LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

TOWARDS REENGINEERING: AN APPROACH BASED ON REVERSE INHERITANCE - APPLICATION TO JAVA

Ciprian-Bogdan Chirila, Pierre Crescenzo, Philippe Lahire

Projet OCL

Rapport de recherche
ISRN I3S/RR–2005-24–FR

Septembre 2005

RÉSUMÉ :

MOTS CLÉS :

ABSTRACT:

In order to facilitate software reengineering, adaptation and evolution we propose a Java language extension which includes a semantics for reverse-inheritance class relation. Semantics is defined using the OFL model and an intuitive approach. The grammar for language extension and implementation solutions are provided in order to build a translator between java extension language and pure java language.

KEY WORDS :

Reengineering, Reverse Inheritance, Java

Towards Reengineering: An Approach Based on Reverse Inheritance. Application to Java

Ciprian-Bogdan Chirilă, Pierre Crescenzo, Philippe Lahire
chirila@cs.utt.ro Pierre.Crescenzo@unice.fr Philippe.Lahire@unice.fr

Faculty of Automatics and Computer Science,
"Politehnica" University of Timişoara,
Bd. V. Parvan no 2, 1900 Timisoara,

Laboratoire I3S (UNSA/CNRS), Project OCL 2000,
Route de Lucioles, Les Algoritmes,
Batiment Euclide B BP121 F-06903,
Sophia-Antipolis CEDEX, FRANCE

August 29, 2003

Contents

List of Figures

Abstract

In order to facilitate software reengineering, adaptation and evolution we propose a Java language extension which includes a semantics for reverse-inheritance class relation. Semantics is defined using the OFL model and an intuitive approach. The grammar for language extension and implementation solutions are provided in order to build a translator between java extension language and pure java language.

1 Introduction

The goal of this paper is to provide a way to adapt class hierarchies to particular applications. By achieving this goal we aim to obtain a clean way to reuse class libraries.

The approach is based on a **reverse inheritance** class relation, but this is just a mean to accomplish our plan. There are also other possibilities like a wizard or a reegineering tool. The first attempt is made for Java programming language, defining a language extension that allows class adaptation.

The reverse inheritance class relation semantics is defined using the OFL model [12, Cre01]. The use of OFL model guides us to a consistent semantics definition, avoiding possible losses of important aspects.

On the other hand we use also an intuitive approach to define the reverse inheritance semantics. It is necessary in order to explain the behavior of new designed mechanisms like: feature adding mechanism, factoring mechanism, descendant access mechanism, renaming mechanism. The reverse inheritance is meant also to favor rapid prototyping of software projects.

Also there are necessary reengineering rules in order to make code transformations. Depending on the semantics of reverse inheritance the transformations will be automatic or semiautomatic.

The features that facilitate software adaptation included in the approach are: inserting a class into an existing hierarchy, extending an existing hierarchy, adding functionality to a class hierarchy, factoring features from classes, applying a design pattern, implementing a design pattern, composing abstractions by emulating multiple inheritance. All these features will be detailed in the next chapters.

During the process of reengineering general features that facilitates adaptation will be extracted and abstracted. Also problems and constraints are extracted and analyzed. The second attempt will be to apply the abstracted features to Eiffel programming language.

A secondary goal is to facilitate software library evolution starting from the considered approach. The advantage of the considered approach is that you can refuse easily any evolutionary changes. Other major facility is the temporary use of an evolution. An evolved version of the reengineered library is made available for test and debug purposes. If the changes are suitable for the desired application they can be made final and, in this way, the class library evolves.

The following lines present the structure of the current paper. In the following section are presented the approach and the motivations behind it.

In third section are explained case studies from the real world programming, in which the use of reverse inheritance facilitates software reengineer-

ing.

In the fourth section is defined the reverse inheritance relation's semantics by choosing appropriate values for OFL parameters. All details regarding the OFL model will be presented in the section.

In the fifth section the structure of the new mechanisms that will help adaptation and evolution is anatomically analyzed.

In the sixth section is defined the rest of the semantics using an intuitive approach analyzing the impact of the newly added mechanism in Java with the already existing ones.

In the seventh section are discussed implementation solutions and their constraints.

2 Motivation

In this section are presented motivations regarding the necessity of introducing reverse inheritance class relationship in object-oriented programming languages as a mean for adaptation and evolution. Also we want to define and to show how the reverse inheritance can be used in software reengineering.

There are some goals that could be achieved by reverse inheritance:

- maintenance for legacy systems;
- rapid prototyping in the development process of a new software project;
- adaptation for single application, adapting classes to new software contexts, modifying classes towards framework development;
- evolution of a class library towards a mature version.

Reverse inheritance can be used in several ways depending on the type of software we deal with. Reverse inheritance will operate over libraries, frameworks, legacy systems which can be generically named software entities. In the real world of software development the source code of software entities can be available so they can be modified, and sometimes not. So software entities may be read-only. Some of the reasons are enumerated in the following lines:

- software may be copyrighted;
- software have to be unchanged for existing applications;
- software is maintained by third party;
- software of source code is not available e.g. precompiled libraries.

Our approach is presented in figure 1.

The approach has two main directions identified by the two main goals of the paper: software adaptation and software evolution. Each of them will be explained in the following paragraphs.

Adaptation By adaptation in the context of software libraries we understand all the changes that the library has to support in order to fit to new concrete necessities. One concrete necessity can be the context of a particular application. For adaptation purposes we don't intend that the changes to be generic and to be reusable. The evolution of the library is made in the direction of concreteness and towards particular situations. As stated in the

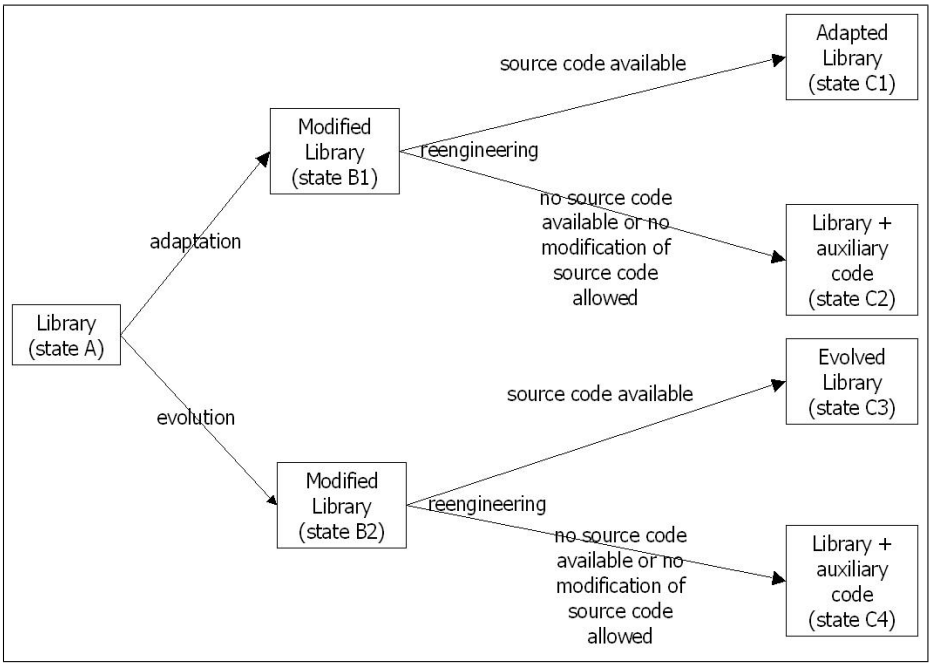

Figure 1: Reverse Inheritance in Software Reengineering

previous chapter we want to achieve adaptation by reengineering. The mean chosen for this aspect in reverse inheritance.

Given a library in state A the designer using reverse inheritance or other equivalent means (e.g. wizard, any other reengineering tool), can freely reengineer the class hierarchy getting it into state B1.

In state B1 we have the possibility to change a hierarchy of classes into a framework. A framework is an abstract design for a particular kind of application which consists of classes [16, Jo88]. The classes can be part of class library or can be application specific. Frameworks have the ability to allow extension of class library components.

In this state the designer is free to use any class relation available in the used programming language: inheritance, reverse inheritance, aggregation composition and any mechanism that these class relations have.

For instance in a Java software library, one can use the factoring mechanism, included the reverse inheritance relation in order to reorganize features in classes. The details regarding this mechanism are explained later in one dedicated chapter.

Another example of adaptation is the one presented in [30, Sak02]. The sample's main idea is class hierarchy reorganization. A version of this modified sample is depicted in figure 2. The example is about a set of classes:

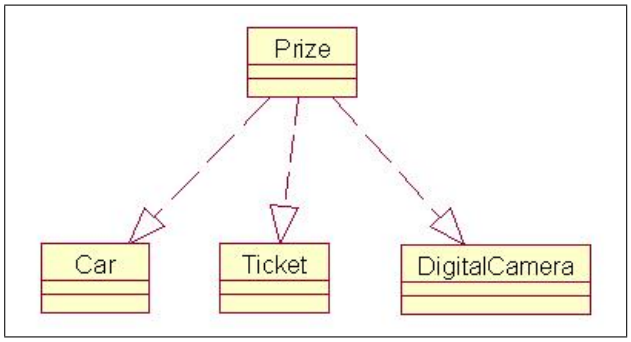

Figure 2: Sample of Class Reorganization

Car, Ticket, DigitalCamera that belong to different hierarchies and have to be adapted to a new situation. The new context is about prize winning and implies a new class hierarchy, the one in figure 2. There are also some other solutions like creating subclasses: CarPrize, TicketPrize, DigitalCameraPrize or changing the superclass to Prize, but both solutions are inconvenient.

In state B1 we obtain an intermediary form of the library, written in a language extension, which is not yet compilable with any classic compiler. In order to achieve this goal, translators towards pure object-oriented languages have to be built. Also this form of library needs validation. In order to obtain the validation, support for test and debug is necessary.

Using reengineering the modified form of the library in state B1 is driven to state C1 or C2 depending on the source code availability.

If we have access to the sources then reengineering is possible and we can obtain new compilable source code of state C1. The reengineering process is made by a translator which possibly has to be assisted by the designer.

During the reengineering process all the reverse inheritance class relations are replaced by entities existing by default in the programming language. For instance some of the reverse inheritance class relations are replaced with simple inheritance relations.

The main characteristic of this state C1 is that all sources have to be pure and readable by everyone even they are generated by a translator. This state of the library can be used very easily for test and debug purposes and also as the adapted final version. On the other hand state C1 implies acceptance.

This situation occurs when source code of the library is not available. In this case the translator in the reengineering process will generate not readable source code and/or byte code. This state is meant just for temporary test and debug actions.

Evolution Evolution in software libraries means all the changes in order to obtain the desired level of abstraction for reusing the code to several kinds of applications.

State A is the initial state of the library and the goal we need to achieve is to obtain an evolved version of it. The following steps are similar to those presented in the adaptation process.

