Towards a Reengineering Tool for Java based on Reverse Inheritance

Ciprian-Bogdan Chirila, Monica Ruzsilla, Pierre Crescenzo, Philippe Lahire, Dan Pescaru, Emanuel Ţundrea

chirila@cs.upt.ro, monomi7@yahoo.com, Pierre.Crescenzo@unice.fr, Philippe.Lahire@unice.fr, dan@cs.upt.ro, emanuel@emanuel.ro

The reverse inheritance class relationship has several potential uses. It can be exploited as a mechanism integrated in one of the object-oriented programming languages or as a mean for reengineering class hierarchies in order to obtain locally adapted software releases. The limited local adaptations and restricted enhancements can be encapsulated in the superclass of the reverse inheritance class relationship and using a software tool they can be applied automatically to each software release. The implementation of the software tool works on Java class hierarchies.

Keywords: reverse inheritance, ordinary inheritance, class hierarchy reorganization, reengineering, local adaptations

1 Motivation

Reverse inheritance [5] is known also as upward inheritance [10], generalization [7], exheritance [8]. The source class of reverse inheritance it is known as foster class [5] or exheriting class [8]. It can be admitted that reverse inheritance is a different model of inheritance, where the subclasses exist first. In this sense it is motivated that it is more natural to start the design from specialized subclasses and than to notice commonalities, which will be later factored into a common superclass [7]. The original idea emerged from the database world where the goal was reusing objects while in object-oriented programming the goal is reusing classes. In order to reuse objects with different interfaces it is necessary to have a heterogeneous common interface [10]. Reverse inheritance is a way of achieving it.

In our approach we use reverse inheritance as a mean for achieving restricted evolution and local adaptation of class hierarchies. We refer in particular to Java applications and class libraries. The basic ideea is to create foster classes [1] containing the necessary local adaptations and enhancements, and to apply them automatically to the class hierarchy. This kind of automation is usefull in the software development process when from a stable release several locally adapted versions have to be obtained. In this way the evolution of the hierarchy and the locally applied adaptations are decoupled: the hierarchy can evolve independently and only later on the local adaptations can be applied.

All the possible local adaptations which can be applied to the class hierarchy give the semantics of reverse inheritance in this approach. The focus of this work is set on the code transformations implied by the semantics of reverse inheritance class relationship which facilitates the adaptation and evolution of class hierarchies The automatic transformation is performed by a software tool in order to generate pure Java code, which will reflect the adaptations and enhancements. We are interested also in identifying the main problems when implementing the semantics of reverse inheritance.

The structure of this paper is presented next. In the following sections we present several code transformations. In the second section we describe the setting a new superclass transformation. There are analyzed different capabilities of the transformation. Section three discusses a transformation concerning behavior factoring. In section four we discus the transformation involved in the situation when a new layer of abstraction is needed in the class hierarchy. Section five deals with state and behavior insertion transformation. In section six prototype details are presented. Section seven discusses related works. Finally, conclusions are drawn and future works are stated.

2 Setting a New Superclass

This kind of class transformation is useful when a set of classes, possibly from different hierarchies, are needed to work together in a new hierarchy. Exherited classes may belong to different hierarchies or different software libraries, so this transformation increases reusability and adaptation.

We will use a formal notation in order to identify classes which will take part in the transformation. Classes B_1, B_2,...,B_n belong to several class hierarchies. The intention of the designer is to reuse those classes by extracting them from their original location and to plant them in a new class hierarchy under a new common superclass A. Using reverse inheritance, class A will exherit B_1, B_2,...,B_n, so in the transformation are involved: one foster class and multiple foster classes. There are also several restrictions about this new designed class hierarchy: i) B_1, B_2,...,B_n are not allowed to have any other superclasses (in fact superclasses are allowed but prior the application of a modified flattening technique, described in [3] is needed); ii) B_1, B_2,...,B_n may have any subclasses; iii) class A must not exist initially in the project, this restriction is set in order to avoid name conflicts; iv) foster class A has to be the only class that exherits B_1, B_2,...,B_n (no multiple foster classes are allowed for the exherited classes); v) foster class A must contain at least one factored feature, this will denote that the designer wants common feature factoring not just adding new features, there are other transformations based on the same preconditions, the designer must choose between them depending on his intentions; vi) factored features of foster class A must exist in the exherited classes B_1, B_2, ...,B_n; vii) types and modifiers of factored features of foster class A and B_1, B_2,...,B_n must be compatible. Next there are presented several adaptation and enhancement use cases of this transformation.

