are included in the subclasses as own resources and renamed at the same time. The delegation of the "super" like calls are not visible in this representation. One can notice that the natural subtyping class relationship between the classes is lost by using such a transformation.

2.1.2.2 Composition

A different approach to transform multiple inheritance into something more simple is to use composition [10]. This approach is based on the fact that if a class needs some services from another class it is either a subclass or it is a client of that class [27]. The transformation consists in transforming all inheritance links in composition links. The former subclass will be composed out of references to instances of the former superclass. Obviously, all "super" like calls have to be delegated to the corresponding component objects. The subtyping relationships between classes is lost also with this kind of transformation. In figure 2.8 it can be noticed that each inheritance link is replaced with a composition link. This transformation is based also on the fact that a class can have an unlimited number of composition links while the inheritance links could be limited to one (in single inheritance based language) or even zero (in the case of procedural languages).

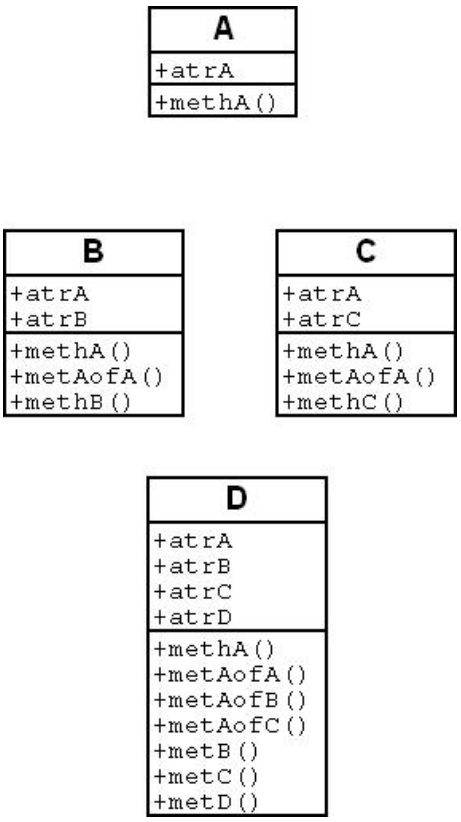

Figure 2.7: Emancipation

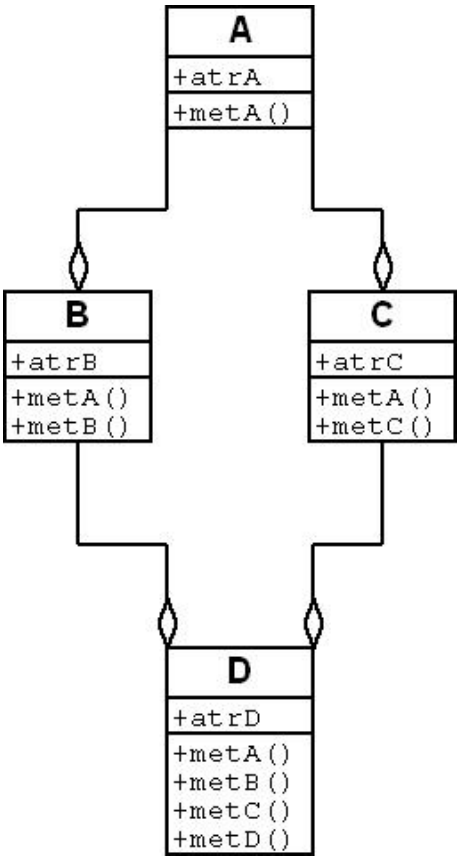

Figure 2.8: Composition

Class D is exhibiting all inherited methods from superclasses implementing them by delegation. The advantage of this transformation is that there will be no code or data duplication, subtyping being lost though.

2.1.2.3 Expansion

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

2.1.2.4 Variant Type

The variant type idea or simulation of variant type comes from procedural programming languages where no polymorphism is available. Simulation is made using a single monitor class which has all the features of all classes and using a flag the different types can be achieved [10]. Depending on the current object type a certain set of features is exhibited. The obtained structure is relatively

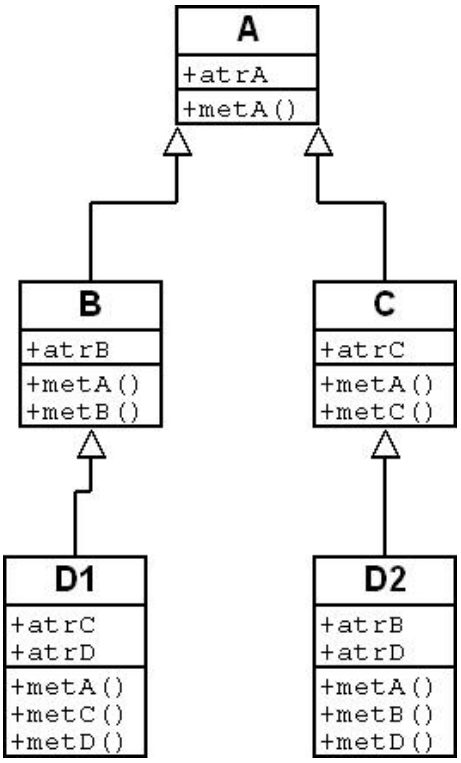

Figure 2.9: Expansion

Example 12 Delegation Sample in C++

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

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

2.1.3 Delegation

As a similar concept with the concept of inheritance, we will discuss about the delegation class relationship in C++ [4, 42]. The idea is that in the base class list of a class declaration there can be specified a pointer to some other class. Example 12 shows such a mechanism.

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

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

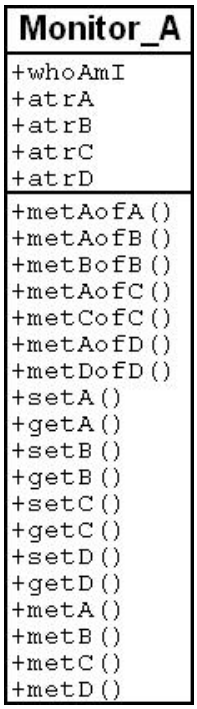

Figure 2.10: Variant Type

Example 13 Delegation Usage in C++

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

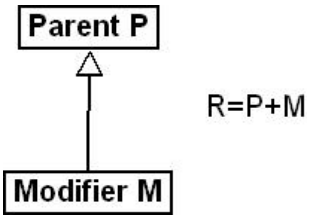

Figure 2.11: Incremental Modification by Inheritance

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

Table 2.1: The Four Incremental Mechanisms

included in the C++ language.

2.2 The "Like-Type" Class Relationship

Incremental modifications can be used in reusing conceptual or physical entities and in the construction of new similar ones [47]. As natural and computational systems evolve there are necessary incremental mechanisms to control this evolution. Inheritance class relationship is a particular kind of incrementation mechanism which transforms the parent P with the help of a modifier M into a result entity R=P+M.

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

Compatibility rules can be set between subclass and superclass [47].

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

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

Signature compatibility guarantees the compatibility between superclass and subclass interfaces.

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

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

In table 2.1 are presented the four incremental mechanisms and their corresponding class relations [47]. It can be noticed that **like** is the most general relation which includes all the rest. Types which are related by the like relationship are called **liketypes**, as subtypes is the name for types related by the subset relation. The subtype relation is asymmetrical while the liketype is symmetrical. If any type R1 like R2 then R2 will be like R1.

2.3 Mixins

Software complexity gave birth to methodologies which divide the problem in solvable parts after they have to be composed in the final software product[38]. The mixin mechanism is based on

Example 14 General Form of Mixin in C++

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

two other concepts: **inheritance** and **genericity**. Mixins are derived from generic programming and they are generic classes which have as generic parameter types their superclasses. The basic idea was to specify an extension without being obliged to specify the unit to be extended. This is equivalent to specifying the subclass letting the superclass as a parameter which will be specified later.

Mixin in C++ is a high power technique for using multiple inheritance with abstract virtual base classes to enable incremental development of both interfaces and implementations of classes [37]. Mixins are considered the greatest achievement in C++ although it was not intended to enable it in the language.

The advantage of this approach is in the fact that there is only one class used for a valid incremental extension specification for a variety of classes. In [46] it is shown how mixins are used to create a role based design. Mixin layers is a derived mechanism from the mixin concept and it was made for concern modeling. They are nested mixins in which the parameter of the external mixin will determine the parameters of the internal mixin [39].

2.3.1 The Mixin Concept

In this subsection we will focus on the implementation of mixins in the C++ programming language. The basic idea is to define an extension without knowing a priori what is extended. This implies the specification of a subclass while the superclass will be specified later using a generic parameter [38]. Mixins can be implemented using parameterized inheritance. The superclass of the mixin will be specified as a parameter which will be specified at the instantiation moment. In C++ such a mechanism can be expressed like in example 14.

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

Example 15 show how the operations of the designed mixin interfere with the operations of the graph modeling class. The counting operations from the mixin in the sample can be applied to all classes having the same interface (see example 16).

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

2.3.2 The Mixin Layer Concept

In this subsection we will focus on the implementation of mixin layers in the C++ programming language. Mixin layers are a particular form of mixins. They are designed to encapsulate refinements for multiple classes [38]. They are nested mixins so the parameters of the external one will determine the parameters for the internal mixin. The general form of a C++ mixin layer is presented in example 17.