State B2 accepts designer's modifications towards the new evolved version.

When sources are available the reengineering helps the designer to get the library in state C3. The main characteristic of this state remains readability.

If no sources are available then we get state C4 which is similar to C2.

Drawbacks of the Approach There are some disadvantages using reverse inheritance. Some reluctant programmers may say that is difficult to understand how to use it. Others may say that it involves overhead activity and others may say that reverse inheritance can break the code.

3 Case Studies Dealing with Reengineering

In this chapter are presented case studies that will introduce the reengineering capabilities of reverse-inheritance. We intend to give intuitive examples on how the new reverse inheritance class relation can help the code to adapt and to evolve. Each of the following subchapters describes one case study which stands for a specific functionality. All samples are designed using UML class diagrams in order to be independent from any implementation object-oriented programming language.

3.1 Need for Abstraction

A class hierarchy can evolve downward using inheritance and we propose an upward evolution using reverse inheritance. The downwards evolution implies the creation of specialized classes, and the upwards evolution implies the creation of abstract classes.

In Java programming language the inheritance class relation stands for both specialization and generalization abstractions [1, AG00]. In UML the situation is viceversa, specialization is implemented using generalization relation [23, OMG03]. Although the two relations are redundant they may be used together [6, CL01].

In order to achieve adaptation and evolution programmer may use reverse inheritance class relation which implements the generalization concept.

Example This example presents a typical situation in which the need for abstraction is highly motivated.

There are several possible contexts for this scenario. The new designed application models concrete integrated circuits: circuits of type A, B and C. In the process of electronic application modelling the architect creates one class for each concrete circuit type: CircuitA, CircuitB and CircuitC. Other possibility is to reuse these classes from a library which is provided by different third-party vendors or they are part of a legacy system. In order to

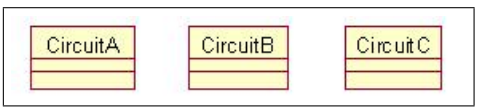

Figure 3: Separating Specialization and Generalization (before reengineering)

manipulate easier integrated circuit instances, the design architect generalizes the concrete circuit concepts and he creates an abstract circuit class named

AbstractCircuit which will be the supertype of circuit modelling classes. All

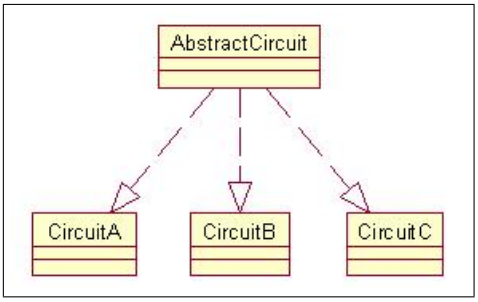

Figure 4: Separating Specialization and Generalization (after reengineering)

the code which is common to all circuits can be placed in the abstract class in order to avoid code duplication. The use of reverse inheritance relation increases code readability and class relation clarity.

3.2 Composing Abstractions

In system modelling, composition of abstractions are often needed. In some programming languages we may find mechanisms fully or partially conform to support this need.

In Java language [1, AG00] there is a partial implementation of this mechanism. The composition works only at the level of interfaces.

In C++ language [32, Str87] and Eiffel language [21, Mey02] is implemented the full mechanism of multiple inheritance between classes relation.

We will give another solution based on reverse inheritance class relation that permits composition of abstractions. This solution addresses the programming languages that do not have the complete multiple inheritance mechanism.

The following example uses reverse inheritance in a special way in order to join several basic concepts that are already modelled by existing classes. The sample is about building a calculator-watch, starting from existing implementations for watch, calculator and cronograph.

The solution is intuitive and implies simply to compose the three concepts into a single one. Composition is made by applying the reverse inheritance relation from each basic concept class: Watch, Calculator, Cronograph to the desired target class CalculatorWatch.

At this level the design is very clear but after the reengineering process the new design obtained may loose it's accuracy.

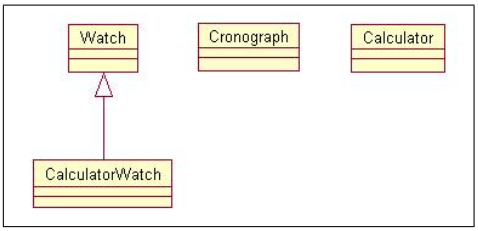

Figure 5: Composing Abstractions (before reengineering)

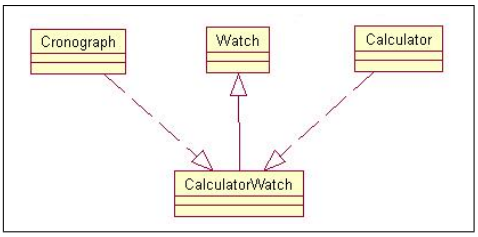

Figure 6: Composing Abstractions (after reengineering)

3.3 Inserting a Class into an Existing Hierarchy

The analyzed case study deals with unforseen changes which involves architecture changes. Adaptability from this point of view discussed in [18, Lie95] is the capability of software to be easily changed.

The sample deals with class hierarchy reorganization which is solved by reverse inheritance.

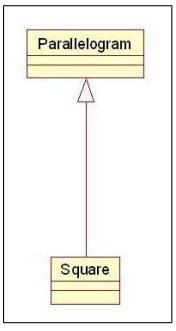

Figure 7: Inserting a Class into a Hierarchy (before reengineering)

A typical situation is one in which we have a library that contains graphical components [6, CL01]. For some of the reasons classes Parallelogram and Square are read-only. We want to add a new class to the hierarchy preserving

architecture correctness from the object-oriented paradigm point of view. In euclidian geometry one may state the followings:

- any rectangle is a parallelogram;
- any square is a rectangle.

Euclidian geometry forces type Rectangle to be subtype of Parallelogram and supertype of Square. The first type relation can be implemented very easily with an existing class relation: simple inheritance between classes.

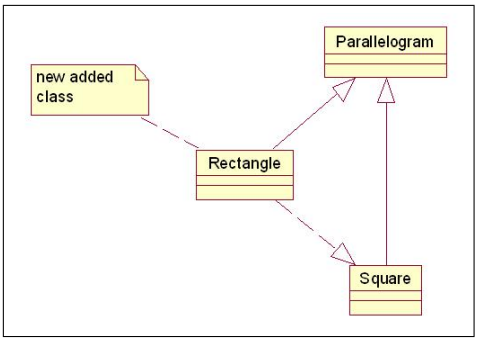

Figure 8: Inserting a Class into a Hierarchy (after reengineering)

The second subtyping relation could be solved using simple inheritance between Rectangle and Square which implies the modification of class Square which is read-only.

Because type Rectangle is a generalization of type Square, it is more natural to use a reverse inheritance relation between the two types.

3.4 Extending an Existing Hierarchy

The studied case involves unforseen structure changes, that are discussed in [18, Lie95], and the possibility of third party software inclusion.

We go further with the example of graphical components and we imagine a new typical situation. Initially a graphic library contains only Parallelogram type components, later on an extension of the library is needed because of the decision to include also Ellipse type graphical components.

The solution is to create an abstract type Shape which is the supertype of Parallelogram and Ellipse. Classes Ellipse and Circle will be derived from Shape and Ellipse and the problem of organizing round graphical component classes is solved.

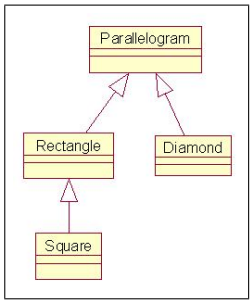

Figure 9: Extending an Existing Hierarchy (before reengineering)

The problem arises in connecting class Parallelogram an class Shape. One possibility is to modify class Parallelogram in order to extend class Shape. The solution is not valid because class Parallelogram is read-only, so the other possibility is to generalize the type of Parallelogram into Shape. This example concludes also that a reverse inheritance relation is needed between the two classes.

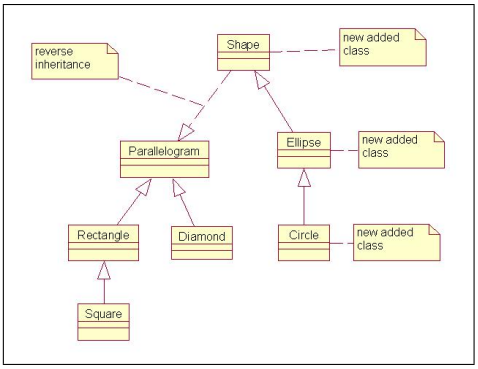

Figure 10: Extending an Existing Hierarchy (after reengineering)

In the figure 10 we managed to build a design solution using reverse inheritance, that preserves consistency and respects object-oriented principles.

3.5 Adding Features to Classes

One benefit of using reverse inheritance class relation is that one may add attributes and methods to a class or to a hierarchy of classes without modifying them manually.

So we can add very easily state and functionality to a class hierarchy [18, Lie95].

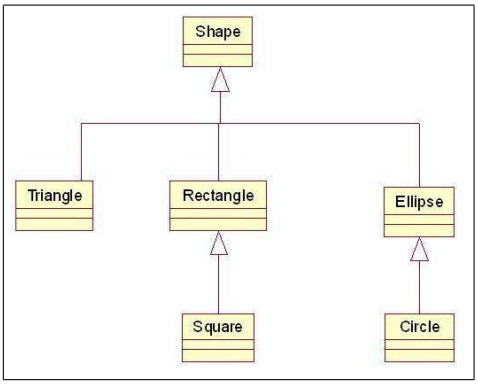

Figure 11: Adding Features to Classes (before reengineering)

The idea of adding common features to a read-only class hierarchy could improve the state and functionality of classes without modifying manually their read-only source code. In the current example we have a hierarchy of classes, actually the same graphical components which misses an attribute regarding area. One solution is to modify class Shape and to add a member and some methods regarding this aspect. The other solution proposed by us is to use reverse-inheritance with a newly created class AbstractShape which contains the new state and behavior.

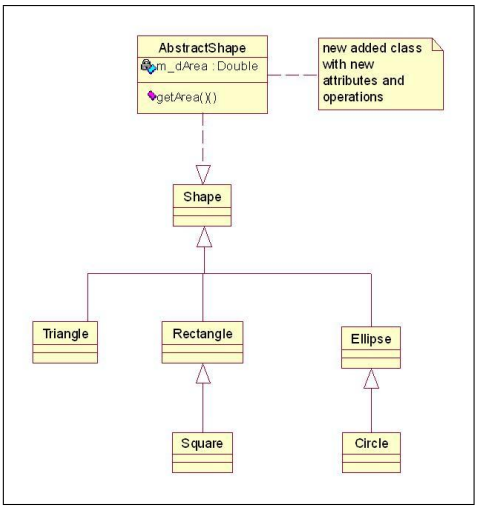

Figure 12: Adding Features to Classes (after reengineering)

The resulted code has the state and functionality extension incrementally kept in newly added class AbstractShape. In fact this is just another way

of class extension by difference, like simple inheritance. Simple inheritance creates the extension on a lower level of abstraction while reverse inheritance manages extension on the highest abstraction level.