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

Example 15 Graph Counting Mixin Sample in C++

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

Example 16 Using Mixins Sample in C++

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

Example 17 General Form of Mixin Layers in C++

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

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

2.4 Traits

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

2.4.1 Motivations

Motivations around this concept attack the weaknesses of the concept of inheritance [36]. First of all inheritance can not factor common features from complex class hierarchies. This gave birth to the multiple inheritance class relationship. Mixins, discussed in subsection 2.3, are a way of composing classes incrementally starting from sets of members. It is admitted in [36] that in practice there are a lot of problems with these mechanisms. One cause are the conflict resolutions when inheriting the same feature on several inheritance path. As it was presented in subsection 2.1 that the solutions involve linearization or renaming which makes the desired behavior hard to achieve. It is stated also that reusable artifact are hard to design without conflicts. Class hierarchies based on inheritance suffer also from the fragile base class problem [29]. Changes in the class hierarchies affect the conflict resolution mechanisms causing anomalies.

2.4.2 Classes and Traits

In [36, 33] is presented a solution to the earlier invoked problems. There is a separation between the concepts of traits and the concept of class. Traits offer a set of services (methods) which implement the behavior but not state (attributes). Traits can depend further on other traits. Traits have no direct access to state, but using accessor methods. In conformance with this model, a class can be built starting from a set of traits and providing the necessary state and the missing services. The missing services represent the linking code which specifies how traits are connected and how possible conflicts have to be solved.

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

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

2.4.3 Composing Traits Use Case

In figure 2.13 is analyzed a sample from [36] of how traits can be composed. Also the case of a conflicting features is considered.

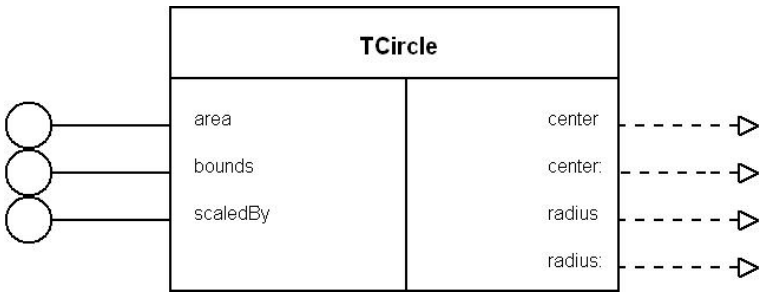

Figure 2.12: Traits Model

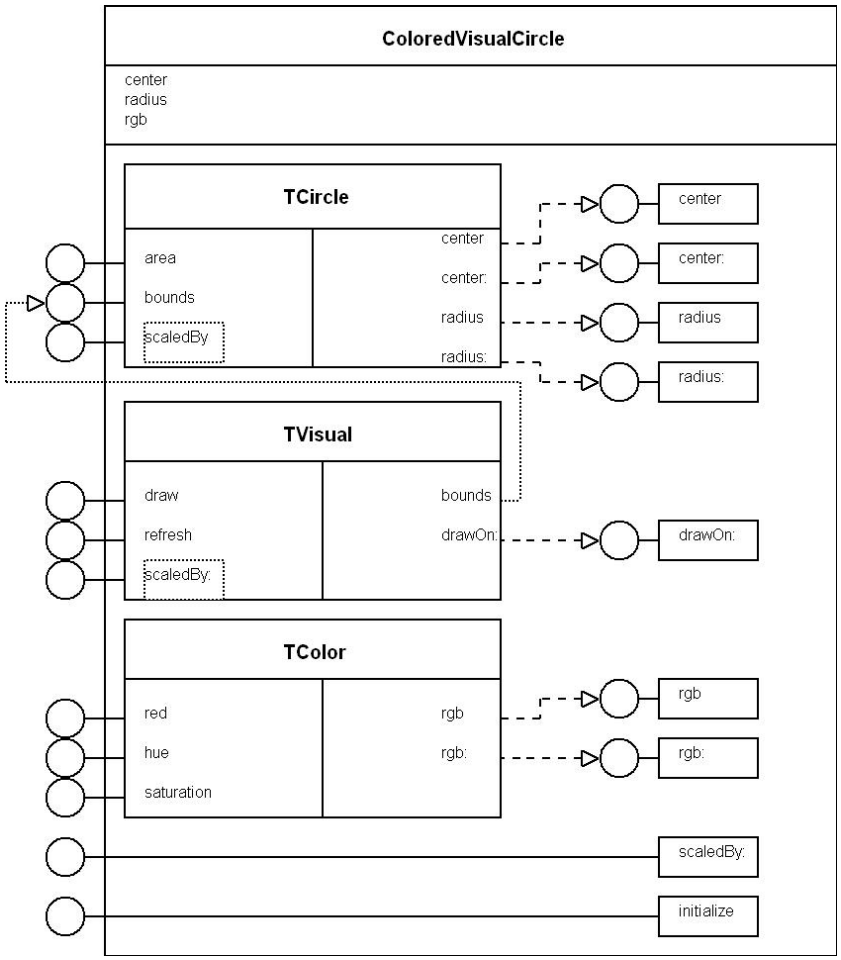

Figure 2.13: Traits Use Case

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

2.4.4 Traits vs. Multiple Inheritance

In this subsection we compare the traits mechanism with another reuse mechanism of the object-oriented technology: multiple inheritance. Traits and inheritance can be combined together in a constructive way. Since there is no inheritance between traits, any "super" like call from one of its methods, is linked to the method of the composing class parent. Comparing the traits mechanism with multiple inheritance it can be admitted that there are some similarities and some differences. The starting point for both mechanisms is the combination of reuse entities. The composing mechanism is semantically the same: feature reunion. It was admitted that features from traits will be incorporated in the composing class. With multiple inheritance which involves subtyping and also subclassing, the same feature composition can be obtained. There can be noticed a difference from the technical point of view. A class in order to be valid is obliged to have all its external calls resolved, while a trait will be connected in the moment of composition. A potential problem of traits model is that it implies developing from scratch all the reuse artifacts, there is no decomposition mechanism for the already existing classes. The "diamond problem" of multiple inheritance analysed in subsection 2.1 in the case of traits model, the authors of [36] claim that since there are no attributes in the structure of traits, there are no conflicts. The possible method conflicts are solved by declaring the conflicted method in the composing class or allows the programmer to favor one mixin to implement a certain service [36].

2.5 Role Programming

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

2.5.1 Roles

Role programming allows the decomposition of the object into several roles. Roles are abstracting the concerns and formalizing their separation. From the collaboration point of view, roles are parts of objects which fulfills their responsibilities in the collaboration [17]. Roles are encountered in many practical situations. Taking the example of a university student, sometimes he can be a football and a basketball player. In some special situation he can became a member in the university council. After a period he can quit this memberships. So objects in object-oriented systems may behave the same way like real-world ones do. Dynamically, during their life time several behaviors can be attached and detached to them [44]. Roles are a volatile concept in the implementation since they do not generally exist as an identifiable component [17]. In [22] the properties of roles are presented. **Abstractivity** facilitates the roles to be organized in hierarchies. **Aggregational composition** is the property of roles which results from the fact that roles can be composed with other roles. **Dependency** property states that a role can not exist without an object. **Dynamicity** refers to the fact that roles can be added or removed during the lifetime of an object. **Identity** of role is the same with the identity of object. **Inheritance** in the context of

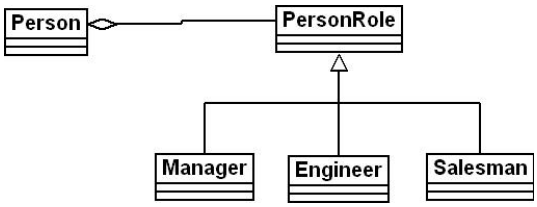

Figure 2.14: Role Object

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

2.5.2 Collaborations

In relation to roles collaborations also have to be discussed. Collaborations involve the cooperation of a group of objects which perform a task or maintain an invariant [17]. The main objectives of roles are to describe the collaboration of objects and to delimit well their boundaries [44]. Objects may be involved in multiple collaborations having different roles. Initially collaborations were described by specifying use cases and observable behavior of the objects. The use case idea originated in [18] and then was adopted by UML [1]. In UML roles were denoting directions of static associations (Association Roles) and afterwards they were related to collaborations (Collaboration Roles). Roles by a set of behavioral functions are able to delimit the boundaries of objects and thus their granularity is smaller, being very close to the level of methods.

2.5.3 Role Implementation Techniques

In the work of [12] are presented several role representations and corresponding design patterns for implementation. A design pattern as presented in [14] is a general solution to a class of software architectural problems. The proposed role representations are: single role type, separate role type, role subtype, role object, role relationship. **Single role type** is a solution where all features of the roles are combined in a single type. **Separate role type** imply that for each role a separate type has to be created. **Role subtype** solution organizes the roles in a hierarchy. Each role has its own type and common behavior can be put in the supertype. **Role object** pattern involves the existence of a host object which has several sub-objects, one for each role. The clients will ask the host object for a specific role feature. In figure 2.14 is presented such a situation. Class *Person* has several roles like manager, engineer and salesman. For this case a role class hierarchy is designed having as superclass *PersonRole*. Class *Person* will have to use one reference of type *PersonRole* which will refer any role instance. The interface of the *PersonRole* with polymorphism and dynamic binding will allow transparent access to different role implementations of the person's behavior.

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

Mixin layers are another way of implementing roles [39]. The model of the mixin layer and its role implementation capabilities were presented in subsection 2.3. In the approach of [17], a

Example 18 Role Implementation

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

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

In [15] in order to allow objectual database evolution, along with the class hierarchy of the objectual paradigm, a role hierarchy is created. A role hierarchy is a tree of special types, named role types. The root of the tree defines the invariant properties of an entity while the roles types reflect refinements. An entity is represented by an instance of the root type and the instances of every role type that the entity has at a given moment. So the traditional object-oriented concepts are extended with the role hierarchy.

In [20] roles are implemented using the AOP of AspectJ and are indicated the relations between the role approach and the aspect-oriented technology. The approach of [44], the role model is created on the base of some principles. One of them is the **support adaptive evolution** which means that objects evolve in environments assuming their roles, the participation can be made dynamically: objects can enter or leave environments freely, an object can belong to multiple environments at a time. The second principle is related to the **separation of concerns**, each concern is modelled by an environment. Concerns will interact using objects simultaneously assuming roles of different collaboration environments. The third principle is the **advanced reuse of roles** within environments. Environments and roles have the status of first block constructs in EpsilonJ proposed programming language.

2.6 Composition Filters

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

2.6.1 Motivations

The concept of composition filters appeared in the context of an object-oriented language database integration model [6]. Motivations in this direction are given: **duality in conception** - language and database models are kept separately, **violation of encapsulation** - object queries makes object structure visible and are not accessed via send messages only as the object-oriented paradigm requires, **fixed views** - relational databases support views on base tables while from the object-oriented point of view there are methods of an object which are not of interest for any client. The integration of database like features in the object-oriented programming language implied the extension of the object model.

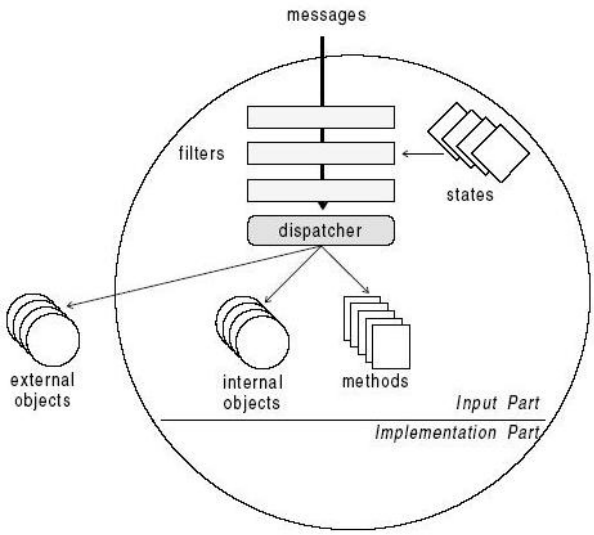

Figure 2.15: Composition Filters Model

2.6.2 The Composition Filters Model

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

Composition filters can be used as mechanism to separate concerns.

2.7 Views

A **view** is a description of the system relative to a set of concerns from a certain point of view [16]. The motivation for **multiple views** is **separation of concerns**. They were introduced to manage the complexity of software engineering artifacts like: requirements, specification and design. UML [1] is the most known language which respects the point of view modeling philosophy.

In the world of object database the notion of view has emerged from the relational view solution. Views [23] in the relational model provide logical data independence and offers possibilities for data to be repartitioned and restructured to fit particular applications. In addition, the database object views offer the introduction of new classes (called virtual) into the class hierarchy [3]. Virtual classes can be populated with objects in several ways: i) the virtual class is a superclass of certain classes (generalization); ii) the virtual class contains all objects returned by a query (specialization); iii) the virtual class contains all objects having a certain behavior.

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

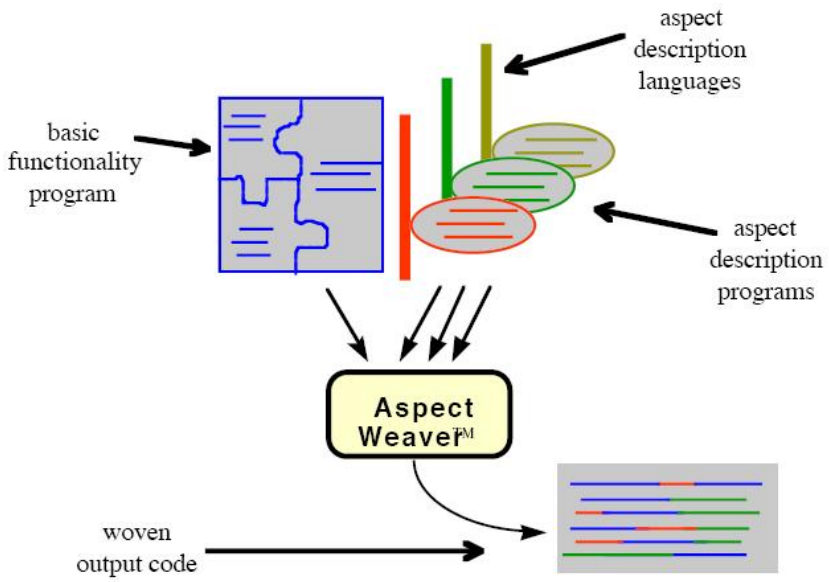

Figure 2.16: Aspect Oriented Programming Main Principle

second level of reuse is exploited in object-oriented databases.

2.8 Aspect Oriented Programming

Aspect oriented programming [21] is a separation of concerns model based on object-oriented paradigm. It deals with crosscutting concerns which can not be well separated by pure object technology. The majority of object-oriented systems are composed out of crosscutting concerns dispersed over several modules. By concern it is meant a concept, a goal in the context of a given domain. For example a concern in the context of debugging a software system would be the logging operations. Another functionality, which can be viewed as a crosscutting concern, needed in the context of objects, is persistence.

There are several concepts of object-oriented programming which facilitate the separation of concerns. First, the abstraction principle implies the creation of separate classes for each concept from the real world [5]. On the other hand the information hiding principle allows interface separation from implementation. Inheritance and delegation are ways of composing behavior. In the context of inheritance, the behavior of the subclass is composed with the behavior of the superclass [5].

In figure 2.16 the main schema of aspect oriented programming is depicted [21]. Each application has a main part where the basic functionality is captured. This part is supposed to be written in a language that suits better to the application domain. Then each cross-cutting aspects are described using several specialized languages. All these programs are taken by the weaver and it produces the output code. The main property of this methodology is aspectual decomposition. Thus, the aspectually decomposed program is easier to develop and to maintain.

Generally speaking, a software system is composed out of several concerns and it is responsible for multiple requirements. In [24] the requirements are classified as **core module level requirements** and **system-level requirements**. The system-level requirements are crosscutting several modules. In example 19 taken from [24] it is presented a class implementing some business logic.

The first observation is that the *other data members* do not belong to the core concern of the class. The *performSomeOperation* method includes along with the core concern, some other operations like logging, authentification, multithread safety, contract validation, cache management.

Example 19 Croscutting Concerns Sample

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

The two operations *load* and *save* having the role of persistence management should not be part of the core concern.

There are two symptoms **code tangling** and **code scattering** which indicate the problematic implementation of crosscutting concerns [24]. Code tangling happens in modules which interact simultaneously with several requirements. Code scattering refers to concerns spread over the software system modules.

AspectJ is an implementation of the aspect oriented paradigm in Java developed by Xerox Parc. The concepts for describing the extensions are **pointcuts**, **join points**, **advices** and **aspects** [45]. Join points are certain well defined points in the execution of the programme. The pointcut is a language construction that specifies several join points. Advice is a piece of code executed when a joint point is reached, it brings together a pointcut and a body of code, to run at each join points. The aspect is a unit of modularity for the crosscutting concern.

2.9 Summary

In this section we talked about inheritance in general and multiple inheritance in particular focusing on crucial aspects. There are opinions that multiple inheritance is bad and dangerous because of the resolution mechanisms which are sensible to any eventual changes of the hierarchy. Other authors consider that the approach of Java [8] is the right way of implementing the concept of multiple inheritance by letting multiple inheritance of types but single inheritance of classes and classes can implement many interfaces. Another important issue is the **resolution mechanism** which could be assisted like in the case of C++ [42, 41] or Eiffel [28] or it could be automatic like in CLOS [19]. In CLOS the class which appears first in the list of superclasses is the one that has priority in the case of conflicts. In Eiffel the conflict resolution decisions can be taken for each conflicting feature separately, while in C++ the "all or nothing" approach is applied: one decision applies to all the features involved. It can be noticed that the Eiffel renaming solution for multiple inheritance name clashes in fact generates the problem of dynamic binding. Such problem do not exist in C++. In C++ to switch between replication and sharing implies placing

the **virtual** keyword at a higher level than on which the effect occurs. It results that such conflict resolutions must be foreseen in advance, in practice this is not always possible. In Eiffel, in the case of dynamic binding problems, the **select** keyword is used in the subclass where the ambiguity arises. Regarding the implementation of multiple inheritance in several programming languages are presented where some semantical implementation decisions can be learned. In each implementation solution there is a balance between several issues like data and code duplication, type conformance, delegation code insertion which work together for maintaining the original semantics of reverse inheritance.