3.6 Factoring Features from Classes

The reverse functionality of the case study presented above, is to extract common state and behavior from hierarchies of classes. This way we can avoid data and code duplication and we facilitate software maintenance [19, Mart00].

In the process of factoring, name conflict problems arises. In [17, LHQ94], [30, Sak02] are described mechanisms which handle name conflicts in reverse inheritance class relations. Some of the ideas will be used in the definition of factoring and renaming mechanisms further on.

In the chosen case study we deal with a couple of classes that have same functionalities and they must be reorganized in order to build a new hierarchy. The functionalities may be stored in methods with different names or signatures.

In our example we considered a simplified situation in which the common members and methods have same signatures in all classes. All the other situations can be reduced, using the rename mechanism, to the current one. The rename mechanism is presented in one of the next chapters.

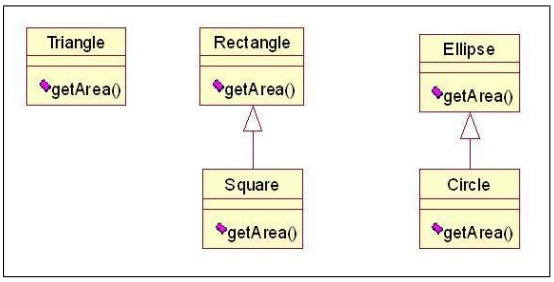

Figure 13: Factoring Features From Hierarchy (before reengineering)

All the graphical component classes have some common features that we propose to factor. The getArea() method is the candidate feature that can be factored.

The idea of factoring features is to manage the uniform distribution of selected behavior between classes. Some classes in a hierarchy may have a concrete implementation of a feature and some may not. We want to enable the inheritance of factored features in classes in which they are not implemented.

Supposing that class Triangle has no implementation for getArea() method, with the factoring mechanism we can fill the implementation gap of methods in classes.

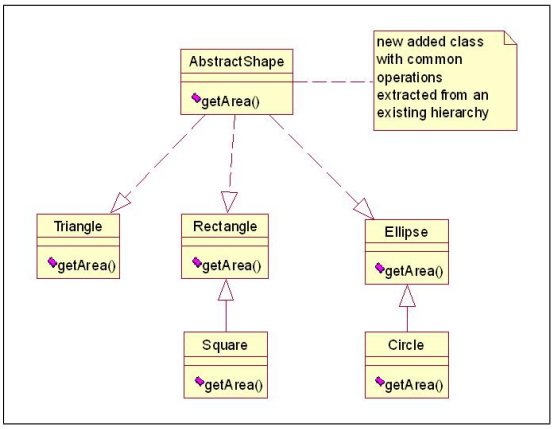

Figure 14: Factoring Features From Hierarchy (after reengineering)

The feature renaming mechanism can help also the process of factorization. There is the case of renaming a feature with the same functionality and different name. We can imagine that class Square has another name for the method that computes area: getSurface(). So in order to build a consistent hierarchy, a renaming is necessary. Method name getSurface() is renamed to getArea().

3.7 Applying a Design Pattern - Composite

This example shows how using reverse inheritance, we can apply a design pattern to a hierarchy of classes. For this case study we selected the Composite [15, GOF96] design pattern.

The Composite design pattern is a structural pattern which composes objects into tree structures. The anatomy of the design pattern is illustrated in figure 15.

The main idea is to have an abstract class Component that represents both primitives and containers. In the object tree the primitives will be leaves and the containers will be composites.

Component declares the interface from the objects in composition, declares an interface for managing child components.

Leaf represents leaf objects in the composition and they have no children.

Composite defines behavior for components having children, stores the

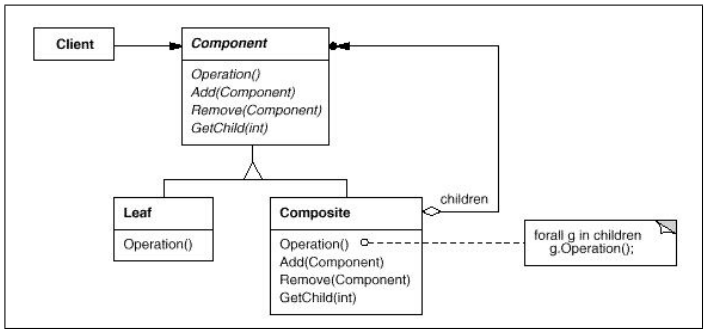

Figure 15: Composite Design Pattern - Structure

child components, and the child related operations from the Component interface.

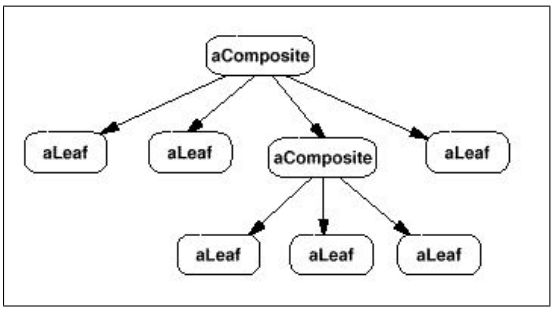

Figure 16: Composite Design Pattern - Sample

The object tree in figure 16 is a sample of how objects of different types can be linked in a hierarchy.

Continuing with our graphical component classes, some end user of the hierarchy would like to create a figure graphical component which has to contain all types of shapes including also figure type. The classes may belong to different libraries.

The idea we want to use in this example is to create a superclass of all concrete classes using reverse inheritance. If we deal with hierarchies of classes the most abstract class from it has to be linked with the new created superclass using reverse inheritance. In this way you do not have to worry about altering classes or class hierarchies.

Then the composite class will be defined inheriting from the superclass using direct or reverse inheritance. It is mandatory for the composite class to contain a collection of superclass types in order to implement completely

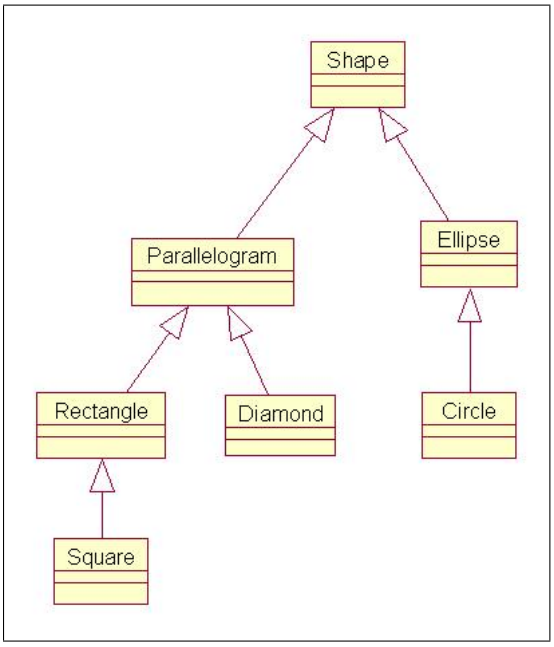

Figure 17: Applying Composite Design Pattern (before reengineering)

the design pattern.

The sample considered contains two independent class hierarchies: one starting with Parallelogram and one starting with Ellipse. Al the level of the most abstract classes from hierarchies we apply the reverse inheritance. We define a new superclass Shape which infers Parallelogram and Ellipse classes. With this transformation we added a new layer of abstraction to the application design. Class Figure is created by deriving from Shape using direct inheritance, because we prefer the usage of direct inheritance instead of reverse inheritance. In all situations we want to favor the "extends" relation, "infers" relation is just a tool for solving the more complex difficult adaptation or reuse problems. To accomplish the design pattern class Figure has a collection of Shape. In this example the collection is implemented by an array.

In conclusion we may state that where a design pattern, based on a higher level of abstraction in the class hierarchy, is needed, reverse inheritance can help. It facilitates the addition of the new abstract layer, thus the desired design pattern can be easily used.

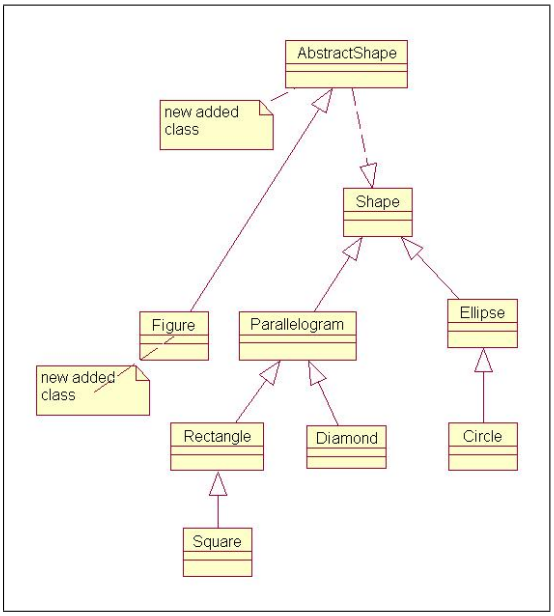

Figure 18: Applying Composite Design Pattern (after reengineering)

3.8 Implementing a Design Pattern - Adapter

This case study starts from the different implementations of the Adapter design pattern [15, GOF96]. First there are presented the idea, the structure of the pattern and the existing implementations.

The main idea of the pattern is to convert the interface of a class to the interface that a client expects. Adapter facilitate classes with incompatible interfaces to work together.

The participant classes involved in the structure of the pattern can be grouped in this manner:

- Target - domain specific interface that Client uses;
- Client - works with object conform to the target; interface;
- Adaptee - existing interface that needs adapting;
- Adapter - adapts the interface of Adaptee to the Target; interface

The structure of the patterns is dual. To adapt from one interface to another there is the possibility of using multiple ihneritance. Some of the object-oriented programming languages like Java do not have this class relation implemented so they have to use something else. The Adapter class inherits from the Adaptee the specific request.

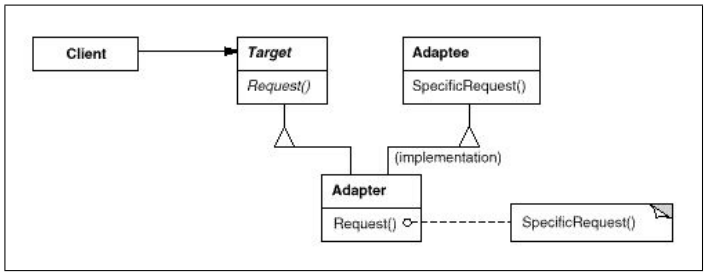

Figure 19: Adapter Design Pattern - Structure (inheritance)

The second possibility is to use class composition. The creation of an instance of a Adaptee class will permit the access to the needed specific request. This solution is realizable in the main of the object-oriented programming

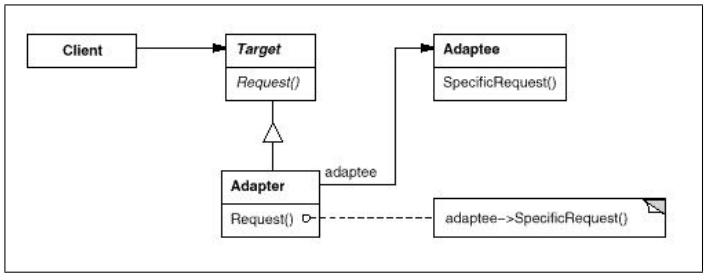

Figure 20: Adapter Design Pattern - Structure (composition)

languages, but includes the overhead of the Adaptee object management.