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

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

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

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

Chapter 3

Generalities About Exheritance

3.1 Main Approaches of Reverse Inheritance

The idea of upward inheritance was born in the database world from the concept of database schema generalization [35]. A type corresponding to a database schema may be a generalization of several specialized ones. It is also the case of generalization in a global multi database view which provides a homogeneous interface to a set of heterogeneous databases. The basic idea of reverse inheritance class relation is the **generalization abstraction** [40], which enables a set of individual objects to be thought generically as a single named object. It is considered to be the most important mechanism for conceptualizing the real world. Generalization helps the goal of uniform treatment for objects in models of the real world.

From the development point of view of a software system, direct inheritance is a top-down approach of construction while reverse inheritance offers the possibility of constructing software in a bottom-up manner. We adhere to the idea that it is more natural to first create the subclasses, than to observe and analyze commonalities, and after that to define the super classes [30, 32]. The autonomous design of class hierarchies or database schema will give rise to **inhomogeneities**. Their reusability depends strongly on their capabilities of adapting their local interface to a common global interface.

3.2 Definition

The **reverse inheritance** class relationship is also known as **exheritance** [32], adoption [25], generalization [30, 1] or **upward inheritance** [35]. The source class of reverse inheritance is known as **generalizing class** [32] or as **foster class** [25]. In the state of the art there are several approaches dealing with reverse inheritance issues in domains like object-oriented programming and design, databases, artificial intelligence.

We start from the definition of reverse inheritance given by Pedersen [30, 32] which states that a class G can be defined as a generalization of A_1, A_2, ..., A_n previously defined classes. If the value of n is 1 then we discuss about **single generalization**, otherwise about **multiple generalization**. Informally it can be defined as another model of inheritance where the subclass exists and the superclass is constructed afterwards.

3.3 Intension and Extension of a Class

In [30] is presented a simplification of the object concept. The **intension** of a class is the set of properties through which it is defined. An example is given in this sense. The "mammal" concept is analyzed. The intension of this concept refers to real-world properties like: these animals have mammae which secrets milk as nourishment for their young. By **extension** of a class we mean all

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

Specialization can be defined in terms of intension and extension of a concept. A concept $C_{special}$ is a specialization of a concept C, if all phenomena of $C_{special}^{extension}$ belong to $C^{extension}$ [30]. Concept worker is a single specialization of concept employee, since all workers have all properties of employees and eventually some extra. A worker can take the place of an employee but not necessarily the other way around. Formally this can be expressed like: a concept $C_{special}$ is a single specialization of a concept C iff $x \in C_{special}^{extension} \Rightarrow x \in C^{extension}$. There can be defined also the notion of **multiple specialization** in the same way: a concept is a multiple specialization of a set of other concepts if it is a single specialization of each concept in the set [30]. Concept calculator-watch is a specialization of both concepts calculator and watch. Calculator-watch fulfils the properties of calculator and watch. Formally, a concept $C_{special}$ is a multiple specialization of $C_1, ..., C_n$ iff $x \in C_{special}^{extension} \Rightarrow \forall i \in 1..n : x \in C_i^{extension}$[30].

Generalization can be defined also in terms of intension and extension of a concept [30]: a concept $C_{general}$ is a single generalization of a concept C if all members of $C^{extension}$are members also in $C_{general}^{extension}$. This means that all phenomena belonging to $C^{extension}$ will belong also to $C_{general}^{extension}$. Concept employee is a generalization of concept worker since every worker is an employee. Formally $C_{general}$ is a generalization of concept C iff $x \in C^{extension} \Rightarrow x \in C_{general}^{extension}$. As in the case of specialization there is **multiple generalization**. A concept is a multiple generalization of a set of other concepts if it is a single generalization of every concept in the set. For example the concept of employee is a generalization of worker, manager, security guard, secretary, because all are employees. In formal notation $C_{general}$ is a generalization of $C_1, ..., C_n$ iff $\forall i \in 1..n, x \in C_i^{extension} \Rightarrow x \in C_{general}^{extension}$.

3.4 Semantical Elements of Reverse Inheritance

It is proposed in [32] the idea that reverse inheritance should have an appropriate symmetrical semantics in order to produce the same class hierarchy structure having the behavior as if it was defined by direct inheritance. So, this class will include all the features (attributes and methods) that are **common** to these classes. Also it can be specified by the programmer which features should be excluded from exheritance.

In [25] two rules are set for defining the semantics of reverse inheritance class relationship: one sets the type conformance between subclasses and superclasses and the other defines the class dependency which is oriented from superclass to subclass. The subclasses will conform to the types of the newly designed superclass and the newly created superclass depends upon the subclasses. Of course, the two rules are not sufficient and a set of restrictions are also defined to complete the definition. These will be presented further.

As we already know from normal inheritance, subclasses depend and conform to their superclasses. So both **dependency** and **type conformance** have the same direction from subclass to superclass. In the analysis made in [25], the reverse inheritance concept changes their directions: the type dependency remains in the same direction, but the dependency is now oriented from superclass to subclass.

As presented in the Unified Modeling Language description document [1], which is considered the standard modeling language for the object-oriented development process, the generalization relationship can be applied to several model elements like classes, associations, stereotypes, actors. By definition, the role of generalization is to relate a more **general** element and a more **specific** element, so the instance of the specific classifier is also an instance of the general classifier.

In the work of [31] an analysis is made on relation of generalization with other UML elements and two aspects are emphasized: an incremental one and an overriding one. The former goes along with classes and the latter fits to associations, stereotypes, signals, use cases, actors. The incremental aspect refers to subclasses which have a richer set of messages in their interface than

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

their parents. Overriding happens when two methods are created, one in the superclass and one in the subclass, denoting the same message, but having different parameter and return types or different behavior.

We can conclude that all semantical definitions include either the idea of intension intersection or extension reunion of the generalized concepts. Next we will focus on the programming languages.

3.5 Reuse of Object vs. Reuse of Class

In this section we will refer to different situations of reuse object and class.

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

Some simple solutions seem to solve the uniformity problem. One is to make local changes in the classes [35]. The disastrous consequence is a dramatic chain of modifications in the clients, which will trigger a new cycle of software development [27].

Another solution is to create new views for these objects. This will determine an explosion of variants of the original ones which can differ slightly [35, 27]. So we created a huge configuration management problem.

Another possible solution is to create a union of the generalized class objects [35]. The common messages of the generalized classes can be received uniformly by all subclass instances. This solution involves inconsistencies when a foreign message is sent to an object which cannot execute it, but it works immediately only for the features having the same name.

In [35] are discussed the several semantical relationships between classes: identity relationship, role relationship, history relationship, counterpart relationship, category relationship. There are investigated situations when two objects are equivalent: one possibility is for their classes to model the same real-world object or their classes can model real world-objects which have some common properties. The **identity relationship** between two classes stands when the real-world objects modelled by those two classes are identical at all points of time. The **role relationship** occurs when two objects may model the same real world object in different situations or context. For example a person could be at the same time university employee and company employee. This person has two different roles which could be modelled by different classes. These classes are an example of role related classes. The **history relationship** between classes occurs when these classes model a real-world object at two different real-world times. The **counterpart relationship** hold between two non-equivalent objects which represent two different real-world objects. It is necessary for them to have some common properties and to represent alternate situations in the real world. For example two classes modeling air connections and train connections are counterpart related because both have properties like departure city, destination city and fare. The **category relationship** holds between objects which share some common properties. Two classes modeling coal plants and oil plants are category related because both share plant common related properties.

3.6 Explicit vs. Implicit Declaration of Common Features

We think that the specification of the excluded features in the definition of the reverse inheritance relationship between two classes has a drawback, affecting clarity. If one wants to develop further a reverse inheritance based class hierarchy, he has to know the list of all the common features from exherited classes. Instead it would be better to have a list of them explicitly declared in the foster class. The explicit list of features in the foster class will be of much more help to the programmer, for example in the definition of a subclass derived directly from the foster class. One more argument to sustain the affirmation are the possible adaptations to be declared around common methods having incompatible signatures. So the syntax will be easier. The exheritance

concept comports two essential aspects: interface[2] exheritance and implementation exheritance. Each aspect will be detailed in the next chapters.

3.7 Allowing Empty Class

In [32] the case of no exherited features is discussed. It can be useful in languages where there is no default superclass for all the user defined classes. This practice fits better to dynamically typed languages. It involves mechanisms of runtime typechecking and casting operators. For instance, in Java there is a class *Object* which is the **absolute superclass of all classes**. In the definition of a new class, the relation with class *Object* is not explicit, it is implied automatically by the semantics of the language. In Java we find two main types of classifiers[3]: classes and interfaces. Analyzing the problem, we draw the conclusion that Java presents an asymmetrical semantics regarding interfaces: there is no primordial superinterface. The problem of lacking a super interface in Java could be solved with the help of reverse inheritance. It is the same for Eiffel language, having an absolute super class named "ANY"[4].