We start from the Adapter design pattern case study from [15, GOF96] and we are modifying it using our approach based on reverse inheritance.

First we identify the classes that compose the pattern. The DrawingEditor class plays the role of the client, the Shape abstract class is the Target of the adaptation. The TextShape class is the Adapter and the TextView is the Adaptee. The idea of the sample is to develop a class hierarchy of Shapes and to include also a class that manipulates text shapes. The behavior regarding text manipulation is encapsulated in class TextView. There are the two possibilities of implementation: class inheritance or object composition [11, Coop98]. We propose to use the reverse inheritance relation instead of inheritance. Class TextShape is the adapter class and it inherits the desired behaviuor from class TextView. Using this relation the advantage of modifying a class from outside in achieved. The designer will create the new class TextShape, inside he will define the specific behaviuor, for instance, createManipulator() method, and all the delegation methods will simply call

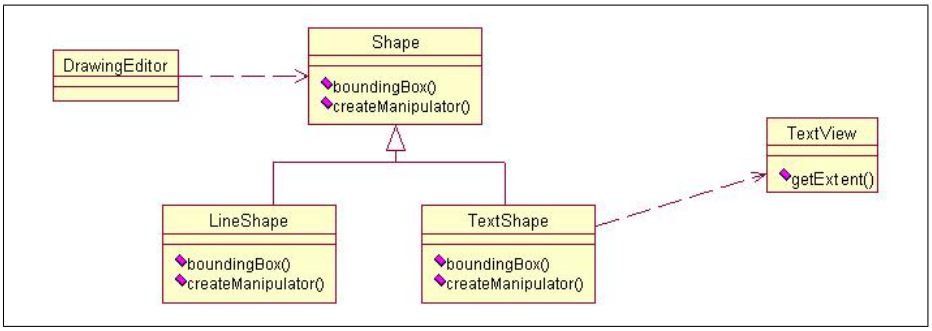

Figure 21: Applying Adapter Design Pattern (before reengineering)

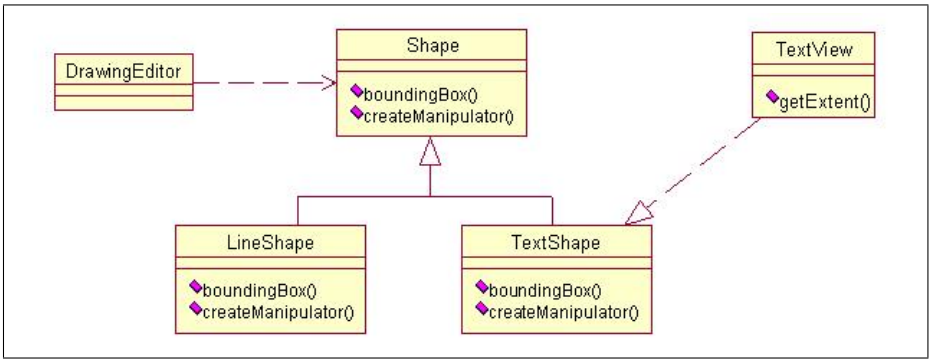

Figure 22: Applying Adapter Design Pattern (after reengineering)

the desired behaviour from TextView. This approach helps only software systems implemented in languages that do not include multiple inheritance class relation. The only possibility is to have a composition object relation. The overhead of this solution is the component object's management.

4 Possible Semantics Derived from the OFL Model

The goal of this section is to define the reverse inheritance semantics systematically. For this purpose the OFL model will be used to catch all the essential features of the relation. This will be made analyzing and choosing between all possible values that OFL parameters can get. For each hyper-generic parameter value there will be presented the motivation. The given semantics does not depend on a particular programming language. For argumentation are used samples of UML diagrams and Java extension code.

4.1 Name

Description The parameter specifies the name of the OFL-component and it must be unique in the language.

Value Reverse-Inheritance Between Classes

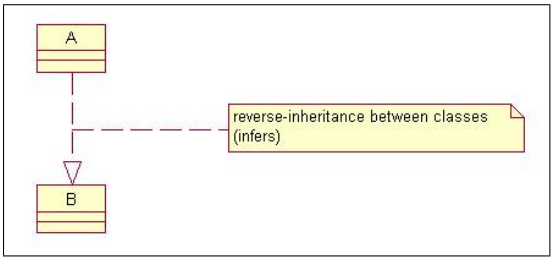

Figure 23: Reverse Inheritance Class Relation

Example We chose a UML like notation for the reverse inheritance class relation.

```
class B
{

}

class A infers B
{

}
```

The *infers* keyword is selected to denote the reverse inheritance relation between classes A and B. Class A is the source ancestor class and class B is the heir target class.

4.2 Kind

Description The parameter specifies the sort of the OFL-component. Import is used for inheritance and all other importation links between descriptions. Use is for aggregation, composition, and all the other use links between descriptions.

Value import For reverse-inheritance class relation the value for this parameter is import because the source class may import features from target classes.

4.3 Context

Description This parameter has two possible values: language or library.

Value Language We choose the value of language because we desire to interact with all the elements of the rest of the language.

4.4 Cardinality

Description Defines the minimal and the maximal cardinality of a relation.

Value several possible values. For each of them we have illustrated a class diagram example. Some of them are just theoretical examples, some are already used in programming languages and some are serious candidates. The order of the possible values is a systematical one in order to include all the possibilities. The values are not absolute, some of them can be obtained combining others.

Example A1 - $1-1$ In example A1, figure 24, using this parameter for class relation some of it's potential mechanisms are favored. The mentioned mechanisms will be presented in the next chapter. If we think about descendant access mechanism there are no difficulties in selecting features. Also the factoring mechanism can be designed in a more simplified manner.

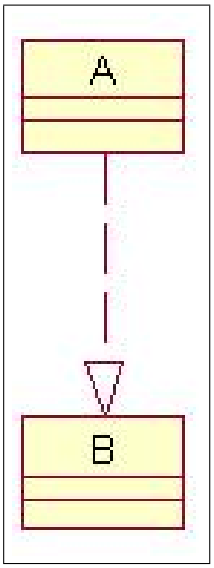

Figure 24: Cardinality - One to One

Example B1 - $(1-1)*N$ This new example B1, figure 25, is derived from the previous one but now we have two targets and two class relations. You may use the reverse-inheritance relation several times with the same source, but the problem which arises is that the interactions related to multiple targets have to be managed globally, from outside relation. This type of cardinality is present in all the simple class inheritance relations implemented in the programming languages. We may mention Java, C++, Eiffel. In the next example we avoid this kind of disadvantage. This kind of usage of the reverse inheritance class relation is forbidden but it is discussed from the analytical point of view.

Example B2 - $1-\infty$ This example B2, figure 26, is very similar with the previous one and it has the advantage of having one relation that manages all the classes involved.

Example C1 - $N*(1-1)$ Sample C1, figure 27, is similar to B1 but having two sources and one destination. There are multiple independent one to one relations sharing the same target.

Example C2 - $\infty - 1$ Example C2, figure 28, depicts a relation that has multiple sources and one target. This kind of cardinality is typical for

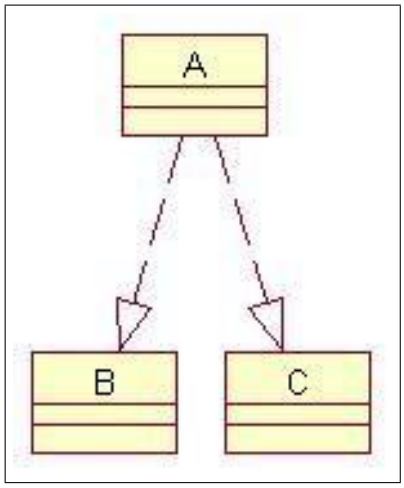

Figure 25: Cardinality - One to One x N

multiple inheritance relation. Languages like C++ and Eiffel allow this value for cardinality in class inheritance relation.

Example D1 - $\infty - \infty$ This example D1, figure 29, is a composition of four one to one relations between 4 classes: 2 sources and 2 destinations. The four one to one relations are the following: A-C, A-D, B-C, B-D.

Example D2 - $\infty - \infty$ The sample above D2, figure 30, is a mix of two basic relations: one to many relation. One basic relations has one source class A and two class targets C and D. The other basic relation is the same as the former but it has as source class B.

Example D3 - $\infty - \infty$ This sample D3, figure 31, is composed of two basic relations. The basic relation involves two sources and one target: A, B sources, C target and A, B sources and D target. The actual value for the cardinality parameter for C++ is this because includes simple and also multiple inheritance.

Example E1 - $\infty - \infty$ This sample E1 figure 32, is very similar to sample B2, figure 26, but allowing two sources. This kind of cardinality is possible only if the language allows the declaration of class relation to be independent of the declaration of the class itself. This is because the declaration of a class limits the multiplicity in one direction for the relation.

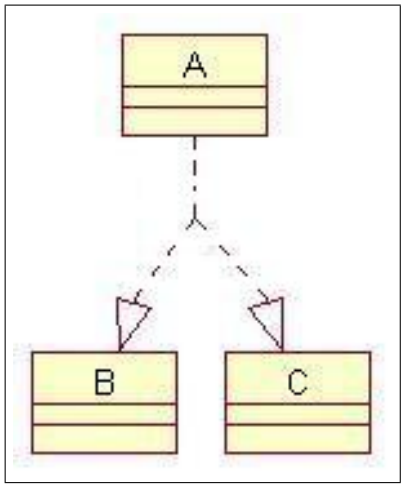

Figure 26: Cardinality - One to Many

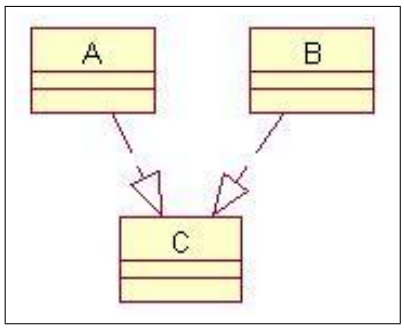

Figure 27: Cardinality - N x One to One

Observations In all the samples above where two sources or targets are found they may be replaced with multiple sources, respectively multiple targets. The simplification does not loose generality and increases clarity. From the samples above results that some relations are basic, they can't be decomposed and some relations are complex composed of basic ones. Modelling the basic relations and the interaction between instances of them, we can get the complex relations. Modelling B1, C1; B2; C2 relations and the interaction between their instances we can get D1 ; D2 ; D3. On the other hand if we model D4 we get B2 and C2 as particular cases. Choosing between the values of this parameter we can achieve more or less functionality in facilitating adaptation and evolution.