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

3.8 Source Code Availability

One important issue about reverse inheritance is the source code of generalized classes [32]. There are several situations that have to be discussed. The most favorable situation is when source code is available. This gives many choices in the implementation of a such a class relationship. We can imagine an implementation by modifying original source code and generating equivalent source code, compilable by the Java/Eiffel/C++ compiler. Another situation is when source code is available but it is read-only. It cannot be modified because of many reasons: copyright policy, increased effort for maintenance. In this case it can be generated equivalent source code using decoupling techniques to protect the original sources. The most problematic case is the one where no sources are available, just the interface and the binaries of classes. In such case byte code modification techniques should be applied.

3.9 Single/Multiple Exheritance

Single exheritance is the most simple case of exheritance. It involves only one target class to be exherited. It can be exherited both interface and implementation. In [30] an example of a double ended queue is given to emphasize the semantics of single generalization. It is started from a class named *Dequeue* having a set of methods which operates at one end of the dequeue *pop*, *push*, *top*; methods for operating at the other end of the dequeue *pop2*, *push2*, and a separate method *empty*.

From the dequeue it is created a stack. Class *Stack* is declared as generalizing *Dequeue* and excluding operations *push2*, *pop2*, *top2* from its interface. From the point of view of interface exheritance there are no conflicts. Name conflicts obviously cannot occur, because initially the generalizing class has no features. Some name changes could be necessary to give a more suggestive meaning to methods. For instance if we generalize a class *Collection* from class *Stack* instead of

[2]By interface we denote the set of public features in a class. It is different from the concept of Java interface, which technically is a pure abstract class [8].

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

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

Figure 3.1: Dequeue Sample

exheriting the *push* method with the original name we should better exherit it with another name, like *add* for example.

3.10 Summary

This chapter encloses all general ideas about exheritance and its semantical elements. First the motivation is presented, the exheritance concept being found in object-oriented databases and in class modeling of the object-oriented paradigm. Formal definitions of single/multiple specialization/generalization, based on the notion of the intension and extension of a concept, are given. Some exheritance basic semantical elements are discussed: common features, type conformance, generalization and specialization. The reverse inheritance concept can be located at the intersection of object-oriented programming and object-oriented databases. Each domain has its own interest of reuse: class reuse, respectively object reuse. Several contexts are presented in which reverse inheritance is the main mean in achieving reuse. The issue of allowing the source class of exheritance as empty is analysed in the context of several programming languages. Then the problem of having available the source code of the reverse inheritance affected classes is discussed. This issue belongs more to the implementation part where such decisions have to be discussed. The decision to include the discussion in this chapter is to suggest the several implementation semantics possibilities.

Chapter 4

Interface Exheritance

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

4.1 Concrete vs. Abstract Generalizing Classes

In [30] it is emphasized that this aspect of exheritance is the most simple. As mentioned in [32], the integration of interface exheritance in Java can be done with minimum of effort because of the notion of "interface" they introduced in the language. A Java interface consists in a set of abstract methods [8]. It can be considered as a pure abstract class. An abstract class in Java may contain abstract methods having no implementation, just signature and also concrete methods with implementation. We note also that interfaces can be created by specialization of several multiple interfaces, they can be implemented by several subclasses and their methods are all public. It is suggested that interfaces could be defined by generalization of classes and other interfaces.

Not all languages possess such an interface concept like Java does, so we have to use the class concept as generalization classifier. For those languages is proposed [32] the idea of generalization into fully abstract classes (e.g. Eiffel, C++, Java).

4.2 The Influence of Modifiers on Exherited Features

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

In order to answer this question we have to analyze what happens if we exherit from non-public methods. The positive reasons are: they can be better reused from the level of the generalized class and they can be inherited later. Negative reasons are class encapsulation violation which implies visibility modification: clients may be affected, exposed methods can be overridden. So, potential problems may be introduced at the implementation level too[1]. In [32] it is advised not to distinguish between public and non-public in the exheritance of method interfaces.

It is mentioned in [32] that attribute exheritance does not involve big problems. Though, some type and visibility problems may occur. Here we can take two cases: public and non-public attributes. Although in the literature the use of public attributes it is not encouraged, still some programming languages allow their use like C++, Java. In Eiffel attributes can be exported but they can not be modified, only read, because the access to an attribute involves an execution of a query which provides the desired result.

We will analyze from the conceptual point of view each type of modifier encountered in common object languages. The problem of visibility consists in finding the appropriate modifiers for the

[1] Implementations issues are not the subject of this analysis, they will be discussed in another work.

exherited features in the superclass. The main idea is that we want to preserve or to affect as little as possible the feature's visibility.

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

If we deal with **package** access type[2] modifiers in the subclasses then it means that features have to be available in their package. We have to discuss several cases:

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

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

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

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

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

4.3 Status of Original Methods: Abstract/Concrete

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

If we decide that we have a sufficiently general implementation we can put it in the superclass without affecting the eventually abstract subclasses. These ideas are exemplified in two samples one Java, the other C++ (See algorithms 20 and 21).

In the two samples class *StaffMember* is abstract: in Java because of the abstract modifier, moreover it has an abstract method named *print()*; in C++ because of the virtual and equals zero *print()=0* method declared. Class *Employee* generalizes two classes in the parallel hierarchies one abstract and one concrete: *StaffMember* and *SecurityAgent*. The *print* method which equips all the classes it could be either abstract, either concrete. If we posses a general enough implementation that will fit to all subclasses from now on, then we can put it in the *Employee* class. Conversely, we declare the print method as abstract, obviously leaving it without implementation. In either cases subclasses like *TeachingAssistant* or *Professor* can override the print method with appropriate behavior. If we create the same class hierarchy but using inheritance this time, then we will find the same semantical behavior regarding the status of the print method.

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

When **concrete methods** are exherited there is the possibility to exherit just the interface or together interface and implementation. This choice should be available to the programmer who uses the reverse inheritance class relation [32]. If it is chosen not to exherit implementation, then

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

Example 20 Examples in Java

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

Example 21 Examples in C++

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

in the foster class just the corresponding abstract method can be specified. The aspects dealing with implementation exheritance will be discussed later, in a specially dedicated chapter.

4.4 Type Conformance Between Superclass/Subclass

Related to interface exheritance issue, in [30] it is demonstrated using an experimental language that from the type conformance point of view, there are **no conflicts** introduced in a class hierarchy having subclasses/superclasses introduced by inheritance/reverse inheritance. The main idea of the demonstration is to prove using formalisms that the feature set of the generalizing class contains at most the intersection of the feature subclasses sets.

Before proof, generally speaking, some notations are necessary:

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

denotes the set of methods of class C.

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

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

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

In the [25] definition of semantics a type conformance rule is set. The type of subclasses have to conform to the type of superclass. From their point of view the superclass type is a generalization of the subclasses types. It can imply type intersection or type union, depending on the type definition. In section 3.3 we discussed about the intension and the extension of an object. Referring to these two conceptual aspects of an object they consider that if a type is a set of features than the type of the superclass should be their **intersection**. If the type is considered as a set of objects, then the superclass type of the generalizing class will be a least the **union** of the subclass types.

4.5 Common Features and Assertions

In this paragraph we discuss ideas from state of the art regarding how common features are defined. An attempt in this direction is made in [25] and two restrictions are set forth: i) common features are those who have same name, ii) it is possible to define a common signature to which all signatures from the subclasses conform.

We present also some ideas of how preconditions, postconditions and invariants are affected by reverse inheritance. We remind that predicates are the main concepts around which the **Design by Contract** technique was built [27]. The purpose was to offer to the programmer tools to express and validate correctness of a program. The relation between a class and its clients may be viewed as a formal agreement expressing rights and obligations for each of the parties.

A **precondition** states all the predicates that have to check when a routine is called. **Postconditions** are predicates which verifies the properties that must hold when a routine returns [27].

In [25] it is stated that assertions rules defined in [27] and [28] should be reversed. The precondition for a feature in the foster class should imply all the preconditions in the generalized subclasses. The postcondition of a feature in the superclass should be no stronger than any of the corespondent preconditions from exherited classes.

There are also mentioned possibilities of building the precondition and postcondition for the features in the superclass. For example the precondition could be the application of the *AND* logical operator against all the preconditions from exherited subclasses. The result will be definitely

Example 22 Name Conflicts (1)

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

a stronger precondition. The postcondition could be obtain in the same manner using logical operator *OR* against all the corresponding postconditions in subclasses. This approach assumes that all variables involved in superclass predicates are defined in each subclass. Otherwise, for each subclass missing variable we could choose not to evaluate the respective term. In other words we could propose to evaluate the predicates only for the classes which have all the variables defined.

As a conclusion regarding assertions in [25] in the definition of common features has to be mentioned that they depend on the possibility to define a precondition other than False, which is no weaker than the precondition of the feature in each class.

4.6 Possible Conflicts

4.6.1 Name Conflicts

In [35, 30, 25, 32] name conflicts are discussed. They occur when two methods have the same semantics but have different names. This conflict is named **lost friends** in [32]. This kind of conflict can not be detected automatically, so this must be set by the programmer. A supporting syntax is suggested to be used in order to indicate which methods should be exherited and which method name to be kept.

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

In [35] the problem of name conflicts is discussed in object-oriented database schema. The conflict takes place between the local and global interface of an object. Local interfaces refer to the original interface of the object, which the object was designed with, while global interfaces denote the common set of messages. The same semantical messages have different names in the two interfaces.

The two classes presented in the sample taken from [35] model two kinds of plants oil and coal. Each of them has a name property, produced energy property, starting method, stopping method and charging method. All these features are common to the two classes, some of them have the same name, like *PlantName* or *PowerOff*, and others have different names, although they have the same semantics, like *FireOn* and *Start*. In this case name conflict situation appears.

Example 23 Name Conflicts (2)

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

Example 24 Name Conflicts (3)

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

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

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

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

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

In [25] the renaming facility from Eiffel [28] it is used in solving name clashes (see example 24).

It is motivated that semantically equivalent features developed in different classes by different programmers will have different names. In the example of example 24, boundary feature from BOX and circumference feature from $CIRCLE$ have the same semantical value, and they are mapped to a unique name.

Example 25 Scale Conflicts

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

4.6.2 Value Conflicts

Value conflicts are encountered when features with different semantics have identical names. They are referred in [32] as **false friends** conflicts. Implicitly it is suggested that features should not be exherited and such situations should be specified by the programmer. They can not be automatically detected and a syntax support for conflict declaration is needed. It is suggested not to exherit such features because they are not the same.

There are cases when name conflicts can be detected automatically. Two classes having a same ancestor can have renamed methods using renaming techniques like those in Eiffel. Both kind of conflicts can happen [32]. In the case of lost friends conflict, the features seem to have the same seed but have different names because of renaming. A solution at compiler level is given: they should be organized as the same feature by the compiler.

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

4.6.3 Scale Conflicts

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

It is discussed in [35] on a sample presented by us in section 4.6.1 how this kind of conflict can be eliminated. We remind the reader that the common attribute named *OilFired* is expressed in barrels of oil while Consumed attribute is expressed in tons of coal. In the generalizing class the desired member should have the same name and the same scale, meaning *MegaJoule*.

In their work [35], in order to solve the problem of scale differences, they proposed the concept of **object transformation**. This concept is implemented with the help of conversion classes. These classes contain a set of methods representing the necessary conversion protocol between several scales. The main idea of the solution is to switch between the **local and global representation** of an object. The local representation uses one scale, while the global one uses another. A conversion class that solves the presented problem can like the one presented in example 25.

In example 25 class *MJOULE* encapsulates all the transformation routines between barrels, tons and *MJOULE*. This transformation methods represent the adaptation behavior from a scale to another. To be more explicit they adapt values. This technique as it is presented in the sample can be applied only to attribute instances which are values or maybe to methods which return values. We think that with the help of some modifications this could be also applied to methods not just attributes. In our opinion the place of such an adaptation behavior code should be in the generalizing class, close to the adapted feature, because of clarity reasons.

Example 26 Parameter Order Conflicts

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

Example 27 Parameter Number Conflict

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

4.6.4 Parameter Conflicts

4.6.4.1 Parameter Order

In [25] another kind of conflict is emphasized briefly. In exherited methods the order of parameter may vary. This conflict can be solved by a **mechanism for binding arguments dynamically**. For Eiffel, an appropriate syntax it is proposed:

In example 26 it is specified the mapping between the method *zoom(center:POINT,factor:REAL)* and method *scale(factor:REAL,center:POINT)*. We notice that besides renaming, parameter order is set also. In order to perform such parameter reorganization, the number of parameters in all the exherited methods and in the superclass must be all equal.

4.6.4.2 Parameter Number

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

As a remark the position of the default parameters is always after the non-implicit ones. Also

Example 28 Parameter Type Conflicts (1)

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

the order of the default parameter is important, because you cannot use implicit *pi* parameter unless you specify first an actual parameter for *t* in *foo* method of class *C*.

4.6.4.3 Parameter Type

The type of parameters can cause problems in exherited features. This kind of conflict has similarities with the one discussed in section 4.6.3. The problem is how can we unify two parameters having different types. First of all we start our parameter type analysis with a small discussion about the **parameter transmission mechanisms** encountered in Java, C++, Eiffel object-oriented programming languages. So in Java we have value transmission mechanism for primitives and object references. The value of the actual parameter is copied into the formal parameter. Changes on primitive typed formal parameters will not affect the actual parameters. When dealing with object references, they cannot be affected, but the referred objects, obviously, can be modified.

In C++ we have types like primitives, objects, pointers and references. The transmission of primitives it is the same like in Java. What is interesting in C++ is the implicit call of the copy constructor[3] when transferring object value parameters [41]. The situation in Eiffel [28] regarding parameter transmission is the same like in Java. There are transferred primitive values and reference values.

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

On example 28 we can discuss covariance issues. Method foo in class *Child* is a redefinition of method *foo* in class *Parent* if: i) the language is covariant and B is a subtype of A; ii) the language is contravariant and B is a supertype of A; iii) the language is nonvariant and A and B represent the same class.

One situation is when we deal with primitives types in object-oriented languages supporting them[4]. If we deal with compatible types we could perform **type casting** implicitly like in the example 29.

In our experiment (see example 29) we have two methods *setX* with two different argument types: *int* in the superclass and *double* in the subclass. For instance if we replace in C++ or Java a parameter's *int* type with a *long* type, arithmetical computations are not affected. But if bit level operations are executed, the result will not be the same. The conclusion is that primitive type substitution implies potential risk to the affected code, a runtime casting mechanism, which

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

[4]It is know that in the object-oriented paradigm the type system should be uniform, so every object should be instance of a class [27].

Example 29 Parameter Type Conflicts (2)

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

can sometimes affect values (e.g. precision loss) and knowing the compatibility rules between the primitives. The most radical solution that can be applied is not to allow the feature exheritance, unless parameters have the same type.

4.7 Summary

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

Chapter 5

Implementation Exheritance

5.1 Impact of Polymorphism in the Generalization Source Class

In the state of the art there are studied two different situations: when exherited methods in the superclass are virtual or non-virtual. When there are no virtual methods in the foster class in [30] there are proposed two unsatisfactory solutions.

Principal Subclass Implementation Pedersen [30] proposes that one of the generalized classes should be chosen as main subclass. It is also motivated that the choice should be made by the programmer because he knows better the implementations of subclasses and because in some languages interface inheritance means also implementation inheritance.

New Implementation The other proposed possibility to deal with exherited implementations is to provide a new implementation in the foster class. Both solutions proposed by Pedersen [30] are criticized in [32] because the semantics of generalization is broken. Because methods are non-virtual or non-polymorphic, the specialized behavior of all subclasses become unusable. In some special cases such a class construction can be useful. One of them is presented in the next paragraph.

Non-Virtual Methods Can Be Useful Sometimes If exherited methods in superclasses are not virtual there is no problem from technical point of view (one can build such a class hierarchy without compiler errors), but it does not express the desired semantics of generalization. Exceptions may occur in the situations where a more general implementation is available and the specialized one from the subclasses can be overridden without affecting the consistency of the class hierarchy, like in example 30.

Generally speaking a virtual method is a method that has a polymorphic behavior. Depending on the type of the object at runtime, a polymorphic method call may exhibit different behaviors. Here we refer only to the dynamic linking component of polymorphism. No matter if the original method is abstract or concrete, generally, the exherited method specified in the foster class should be made virtual [32]. It is known that in any language virtual methods can be overridden in the subclasses.

Empty Method The first solution discussed in [30] is to equip the exherited method in the foster class with empty body in the case of no common behavior. This method could be easily overridden by specialized subclasses. In languages like Java we could define it concrete with empty body, except triggering an exception in case someone calls this method, or abstract with no body, meaning that all the concrete subclasses are obliged to implement it. In Eiffel we can declare it

Example 30 Impact of Polymorphism

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

in the superclass as deferred method. In C++ we have two possibilities by declaring it abstract or as pure virtual. A pure virtual method is a method declared with the "virtual modifier" and having also "=0" suffix meaning that it is a pure method.

Main Class Behavior The second proposition and the default one made in [30] is to exherit implementation from the selected main subclass, when all implementations exhibit the same behavior. In practice it seems a rare case that a set of classes to be equipped with exactly the same body.

Some Common Behavior The idea promoted in this case by [30] is a manual selection of the common behavior from subclasses and the definition of a new method in the generalizing class. This is motivated due to the fact that later on this implementation could be inherited in other subclasses of the generalizing class.

Selective Method Exheritance In the case of virtual methods in foster class, [32] proposes an alternative solution to method body exheritance. It is encouraged the idea of exheriting methods from different subclasses. The solution given seems more flexible and attractive. We think that some adaptations are still necessary to provide a more advanced degree of class reuse.

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

The two classes in example 31 have the same precondition and postcondition for method *increment()*, but different behaviors.