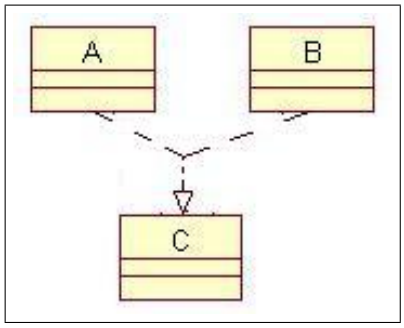

Figure 28: Cardinality - Many to One

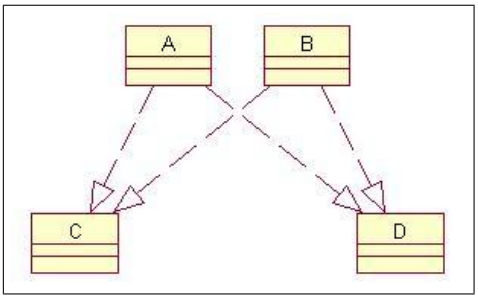

Figure 29: Cardinality - Many to Many (1)

4.5 Repetition Of Sources

Description The parameter indicates if the repetition of source-descriptions in the relation is authorized.

Value forbidden

Example The example above illustrates a theoretical situation in which is possible to specify a class relation using multiple sources and multiple targets. Java, C++, Eiffel philosophy does not have any means for expressing relations with multiple sources. So this is why the values of the discussed parameter is false.

4.6 Repetition Of Targets

Description The parameter indicates if the repetition of target-descriptions in the relation is authorized.

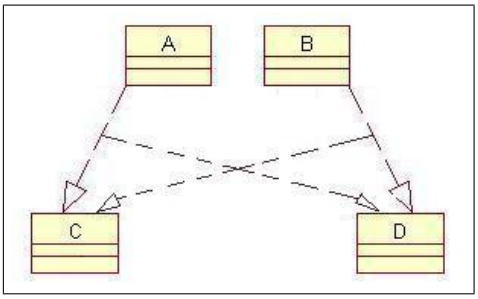

Figure 30: Cardinality - Many to Many (2)

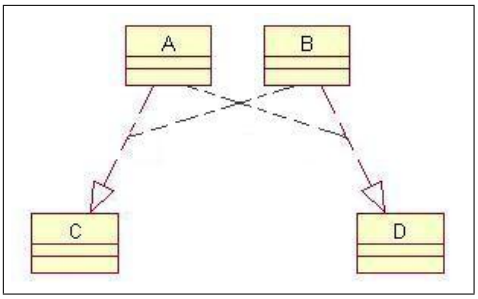

Figure 31: Cardinality - Many to Many (3)

Value forbidden

Example Repetition of targets is not allowed because for this relation makes no sense.

4.7 Circularity

Description The parameter indicates if the relation allows a circular graph.

Value forbidden

Example 1 From the theoretical point of view a class can extend and infer itself. This capability does not improve any aspect of class relation.

Example 2 This example can be correct and accepted from the theoretical point of view if and only if the two classes involved have empty bodies. Otherwise this kind of construction is not allowed because conceptually is impossible to have a class which is at the same time the superclass and the

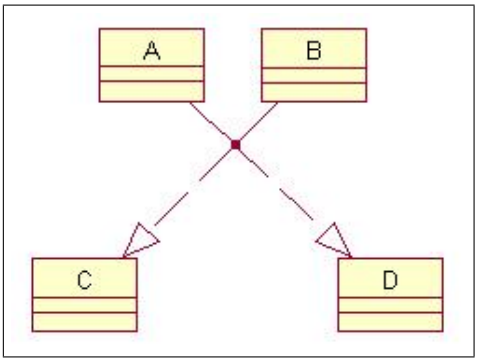

Figure 32: Cardinality - Many to Many (4)

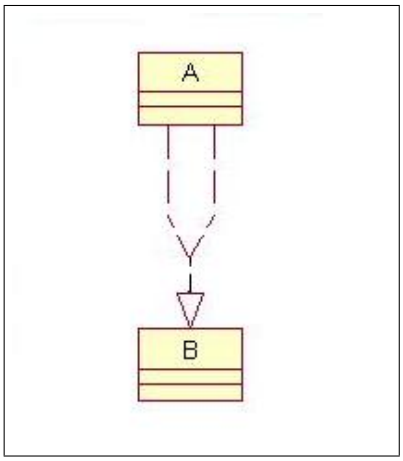

Figure 33: Repetition of Sources

subclass of the same class. The polymorphism mechanism will also fail in such class construction. Another problem arises in the object construction process. Because in object-oriented programming languages, the creation of an subclass instance implies calls to all constructors in the inheritance hierarchy, the chain of calls is recursive, thus infinite. The descendant access mechanism presented in the previous chapter, will not work correctly also.

Example 3 If we build cyclic class diagrams we get the same disadvantages as in the previous example.

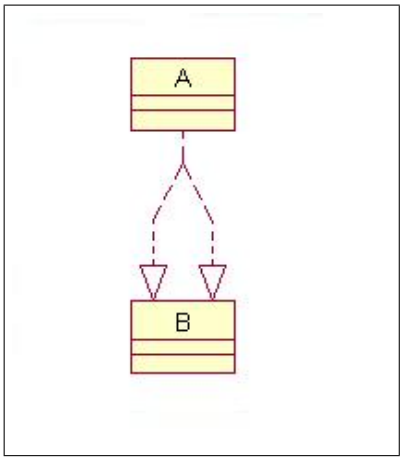

Figure 34: Repetition of Targets

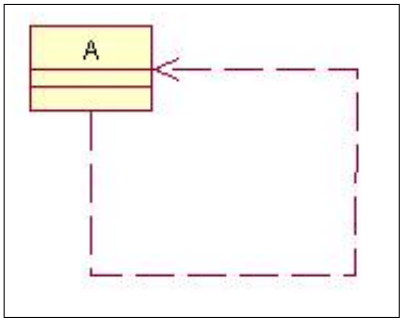

Figure 35: Circularity - One Class

4.8 Symmetry

Description The parameter points out the symmetry of the relation.

Value false.

Example The relation is not symmetrical except the case in which the classes are empty.

4.9 Opposite

Description The parameter indicates a relation which should be equivalent if source and target are replaced.

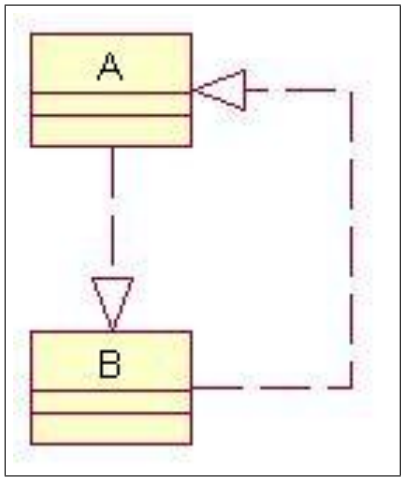

Figure 36: Circularity - Two Classes

Value Extends Between Classes

Example The following two declarations are equivalent from the semantical point of view.

Observation If we agree on the value of the parameter we can not define any new mechanisms like renaming mechanism. The simple inheritance class relation does not have a similar mechanism.

4.10 Direct Access

Description The parameters indicates whether features of the ancestor are directly visible in the heir. This parameter allows to choose the policy of the visibility.

Value allowed If the value is allowed than some features are visible and some are not, the differentiation can be made using a keyword like: *private*, *protected*, *public*.

We are going to discuss about each of the semantics of the access modifiers that will be used by the reverse inheritance class relation. Some aspects from simple inheritance direct access will be included in the semantics of the new designed relation. We intend to be as much general as possible regarding the programming language. The descriptions can be applied to Java and C++.

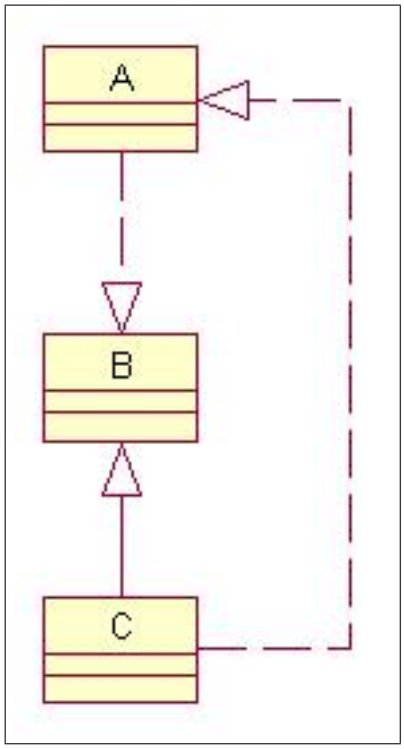

Figure 37: Circularity - Many Classes

The *private* access modifier allows access to features only from inside the class.

The *protected* access modifier allows access to features from inside class, from all it's subclasses and from all the classes in the current package. By current package we mean the package which contains the class having the desired feature.

The *package* access modifier or implicit access modifier allows access to features from inside class and from all the classes in the current package.

The *public* access modifier allows access to features from anywhere.

We will explain how these modifiers affect visibility in a concrete example.

Example

```
package p1;
class A infers B
{
 private int m1;
 protected int m2;
```

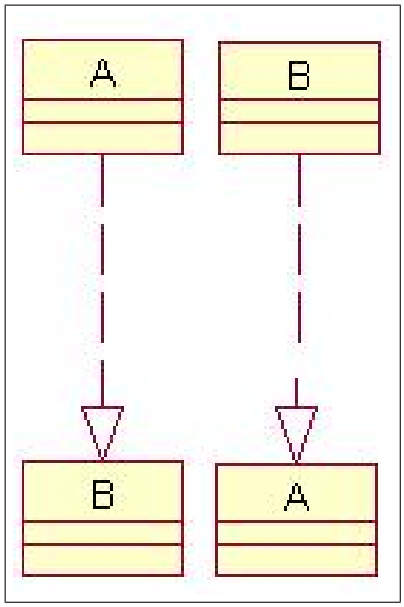

Figure 38: Reverse-Inheritance Symmetry

```
 int m3;
 public int m4;
}
```

We start from a class A which has 4 members with all the possible access modifiers. This is considered the subject class and we will access it's members from different locations, marking the visibility areas. Also we have to mention that class A is located in package p1. On the other hand class A is involved in a reverse inheritance relation with class B which at the moment is not defined. It is not the natural way of building a reverse inheritance relation but for the sake of explanation we made this choice. The access modifiers apply the same way for fields and methods, so in our undergo, for simplicity, we used only fields.

```
package p1;
class B
{
 B b=new B();
 b.m1; // error! not visible
 b.m2; // visible
 b.m3; // visible
```

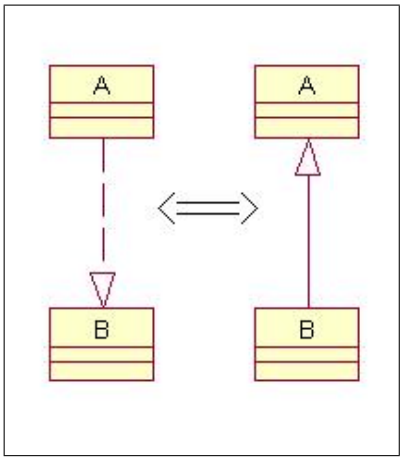

Figure 39: Reverse-Inheritance Opposite

```
 b.m4; // visible
}
```

Class B is the subclass of A from the reverse inheritance relation. Class B will access the members of class A. Except member m1 which is private, all the other members are accessible. So from a heir class from the same package are visible all members except private ones.

```
package p1;
class Test1
{
 public static void test()
 {
  B b=new B();
  b.m1; // error! not visible
  b.m2; //error! not visible
  b.m3; // visible
  b.m4; // visible
 }
}
```

Now we are dealing with a class Test1 which is in the same package p1 as class A. Members m1 and m2 are not visible, but m3 and m4 are. From an outside class are accessible only package access type and public members.

```
package p2;
class Test2
{
 public static void test()
 {
  B b=new B();
  b.m1; // error! not visible
  b.m2; // error! not visible
  b.m3; // error! not visible
  b.m4; // visible
 }
}
```

The final stage is to access members from an outside class Test2 from a different package p2. From this area only member m4 which is public can be accessed.

Conclusions The example illustrates a piece of the reverse inheritance semantics regarding direct access. This part of semantics is also present in the extends between classes relation. If we accept the chosen value than we restrict the addition of some new mechanisms. The factoring mechanism presented in the previous chapter has different semantics. So we accept the chosen value for the discussed parameter.

4.11 Indirect Access

Description Like the previous parameter it indicates whether features of the ancestor are indirectly visible in their heir, naming the ancestor.

Value allowed In Java class inheritance the solution is based on the usage of the *super* keyword. In C++ inheritance, indirect access is made using the resolution operator *::*.

Example 1 In this example is presented how a heir class can access data from an ancestor class.

```
class B
{
 public int m=1;
 public void list()
```

```
 {
  System.out.println(super.m);
  System.out.println(m);
 }
}

class A infers B
{
 public int m=2;
}

class Test
{
 public static void test()
 {
  B b=new B();
  b.list();// output: 2 1
 }
}
```

The reverse-inheritance relation has also to provide the basic mechanisms offered by direct inheritance if we want to replace it with the new relation. This example shows that the inheritance and the super mechanisms are preserved from direct inheritance.

Example 2

```
class B
{
 public int m;

 public void list()
 {
  System.out.println(super.m); // compile time error !!!
  System.out.println(m);
 }
}

class A infers B
{
```

```
 factored public int m;
}
```

This example emphasizes that a factored member can not be inherited so the usage of the super access mechanism is not legal and it will result in a compile time error.

Example 3

```
class B
{
 public int m;

 public void modify1()
 {
  super.m=2;
 }

 public void modify2()
 {
  m=3;
 }

 public void list()
 {
  System.out.println(super.m);
  System.out.println(m);
 }
}

class C
{

}

class A infers B,C
{
 factored public int m=1;
}
```

```
class Test
{
 public static void test()
 {
  B b=new B();
  System.out.println(b.m); // output: 1
  b.modify1();
  b.list(); // output: 2 2 !!!
  b.modify2();
  b.list(); // output: 3 3 !!!

  C c=new C();
  System.out.println(c.m); // output: 1 !!!
 }
}
```

The idea if this example is that the factored member is not inherited always, but is shared. An instance of B will not inherit member m because it already has it. An instance of C will do inherit because it does not has it. This mechanism aims to avoid the duplication of inherited members through reverse-inheritance class relation. A value for the parameter is proposed to express this semantics: allowed-shared.

Example 4

```
class B
{
 public void m()
 {
  System.out.println(1);
 }
}

class C {
 public void m()
 {
  System.out.println(2);
 }
}
```

```
class A infers B,C
{
 public void m()
 {
  inferior(A).m(); // output: 1
  inferior(B).m(); // output: 2
 }
}
```

The descendant access mechanism must have a selector if we want to switch between methods with the same signature from descendant classes. There are some situations in which no name ambiguities occur and the selector is not mandatory.

Example 5

```
class B
{
 public void m1()
 {
  System.out.println(1);
 }
}

class C
{
 public void m2()
 {
  System.out.println(2);
 }
}

class A infers B,C
{
 public void m()
 {
  inferior.m1(); // output: 1
  inferior.m2(); // output: 2
 }
}
```

The example above explains an aspect of how the descendant access mechanism works.

4.12 Polymorphism Implication

Description The parameter indicates the direction of polymorphism. There are available four values we select one and we motivate the choice.

Value down If polymorphism implication has the value down, all instances of target descriptions must be also instances of source description. In our case, source classes are superclasses and target classes are subclasses.

Example All instances of target description are also instances of source description.

```
class B
{

}

class A infers B
{

}

class Test
{
 public static void test()
 {
  A b=new B(); // correct
  B a=new A(); // compile time error !!!
 }
}
```

Class B is a initially created class. Then class A is created in reverse inheritance relation with B. Some tests are made instantiating A and B objects and using different references. First line from the test function of class Test refers using type A an instance of B which is correct, because a B object is also an A. The following line in which reference B tries to hold an A object, generates compile time error, because an A object is not a B object.

4.13 Polymorphism Policy For Methods

Description The parameter indicates if a new declaration of a feature (method) in the source description hides the feature (method) in the target description or redefines it. Possible values are: hiding and overriding.

Value overriding

Example

```
class B
{
 // old method
 public void m()
 {
  System.out.println(‘‘class B method m’’);
 }
}

class A infers B
{
 // new created method
 public void m()
 {
  System.out.println(‘‘class A method m’’);
 }
}

class C
{
 public static void main(String args[])
 {
  A a=new A();
  A b=new B();
  a.m(); // output: class A method m
  b.m(); // output: class B method m
  ((A)b).m(); // output: class B method m
 }
}
```

In class B, method m is inherited from class A by reverse inheritance. Also in class B method m is redefined overriding the inherited one. The same mechanism is present in simple class inheritance. Method m from class B overrides inherited method m from class A.

4.14 Polymorphism Policy For Attributes

Description The parameter indicates if a new declaration of a feature (attribute) in the source description hides the feature (attribute) in the target description or redefines it. Possible values are: hiding and overriding.

Value hiding

Example 1

```
class B
{
 public int m;
}

class A infers B
{
 public int m;
}

class C
{
 public static void main(String args[])
 {
  A a=new A();
  A b=new B();
  a.m=1;
  b.m=2;
 }
}
```

The semantics of the value given to the OFL parameter is the same as in simple inheritance. Object b has two members called m: one inherited from class A and one defined explicitly in class B. The explicit attribute hides the inherited one.

4.15 Variance For Number Of Parameters In Methods

Description This parameter expresses if the redefinition of a method permits the variance of parameter number.

Value non-variant We chose the non-variant value because in most object-oriented programming languages the signature of a method includes the number of parameters. In the next example we try to redefine a method changing the number of parameters.

Example

```
class B
{
 public void m(int i)
 {
 }
}
```

There is defined class B with method m having one parameter i.

```
class A infers B
{
 public void m(int i,int j)
 {
 }
}
```

Next we create the superclass A of B using the reverse inheritance class relation and we define method m with a different number of parameters, two, i and j. Method m is not a redefinition of method m from class B because it has different number of parameters.

```
class Test
{
 public static void main(String args[])
 {
  B b=new B();
  b.m(1);
  b.m(1,2);
```

```
 }
}
```

The two methods declared in the two classes are considered different and does not one redefine another.

4.16 Variance For Type Of Parameters In Methods

Description OFL parameter expresses if the redefinition of a method authorizes or not the variance of parameters type.

Value non-variant The choice is motivated by the definition of a method signature in common object-oriented programming languages.

In the first example we define two reverse inherited classes having a method with the same name, same return type, same number of parameters, but with the type of parameters different.

Example 1

```
class B
{
 public void m(int a,int b)
 {
 }
}

class A infers B
{
 public void m(String s1,String s2)
 {
 }
}
```

The two methods m differ only by the type of their parameters: int and String.

```
class Test
{
 public static void main(String args[])
 {
```

```
  B b=new B();
  b.m(1,2);
  b.m("foo","bar");
 }
}
```

We conclude that m method is not redefined in class B, but inherited, because this example demonstrates that both methods m are available at the level of B instances and none of them is redefined.

Example 2 This example was designed to show how method selection is made when polymorphism is involved.

```
class X
{
}

class Y extends X
{
}

class Z extends Y
{
}
```

We designed a class hierarchy that will be used for parameterizing redefined methods but with polymorphic parameter types. These classes will be the types of the methods' parameters.

```
class B
{
 public void m(X p)
 {
 }
}

class A infers B
{
 public void m(Y p)
 {
```

```
 }
}
```

We have a reverse inheritance class hierarchy, each class has one method m but with polymorphic parameter types: X and Y.

```
class Test
{
 public static void main(String args[])
 {
  B b=new B();
  b.m(new X()); // call m(X)
  b.m(new Y()); // call m(Y)
  b.m(new Z()); // call m(Y)
 }
}
```

The test class will invoke the m method of the B instance with different objects passed as parameters. First an X instance is passed and the call will be made to the method m from class B because it's declaration has the parameter X. Second call is made using a Y object and the call is directed to the method m of class A. The third call is using as argument a Z object. The problem is which of the methods will be called: the one with X argument or the one with Y argument. Because Y is the closest class to Z in the inheritance tree, the answer will be the method with the Y parameter from class A.

4.17 Variance For Type Of Result In Methods

Description The parameter expresses if the redefinition of a method authorizes or not the variance of the return type.

Value variant Because in the signature of a method the return type is not included, two methods can have the same signature, so one can override the other, even if they have different return type.

The following example will demonstrate that once it is redefined a method with the same name, parameters (number and type), but with different return type, it is impossible to access the first method.

Example

```
class B
{
 public void m(int x)
 {
  System.out.println("class B method m");
 }
}
```

In class B we have method m with void return type and one parameter of type int.

```
class A infers B
{
 public int m(int y)
 {
  System.out.println("class A method m");
 }
}
```

In class A we have method m with int return type and one parameter of type int.

```
class Test
{
 public static void main(String args[])
 {
  B b=new B();
  b.m(1); // output: class B method m
 }
}
```

Because the call mechanism do not has the possibility to select by return type which function will be called, method m from class A is redefined at the level of B instances. There is no access from a B instance to the m method of class A.

4.18 Variance For Attributes

Description Parameter expresses if the redefinition of an attribute authorizes or not the attribute's type.