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

In the proposed sample we can find two classes modeling Alcatel and Nokia phones. We need to treat these classes in a uniform manner, so we decided to create a foster class generalizing

Example 31 Selective Method Exheritance

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

Example 32 Adaptive Approach

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

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

them. There can be noticed that the two classes have implemented ringing behavior[1]. It is quite natural that the new created class, baptized *GeneralizedPhone*, to exhibit the ringing behavior. So we exherit the signature of method *ring()* from both classes. We suppose that each subclass has specific ringing behavior. We need to reuse this behavior but in a different way. For example we would like to add actions before and after the each ringing operation. This can be done with the help of a descendant access. It can be imagined an implementation of method *ring()* having a call to the *pre_ring()* operations, a call to the original specific method from the subclass and a call to the *post_ring()* operations. The main drawback of this approach is that the semantics of the original classes may be changed by adding the new features.

5.2 Adding New Behavior

Referring to the example 32 we showed that new behavior added into the foster class can be useful. The *pre_ring()* and *post_ring()* methods added to the *GeneralizedPhone* foster class were used to create an enhanced ringing method. Other advantages of this feature will be presented in the section dealing with the mixing of inheritance with reverse inheritance. Still the classes are in danger of having the semantics changed. Such a capability would be a valid option in a potential reengineering tool based on the concept of reverse inheritance, but not in the programming language semantics.

5.3 Exheriting Dependencies Problem

In [32] the problem of dependency exheritance is mentioned. It is stated that exherited methods dependencies have to be exherited too in order to be usable in the foster class. There are two possible solutions for this problem. One would be to exherit the dependencies as well. In practice this would mean to exherit almost all the features in a class. The other approach is to provide the missing dependencies in the class where the exherited methods were exported. Such an idea is used in the traits mechanism presented in section 2.4.

5.4 Type Invariants Assumptions

It is noted in [32] that type assumptions executed in the context of an exherited method in the generalizing class can be broken. This can happen not only in assertions but in the code of a regular method. Example 33 presents such a situation. The type assumptions, investigated in the condition of the *if* instruction, were designed to work in the context of the original class. The first branch of the *if* instruction is taken. If the code of method f would migrate in a potential generalizing class A, then the f method semantics would change. In this case the second branch of *if* would be taken.

From this sample results that a use of the reflection mechanism in the implementation of a method would make exheritance impossible. Related to this problems the solution would be to choose not to exherit the methods having such assumptions. In order to do so, these assumptions have to be detected first, so a dedicated technique would be required.

5.5 Summary

In this chapter implementation exheritance problems were discussed. The most severe problem is the one of polymorphism impact on the generalizing class. When there is no polymorphism implementation exheritance is problematic. In the case of polymorphic methods there is no problem since they can be very easily overridden in the subclass. The adding of new behavior in the

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

Example 33 Type Invariant Assumptions

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

generalizing class is studied in the context of exheritance. This capability is a disabled options since the semantics of reverse inheritance do not include feature inheritance and the semantics of exherited classes is changed, which is a severe aspect. Then dependencies problems are analysed when implementations are exherited and two solutions are discussed. A special aspect related to type verifications is discussed in the context of implementation exheritance.

Chapter 6

Mixing Inheritance With Exheritance

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

6.1 Fork-Join Inheritance

We begin with the analysis of a conceptual sample from the state of the art. In [32] is presented a case of fork-join like inheritance scheme. There are considered two cases: i) class A is defined then classes C and D are defined as inheriting from A. Class B is defined by generalization of classes C and D; ii) classes A and B are defined first, then class C is defined as a generalization of the two. Class D finally inherits multiply from A and B.

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

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

In case 2 the features of class C are propagated via multiple reverse inheritance to classes A and B, then they are inherited directly into class D by normal multiple inheritance. A subset[1] from class C message set are transferred to A and B and from there, the subset is included in the class D message set by the multiple inheritance. Again, there is no subtype relation between C and D.

In [32] two particular sub-cases are analyzed: if there are no features excluded in the generalization and more if there are no features added in inheritance. In the first sub-case features in class B will have all the common features of C and D obtained by inheritance from A but it will also have specific exherited features. If both generalization and specialization have not excluded or added new features it is created an effect of **cloning** class A in class B of case 1 or class C in class D of case 2.

6.2 Reusing Common Behavior

Another interesting idea is presented in [30] about how common behavior resulted using reverse inheritance, can be reused. For exemplification, a real world case is analyzed dealing with terminals. Given two classes *Terminal1* and *Terminal2*, which model two different terminals, the

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

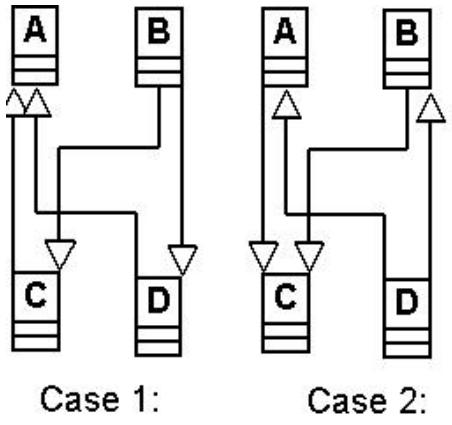

Figure 6.1: Fork-Join Inheritance Sample

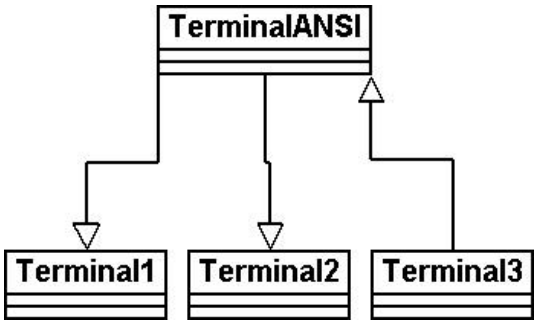

Figure 6.2: Terminal Sample

decision of creating a more general class of the two is taken, in order to group their commonalities for reusing purposes.

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

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

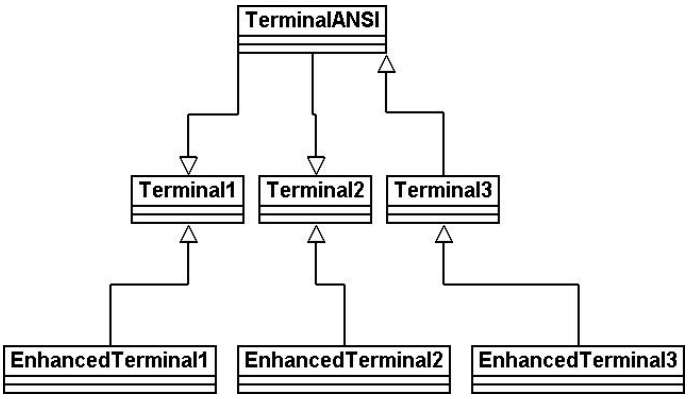

Figure 6.3: Terminal Enhancement (1)

6.2.1 Specialization - The Classic Solution

One possibility is to create three subclasses for each terminal class using direct inheritance and to equip them with the enhancing behavior. The class hierarchy describing this solution is presented in figure 6.3. The three new added classes are: *EnhancedTerminal1*, *EnhancedTerminal2* and *EnhancedTerminal3*. So we created three more classes in the system which will contain the enhancements. There are two possibilities for the enhancements to be contained: either are copied directly in all the subclasses or are encapsulated in a new class, and a member of this type will be declared in all the three subclasses. The first possibility involves code duplication while the second one implies the creation of a new class and the usage of composition. The second possibility can be used only if the enhancements are the same for all the three terminals, still the code dealing with the manipulation of the component object is duplicated.

6.2.2 Feature Adding in Foster Class

Another possibility is to put directly the enhancements directly in the foster class *TerminalANSI* and then they will be automatically inherited in all the subclasses. In other words this means foster class revision creation. This solution addresses the "fragile base class problem" [29]. This type of problem appears when acceptable revisions of the base classes are created which damage the extensions. In [29] this problem is viewed as a flexibility problem and restrictions can be set in order to discipline inheritance. On the other hand, a conceptual drawback is noticed because reverse inheritance should not imply feature inheritance but only feature exheritance.

6.2.3 Setting Superclass for Foster Class

An alternative to the previous solution is to create a new foster class having as target the *TerminalANSI* class and containing the enhancing behavior discussed earlier. So, we don't have to touch the base class of the hierarchy, this solution could be used in situations where source code is not available or no class maintenance responsibilities are accepted. The idea of adding features to a base class using reverse inheritance still suffers from semantical breaking and contamination with fragility. In figure 6.4 such a situation is depicted.