Value non-applicable Because attributes in Java cannot be redefined this parameter is ignored.

4.19 Redefining For Modifiers Of Features

Description The parameter expresses if the redefinition of features authorizes the redefinition of modifiers.

Value allowed

Example

```
class B
{
 public void m1()
 {

 }

 private void m2()
 {

 }
}

class A infers B
{
 private void m1()
 {

 }

 public void m2() // error !!!
 {

 }
```

```
}
```

The modifiers of redefined features in subclasses have to be weaker that the features in superclasses. So if we consider reverse inheritance the redefined features from superclass have to have stronger modifiers. If this rule is not respected a compile-time error occurs.

4.20 Redefining For Signature Of Methods

Description This parameter defines the capabilities of redefining signature of methods. It expresses if is possible to change the signature of a redefined method.

Value forbidden

Example

```
class B
{
 public void m(int x)
 {
  // implementation 1
 }
}

class A infers B
{
 public int(int y)
 {
  // implementation 2
 }
}

class Test
{
 public static void main(String args[])
 {
  B b=new B();
  b.m(1); // implementation 1
```

```
  A a=new A();
  a.m(1); // implementation 2
 }
}
```

The signature is the only mechanism which can distinguish between two methods. A signature identifies a method. So if we cannot redefine methods and change their signature. The signature is invariant regarding returned type and parameter names.

4.21 Redefining For Body Of Methods

Description This parameter describes if the redefinition for body of methods is authorized.

Value allowed The body of a method is an entity that is not part of it's signature.

Example

```
class B
{
 public void m()
 {
  // implementation 1
 }
}

class A infers B
{
 public void m()
 {
  // implementation 2
 }
}

class Test
{
 public static void main(String args[])
 {
```

```
  A a=new A();
  A b=new B();
  a.m(); // implementation 2
  b.m(); // implementation 1
 }
}
```

This example illustrates that redefinition of method bodies is preserved in this class relation.

5 Main Reverse Inheritance Mechanisms applied to Java

The reverse inheritance class relation is a mix of several mechanisms which are presented separately. In this chapter the interaction between these mechanisms is discussed and code transformations ideas regarding their implementation are outlined. We anticipate what a programmer would expect to find in the reengineered evolved sources.

5.1 Feature Adding Mechanism

5.1.1 Motivation

This mechanism takes the idea from the simple inheritance relation and adapts it to reverse inheritance. The member and method addition in a class is a very used reengineering feature in the examples presented in the motivation section. We will define this mechanism with all the necessary adaptations from the point of view of our approach.

5.1.2 Applicability

This mechanism can be easily applied to a class or to a hierarchy of classes. We apply it to a class if we want to reuse the class without changing it, or if we want to evolve in a controlled way. If we use this mechanism on a hierarchy of classes we should add only common features that have sense for all classes in the hierarchy. The difference between the two type of adaptations is that one class adaptation can be more specific, and hierarchy adaptation is more generic.

5.1.3 Anatomy

The anatomy of the mechanism consists of attribute and method declarations in the superclass that will be imported in all the subclasses from the hierarchy.

Components of the mechanisms are:

- feature - can be a method or an attribute;
- source class - the place of the feature's declaration;
- destination classes - the classes that source class infers.

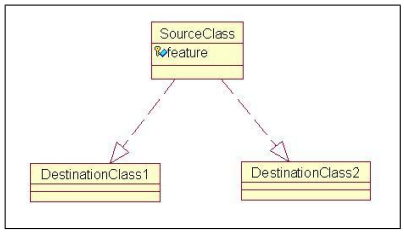

Figure 40: Anatomy of Feature Adding Mechanism

5.1.4 Constraints

We define general situations in which this mechanism can not be used or should not be used because it can generate conflicts. One situation in which we have an applicability constraint is the one that generates a name conflict. When we deal with a class hierarchy, we need a feature that in some classes is already present and we use inheritance mechanism, feature hiding/overriding happen. In order to avoid this ambiguity the mechanism in this kind of situations should be used with factoring. The factoring concepts will be presented in one of the following subsections.

5.1.5 Interactions with Other Mechanisms

Inheritance mechanism can be used independently with all the rest of mechanisms except factoring. Factoring and inheritance mechanism together permit a selective inheritance. This aspect will be described in the factoring subsection. For inherited features only no renaming is provided. Access control mechanism represented by *private*, *protected*, *public* keywords and implicit package access type is preserved entirely from Java language [1, AG00].

5.1.6 Sample of Use

The following example briefly illustrates the main features that can be obtained using the discussed mechanism.

```
class B
{
}

class C
{
}
```

```
class A infers B,C
{
 private int attA;
 public void methA()
 {
 }
}

class Test
{
 public static void main()
 {
  B b=new B();
  b.methA();
 }
}
```

The example consists of three classes. The first two classes B and C have a readonly status and they request new features. The third class A has inside it the new features that are needed by B and C classes. We use reverse inheritance class relation from A to B and C in order to get the desired functionality into A and B classes.

5.1.7 Code Transformation Strategies

One possibility of implementing this mechanism is using the "extends". This solution can be used only in situations in which there are no more than one superclass which infers some other class targets, due to the limitation of single inheritance in Java. Other possibility is to encourage a composition based code transformation like in all object-oriented compiler implementation [4, Bur]. The drawback of this solution is that the resulted application model differs slightly from the designed model. Another strategy can be based on emancipation of the classes on the first level of the inheritance tree [9, CMR02].

5.1.8 Conclusions

Reusing the principles from the classical direct inheritance mechanism we manage to help code reengineering. On the other hand code consistency is

assured by checking constraints which avoids conflicts, ambiguities, violations.

5.2 Factoring Mechanism

5.2.1 Motivation

Factoring does the reverse of what feature adding mechanism does. Features with same semantics from different classes should be factored in one single superclass. Using reverse inheritance we achieve this goal. Factoring features, we avoid code and data duplication.

5.2.2 Applicability

Actually there are several ways of using this mechanism: two elementary usages and combinations with others. One is applied to classes which all have the factored feature, and the other is applied to a hierarchy of classes in which not all classes posses the desired feature. First usage is pure factoring and the second one is a combination of factoring and feature adding.

5.2.3 Anatomy

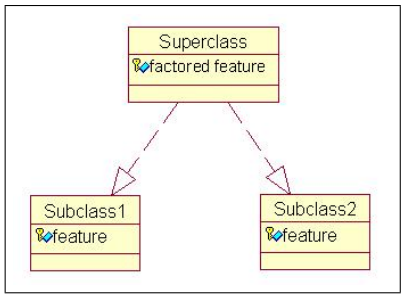

Figure 41: Anatomy of Factoring Mechanism

The components of this mechanism are:

- factored feature declaration proceeded by *factorized* keyword;
- superclass which hosts the declaration of factored feature;
- the subclasses of the target hierarchy which are altered by infers relation.

5.2.4 Constraints

Constraints may appear when desired features from subclasses have different names and the renaming mechanism has to be used also. The renaming mechanism will be presented in the next chapter. The factoring of methods implies the creation of abstract methods in the superclass or the writing of a generic one. If one decided to write a generic method the problem is how to state for each subclass if the generic or the specific implementation should be executed.

5.2.5 Interactions with Other Mechanisms

Adding features mechanism and factoring mechanism are compatible and orthogonal.

5.2.6 Sample of Use

The following example is a simplified possible use of the mechanism. It will explain how the mechanism works.

```
class CircuitA
{
 private int m_nNoOfPins;
 public void test()
 {
    // specific implementation A
 }
}

class CircuitB
{
 private int m_nNoOfPins;
 public void test()
 {
    // specific implementation B
 }
}
```

In the first part of the sample there are two classes CircuitA and CircuitB, possibly from two different libraries having no super class. The reverse inheritance class relation offers the possibility to create a new superclass but

does not provide access to their common features. So using the factoring mechanism it is possible to access at the level of superclass features from subclasses.

```
class Circuit infers CircuitA, CircuitB
{
 factored private int m_nNoOfPins;
 factored public void test()
 {
 }
}
```

Class Circuit is the superclass created by reverse inheritance relation. Also in order to access features from the two classes m_nNoOfPins member and test() method is factored. At the level of Circuit class for the method test we have two choices: it should have a generic implementation or it should be abstract.

5.2.7 Code Transformation Strategies

Regarding the code transformation one possibility is to use simple inheritance, to delete the factored members from subclasses and to put one copy of them into the superclass. The idea is that they will be inherited in all the subclasses. For the factored methods no code transformation has to be made. The polymorphism of inheritance will decide which code has to be executed depending on the type of the calling object. Other possibility is exactly at the opposite pole: to make abstract all the factored features in the superclass and to work with the data and code from instances of subclasses. By abstraction we mean a way of avoiding inheritance of data. Either strategy we choose the idea of organizing data and code for the superclass to exploit them, remains.

5.2.8 Conclusions

Using factoring we help flexibility by eliminating duplicated data and code. Also reverse inheritance gains in functionality at the level of class relation.

5.3 Descendant Access Mechanism

5.3.1 Motivation

If we look at the simple inheritance class relation in object-oriented programming languages we notice that all have the possibility to refer in the subclass features from superclass. This is necessary when in the subclass some superclass code has to be reused. This mechanism already present in Java, C++, Eiffel is motivated by avoiding code duplication and all the errors that may appear when changing needed. Following this idea in the reverse inheritance class relation we think it is necessary to have a similar mechanism. In C++, for instance the access to the features in the superior classes is made using the resolution operator *::*. In Java it is provided a keyword *super*, which is a reference to the superclass part of the object.

5.3.2 Applicability

For this mechanism there are no multiple possibilities to use it.

5.3.3 Anatomy

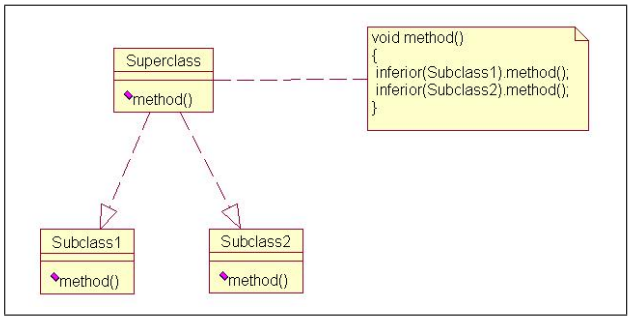

Figure 42: Anatomy of Descendant Access Mechanism

The components of this mechanism are:

- descendant access call using *descendant* keyword, parameterized with the desired subclass. The *descendant* keyword plays the role of a pointer to one of the descendant classes;
- the methods from the subclasses that are called with the help of this mechanism;
- superclass which hosts the descendant calls;

- the subclasses of the target hierarchy which are linked with the infers relation, by the superclass.

5.3.4 Constraints

One severe restriction is regarding the methods from the descendant that has to run in the superclass context. To be more specific if a method from a subclass is invoked using this mechanism to run in the superclass it is necessary to provide also every data that it depends on.