6.3 Dynamic Binding Problems

It is demonstrated in [30] that multiple generalization conflicts are the same as for multiple inheritance thus the solution to the second problem could be applied to the first. Conflict resolutions in multiple inheritance were analyzed in section 2.1.1. In [25] a similar problem of accessing a unified

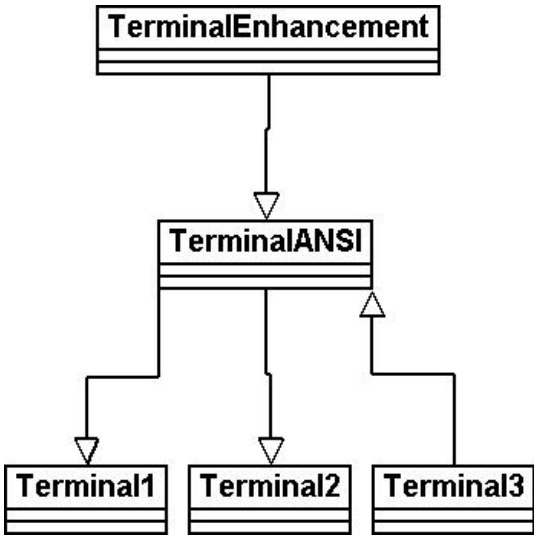

Figure 6.4: Terminal Enhancement (2)

feature, which is multiply inherited, is discussed. In this case the multiple inheritance mechanism for Eiffel is used and it is combined with renaming.

In example 34 which was presented in a different manner in section 4.6.1 it is now created a class that derives from both *BOX* and *CIRCLE*. The problem is which implementation will be used when having a reference to class *SHAPE* and *perimeter* feature is called. This is an ambiguous situation which has the following alternatives: boundary from class *BOX* or circumference from class *CIRCLE*. A proposal is made in [25] to take as implementing feature the one present in the first declared subclass. In our case it is boundary from class *BOX*, because class *BOX* is the first in the inheriting list of class *CIRCULAR_BOX*. This approach is wrong because the selection method does not allow any configuration of selection in more complex class hierarchies. Such an example is given in figure 6.5 where classes A, B, C were created first and have features x, y, z. Then the generalizing class F is created having A, B, C as subclasses. Later on the AB, BC and AC classes are created having the first superclass in the list the class denoted by the first letter in their name. So for class AB the first superclass is A while B is the second. In this configuration if one wants to select feature x for AB, feature y for BC and feature z for AC it is not possible. The first two choices are valid while the third one is not possible. The order selection solutions does not work in all cases.

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

6.4 Architectural Restrictions

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

Example 34 Exheritance Dynamic Binding Problem

```
class BOX
feature
  draw is do  end
  height, width, area, boundary: REAL

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

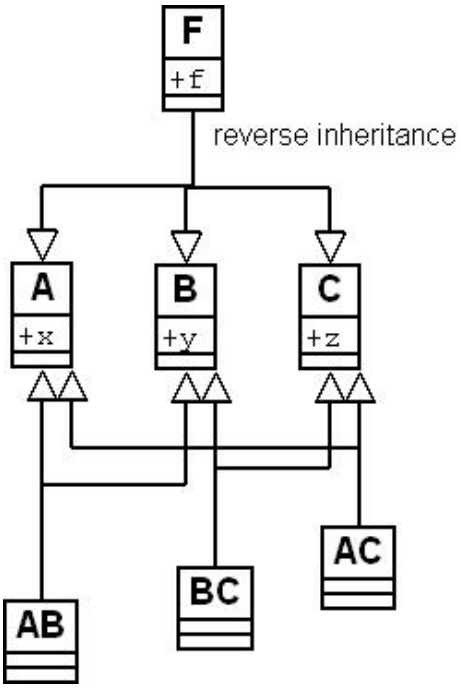

Figure 6.5: Exheritance Dynamic Binding Solution

6.5 Summary

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

Chapter 7

Conclusions and Future Work

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

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

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

In Eiffel the implementation of such a concept like reverse inheritance would fit better because of several reasons. One main reason is the fact that the language supports multiple inheritance. Another big advantage is the philosophy of the language which includes feature adaptations. Reverse inheritance needs such adaptation mechanisms. Another argument in favor of Eiffel is the uniqueness of the feature names. This is due to the fact that no overloading is possible. From the class client point of view there is no difference between a method or an attribute query. In other words a feature can be implemented by computation or by storage in a free way. A possible problem related to the other programming languages is the existence of assertions which can easily prevent potential features from being exherited.

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

A second future work would be the implementation of a prototype which will allow using the reverse inheritance class relationship between already existing classes, thus building new class hierarchies. In fact this prototype will implement the semantics of reverse inheritance. One way of doing this is to transform class hierarchies using reverse inheritance in an IDE (Integrated Development Environment). From this point there are several possibilities for obtaining the executable

system: either by writing a compiler capable of generating binary code for the reverse inheritance class relationship, or by writing a translator to convert the source code using reverse inheritance extension with equivalent pure source code.

List of Algorithms

List of Figures

List of Tables

Bibliography

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

[2] Eiffel analysis, design and programming language, June 2005.

[3] 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.

[4] Gul Agha. An overview of actor languages. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, pages 58–67, New York, NY, USA, 1986. ACM Press.

[5] M. Aksit. Separation and composition of concerns in the object-oriented model. *ACM Comput. Surv.*, 28(4es):148, 1996.

[6] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An object-oriented language-database integration model: The composition-filters approach. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 372–395, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.

[7] Walid Al-Ahmad and Eric Steegmans. Integrating extension and specialization inheritance. *Journal of Object-Oriented Programming*, December 2001.

[8] K. Arnold and J. Gosling. *The Java Programming Language*. Sun Microsystems, 3rd edition, USA, 2000.

[9] Grady Booch. *Object-Oriented Analysis and Design with Applications. Second Edition.* Addison-Wesley, 1994.

[10] Yania Crespo, Jos Manuel Marques, and Juan Jos Rodryguez. On the translation of multiple inheritance hierarchies into single inheritance hierarchies. In *In European Conference on Object-Oriented Programming*, 2002.

[11] Jeff Ferguson, Brian Patterson, Jason Beres, Pierre Boutquin, and Meeta Gupta. *C# Bible.* Wiley Publishing, Inc., 10475 Crosspoint Boulevard, Indianapolis, IN 46256, 2002.

[12] Martin Fowler. Dealing with roles. In *Inproceedings of the 4-th Annual Conference on the Pattern Languages of Programs*, Monticello, Illinois, USA, September 1997.

[13] Peter H. Frohlich. Inheritance decomposed. In *Proceedings of the Inheritance Workshop at ECOOP 2002*, Malaga, Spain, June 2002.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1997.

[15] Georg Gottlob, Michael Schrefl, and Brigitte Rock. Extending object-oriented systems with roles. In *ACM Transactions on Information Systems*, volume 14, pages 268–296, July 1996.

[16] 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.

[17] Michael Van Hilst and David Notkin. Using role components to implement collaboration-based design. In *Proceedings of Conference on Object-Oriented Programing, Systems, Languages and Applications (OOPSLA'96)*, California, USA, 1996. ACM Press.

[18] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press, 1992.

[19] Sonja E. Keene. *Object-Oriented Programming in Common Lisp. A Programmer's Guide to CLOS*. Addisson Westley, 1989.

[20] Elisabeth A. Kendall. Role model designs and implementations with aspect-oriented programming. In *Proceedings of the 1999 Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, Denver, Colorado, USA, November 1999.

[21] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[22] B. B. Kristensen. Object-oriented modelling with roles. In *Object Oriented Information Systems*, Dublin, Ireland, 1996.

[23] 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.

[24] Ramnivas Laddad. I want my AOP. January 2002.

[25] 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.

[26] Barbara Liskov and Jeanette Wing. A behavioural notion of subtyping. In *ACM Transactions on Programming Languages and Systems*, November 1994.

[27] Bertrand Meyer. *Object-Oriented Software Construction 2nd ed.* Prentice Hall, 1997.

[28] Bertrand Meyer. Eiffel: The language. http://www.inf.ethz.ch/ meyer/, September 2002.

[29] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1445, pages 355–382. Springer-Verlag, 1998.

[30] 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.

[31] Claudia Pons. Generalization Relation in UML Model Elements. In *Inheritance Workshop of European Conference on Object-Oriented Programming*, 2002.

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

[33] 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.

[34] Herbert Schildt. *C++ The Complete Reference, Third Edition*. McGraw-Hill, 1998.

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

[36] 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.

[37] John Max Skaller. Mixin article from the comp.lang.c++ newsgroup. http://cpptips.hyperformix.com/cpptips/mixins, 1993.

[38] Yannis Smaragdakis and Don Batory. Mixin-based programming in C++. In *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE)*, 2000.

[39] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[40] John Miles Smith and Diane C.P. Smith. Database Abstractions: Aggregation and Generalization. In *ACM Transactions on Database Systems*, volume 2, pages 105–133, June 1977.

[41] Bjarne Stroustrup. *The C++ Programming Language Third Edition*. Addison-Wesley, 1997.

[42] Bjarne Stroustrup. Multiple inheritance for c++. In *European UNIX Users' Group Conference*, Helsinky, Finland, May 2002.

[43] Antero Taivalsaari. On the notion of inheritance. In *ACM Computing Surveys, No. 3*, volume 28, September 1996.

[44] 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.

[45] The AspectJ Team. The aspectj programming guide. Technical report, Xerox Corporation, Palo Alto Research Center, Incorporated, 2003.

[46] 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.

[47] P. Wegner and S. B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *on ECOOP '88 (European Conference on Object-Oriented Programming)*, pages 55–77, London, UK, 1988. Springer-Verlag.