5.3.5 Interactions with Other Mechanisms

The descendant access mechanism is depending very much on the factoring mechanism because of the constraints explained above. With the rest of the mechanisms there are no interferences.

5.3.6 Sample of Use

In the next sample it is presented a way of how this mechanism can be used.

```
class Algorithm1
{
 public void execute()
 {
  // implementation version 1
 }
}

class Algorithm2
{
 public void execute()
 {
  // implementation version 2
 }
}
```

Starting from two classes which will become subclasses later. Each of the classes has one execute() method with two different implementations.

```
class GenericAlgorithm infers Algorithm1, Algorithm2
{
```

```
 factored public void execute()
 {
  inferior(Algorithm1).execute(); // implementation version 1
  inferior(Algorithm2).execute(); // implementation version 2
 }
}
```

The new created class GenericAlgorithm is meant to reverse inherit the first two classes and to access methods from them. Because there can be more than one subclasses the inferior reference has to be augmented with the desired subclass. So an instance of a GenericAlgorithm in it's execute method using this mechanism, are made two calls to the descendants, beneficiating from both implementations.

5.3.7 Code Transformation Strategies

One solution of implementing this mechanism is to use composition [9, CMR02] with some modifications. First the temporary composed object is created, all the factored members values from superclass instance will be copied to the temporary object. The specific method is executed on the temporary object. The new values from the factored fields of the temporary object are copied back to the superclass instance. This algorithm can be repeated as many times as necessary. The main disadvantage is that all data that methods from subclasses depend on has to be factored in the superclass. Other disadvantage could be the performance: low speed of the execution and the memory needed for copying data.

5.3.8 Conclusions

The descendant access mechanism is meant to provide a way of accessing data and code from subclasses. The execution is not straight forward as the one from simple inheritance.

5.4 Renaming Mechanism

5.4.1 Motivation

This mechanism came to help the factoring process. Because in real life software applications the names of the features that denote the same functionality aspect differ. In the presentation of the factoring mechanism was

made a restrictive supposition that all factored features from different libraries have the same name. Now we break the limitation introducing this mechanism. Another motivation for this mechanism is to solve name conflicts. In 1989 Pedersen [30, Sak02] distinguishes two kinds of name conflicts in multiple generalization between two names N1 and N2 from two subclasses A1 and A2. The first conflict arises when the names N1 and N2 are equal, but they denote different methods, the second one is when the names N1 and N2 are different, but they denote the same method. This discussion has to be applied equally to attributes and to methods.

5.4.2 Applicability

The applicability can be made for both types of name conflict discussed in the previous paragraph.

5.4.3 Anatomy

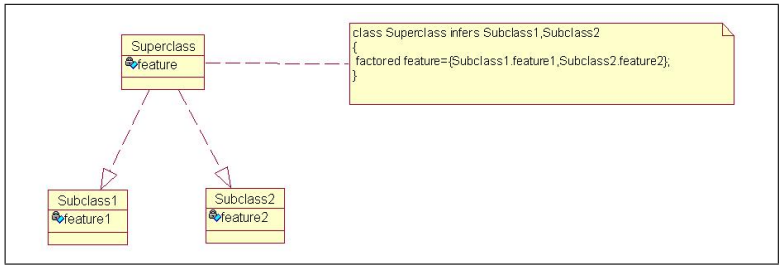

Figure 43: Anatomy of Renaming Mechanism

The components of this mechanism are:

- ??? descendant access call using *descendant* keyword, parameterized with the desired subclass. The *descendant* keyword plays the role of a pointer to one of the descendant classes;
- the methods from the subclasses that are called with the help of this mechanism;
- superclass which hosts the descendant calls;
- the subclasses of the target hierarchy which are linked with the infers relation, by the superclass.

5.4.4 Constraints

Most of the times the renaming has to be coupled with the factoring mechanism. If it is not used the *factored* keyword the new feature will be inherited in all the subclasses. This leads to feature hiding, overriding or to data duplication. The inherited features will be accessible only at subclass level, because at the superclass level the new name is pointing the renamed features in subclass.

5.4.5 Interactions with Other Mechanisms

As mentioned above the main interaction for this mechanism is designed to be with the factoring mechanism. Individual usage of this mechanism is not recommended. The interaction with the adding features mechanism has no side effects. The descendant accesses does not influence the renaming.

5.4.6 Sample of Use

The usage sample will present three classes that have a feature in common. The first two are library classes and the common feature has different names. The third one is the newly created one linked with the first two by reverse inheritance class relation.

```
class Parallelogram
{
 private int area;
}

class Ellipse
{
 private int surface;
}
```

The above classes model geometric shapes and each of them has a member that denotes the area of the geometric shape. The problem encountered is that the features have different names: area and surface. The manipulation of subclass instances can be made only the different features are reunited under a unique name.

```
class Shape infers Parallelogram, Ellipse
{
 factored protected int area={Parallelogram.area,Ellipse.surface};
```

```
}
```

The solution stands in designing a class using the reverse inheritance class relation and applying the factoring and renaming mechanisms. Class Shape infers Parallelogram, Ellipse and it has a factored member for the area of the shapes. It is noticeable the used syntax: initialization with the classified names of the subclass members.

5.4.7 Code Transformation Strategies

Regarding code transformations there are some solutions. One could be based on using access methods for the renamed features.

5.4.8 Conclusions

This mechanism was designed to be closer to real life situations of reengineering.

6 Conclusions, Perspectives and Future Work

With this research report we demonstrated that reverse inheritance class relation can bring serious advantages in facilitating adaptation and evolution.

The next step in our research activity is to define a grammar of the Java language extension to permit syntactically to use the reverse inheritance class relation.

The AST representation of the extension code has to be transformed into pure Java accepted AST. The obtained tree will be unparsed in pure Java code. This code is meant to be human readable, as documented as possible in all the parts where transformations were made.

Also there has to be made an analysis to determine how much of the extension code transformation into pure Java code is automatic, semiautomatic or manual.

The code transformation will be done by a translator with human assistance. At the moment there are in work some transformations schemes from extended Java AST to pure Java AST.

The main future work is to create tools, like plug-ins for the Eclipse programming environment, to provide a friendly user interface to the translator of the extension defined above.

Other future works includes the extension of this approach to other object-oriented programming languages such as Eiffel.

References

[1] K.Arnold and J. Gosling: *The Java Programming Language*
The Java Series from the Source. Sun Microsystems, 3 edition, 2000.

[2] Eric Allen: *Behold the power of parametric polymorphism*
www.javaworld.com, February, 2000.

[3] G. Booch: *Object-Oriented Analysis and Design with Applications*
Second Edition, Addison-Wesley, 1994.

[4] Burlacu Mihai: *Compilation of Object Oriented Languages*
Lecture Notes in Computer Science.

[5] Luca Cardelli: *On Understanding Types, Data Abstraction, and Polymorphism*
Computing Surveys, Vol 17 n. 4, pages 471-522, December 1985.

[6] Pierre Crescenzo, Philippe Lahire *Using both Specialization and Generalization in a Programming Language: Why and How ?*
Research Report, Laboratoire Informatique, Signaux et Systmes de Sophia-Antipolis (UNSA / CNRS), Sophia-Antipolis, France, 2001.

[7] Pierre Crescenzo, Philippe Lahire: *Customisation of Inheritance.* Proceedings of the Inheritance Workshop at ECOOP 2002, Malaga, Espagne, June 2002.

[8] Robert Chignoli, Pierre Crescenzo, Philippe Lahire: *OFL: An Open Object Model Based on Class and Link Semantics Customization*
Research Report 99-08, Laboratoire Informatique, Signaux et Systmes de Sophia-Antipolis (UNSA / CNRS UMR 6070), Sophia-Antipolis, France, March 1999.

[9] Yania Crespo, Jos Manuel Marques, Juan Jos Rodryguez: *On the Translation of Multiple Inheritance Hierarchies into Single Inheritance Hierarchies*
In European Conference on Object-Oriented Programming, 2002.

[10] William Cook: *A Proposal for Making Eiffel Type-safe*
In European Conference on Object-Oriented Programming, pages 57-72, 1989.

[11] James Cooper *The Design Patterns Java Companion*
Addison Wesley Design Pattern Series, October, 1998.

[12] Pierre Crescenzo: *OFL: un modele pour parameter la semantique operationnele des langages a objets - Application aux relations inter-classes*
PhD. Thesis, University of Nice, Sophia Antipolis, December 2001

[13] Pierre Crescenzo: *OFL : les relations et descriptions d' Eiffel et de Java*
Research Report, Laboratoire Informatique, Signaux et Systmes de Sophia-Antipolis (UNSA / CNRS), Sophia-Antipolis, France, 2003.

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

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

[16] Ralph Johnson, Brian Foote *Designing Reusable Classes*
University of Illinois, Urbana-Champaign, Journal of Object-Oriented Programming, June/July, 1988.

[17] Ted Lawson, Christine Hollinshead, Munib Qutaishat: *The Potential for Reverse Type Inheritance in Eiffel*
In Technology of Object-Oriented Languages and Systems, 1994.

[18] Karl Lieberherr: *Workshop on Adaptable and Adaptive Software* Addendum to the Proceedings of OOPSLA 95.

[19] Robert Martin: *Design Principles and Patterns*
http://www.objectmentor.com, 2000.

[20] Bertrand Meyer: *Object-Oriented Software Construction*
Professional Technical Reference, Prentice Hall, 2nd ed., 1997.

[21] Bertrand Meyer: *Eiffel: The Language*
http://www.inf.ethz.ch/ meyer/, revision: 4.82-00-00, September 2002.

[22] Norman Neff: *OO Design in Compiling an OO Language*
Lecture Notes in Computer Science.

[23] Object Management Group - OMG: *Unified Modelling Language Specification*,
version 1.5, 1st ed., March, 2003, http://www.omg.org

[24] William Opdyke: *Refactoring Object-Oriented Software to Support Evolution and Reuse*
AT&T Bell Laboratories, July, 1995.

[25] Dan Pescaru: *A Framework for an Hypergeneric System implementation based on OFL*
Research Report, Politehnica University Timişoara, October 2001.

[26] Claudia Pons: *Generalization Relation in UML Model Elements*
In European Conference on Object-Oriented Programming, 2002.

[27] Paul Rogers: Thanks type and gentle class,
www.javaworld.com, January, 2001.

[28] Paul Rogers: *A primordial interface ?*,
www.javaworld.com, March, 2001.

[29] Paul Rogers: Reveal the magic behind subtype polymorphism,
www.javaworld.com, April, 2001.

[30] Markku Sakkinen: *Exheritance - Class Generalization Revived*,
In European Conference on Object-Oriented Programming, 2002.

[31] Herbert Schildt: *C++, Manual complet*, Editura TEORA, 1997.

[32] Bjarne Stroustrup: *Multiple Inheritance for C++*, European UNIX Users' Group Conference, Helsinki, May 1987.