2.1 Feature Enhancement in a Class

The first use case when such a class configuration based on reverse inheritance is necessary is when we want to affect a class by changing its behavior in order to refine it, to enhance it or to adapt it to a new context. We could use inheritance for these purposes but the reverse inheritance approach is an alternative in case we want to keep the original design. Calls to the original version of the code are possible through the ”inferior” calling mechanism like in Beta programming language. The implementation consists in adding the new method in the desired class and renaming the old one. Calls to the old method due to the ”inferior” calling mechanism could be redirected to the renamed method. Another implementation possibility is to insert the old code into the newly added method with appropriate adaptations. The two implementations solutions are presented in the next code sample.

```
// before transformation
class Phone {
 void ring() { // original implementation }
}
class PhoneEnhancement exherit Phone
{
 void pre_ring() {/* pre ring operations */}
 void post_ring() {/* post ring operations*/}
 factored void ring() {
  pre_ring();
  inferior.ring();
  post_ring();
 }
}

// after transformation - solution #1
class Phone {
 void pre_ring() {/* pre ring operations */}
 void post_ring() {/* post ring operations */}
 void old_ring() {/* original implementation */}
 void ring() {
  pre_ring(); // method call insertion
  old_ring(); // original method call
  post_ring(); // method call insertion
 }
}
// after transformation - solution #2
class Phone
{
 void pre_ring() {/* pre ring operations */}
 void post_ring() {/* post ring operations*/}
 void ring() {
  pre_ring();
  // original ring method code inserted here
  post_ring(); }
}
```

The first solution keeps the old implementation in a renamed method. The second solution involves the insertion of the old version code in the enhanced method.

2.2 Feature Adding

Second use case of this class configuration is when we want to add to a class some new features: attributes and methods. This modification could be considered as an enhancement of the class from the state and behavior point of view. The new features are defined in the foster class, which will become the superclass of all the exherited classes. In the implementation, when the reverse inheritance class relationship will be replaced by the ordinary inheritance, the added features will be inherited in all the subclasses, thus features are added into subclasses.

2.3 Feature Factorization

The third use case of the transformation is dedicated to feature factorization. This solves value conflicts, making a common nomination for the same property having different values. This means to extract all the factored features from exherited classes and to move them in the foster class in one copy. Attributes can be factored more easilly than methods. Factoring methods implies factoring signatures and sometimes also bodies. With signatures the restriction is to have the same or compatible signatures in the subclasses. Factoring method bodies implies having the same body in all the subclasses, such a case seems in practice to be very rare. Feature factorization may seem simple at first glance but in fact it involves a lot of problems: class flattening, name conflicts, adaptation routines, signature matching, modifier changing. By signature matching we mean parameter number and type adaptations, return type adaptation, access modifier adaptation.

All three use cases presented till now, can be used together, the designer can benefit from the facilities of each of them. The benefits of using this approach are foster classes reusability, original design preservation. In the following sample, we have a set of classes, which models a set of graphical objects. The intention is to reuse these classes and to reorganize them in a new hierarchy with the help of reverse inheritance.

```
// before transformation
class Square {
 public int x;
 public int y;}
class Circle {
 protected int _x;
 protected int _y; }
class Triangle {
 private int m_x;
 private int m_y; }
foster class Shape exherits Square, Circle, Triangle {
 factored int X={Square.x,Circle._x,Triangle.m_x};
 factored int Y={Square.y,Circle._y,Triangle.m_y}; }
```

The classes above contain a pair of coordinates representing the center of each geometric figure but having different names. We want to reuse the classes in an uniform manner having ”Shape” as superclass.

```
// after transformation
class Shape {
 public int X; }
class Square extends Shape {
 // replace all occurrences of x with X }
class Circle extends Shape {
 // replace all occurrences of _x with X }
class Triangle extends Shape {
 // replace all occurrences of m_x with X }
```

2.4 Implementing Attribute Factorization

All the attributes from exherited classes will be ”moved up” in the newly created superclass. All the occurrences in the subclass will be renamed to the new name of the attribute. In case of public, protected or package access type attributes the changes in all the affected clients must be evaluated. The impact of the attribute renaming is a problem of implementation for the factoring mechanism and needs a solution. A possible source of conflict could be the presence of a parameter, local variable having the same name as the new name of the factored attribute. A possible solution for this incovenience is the inclusion of ”this” keyword in the occurrences of the renamed attributes. The private declared attributes which are subject to renaming can be easily changed because it involves only local changes at class level, affecting no clients. Another pssibility is to duplicate data, but this subject is not interesting for us, because it has many drawbacks.

2.5 Implementing Method Factorization

With methods, implementation is simpler, because in the newly created class we have to put either an abstract or a concrete one, containing the default implementation. In subclasses, new delegating methods can be easily added. This change will not affect any dependent clients, because the original versions will be kept. The advantages are that in the body of the delegating methods, before and after the call to the original version, adaptations can be inserted. The disadvantage of this implementation solution is the duplicated number of factored methods.

2.6 Flattening Class Hierarchies

In the precondition specification part of the transformation the problem of class flattening was raised. Flattening is useful when we want to reuse a class from a hierarchy into another one and we do not want to import all its superclasses. In other words we will extract the class with all its features (locally declared or inherited) and we will adapt it to a different context. We will now enter into the details of this specific problem. The ideas of class hierarchy flattening technique are taken from [2]. It is also known as emancipation in [3]. We need to adapt the linear strategies to fit our class reuse approach. All the features across the inheritance path of the reused class will be reunited in one flattened class. In the case of polymorphic methods the last method from the inheritance path will be added to the flattened class. Another problem are the potential calls to several upward defined versions of that polymorphic method. These calls must be redirected accordingly. A method renaming strategy is necessary in order to manage all the versions of a polymorphic method and its calls. One possibility is to start from the root class down to the reused class and for each level to include in the name prefix of the polymorphic method the name of the class. All polymorphic calls will be updated accordingly. In the case of C++ multiple inheritance, name conflicts may arise because of different features having same names. The flattening of classes which contain multiply inherited features on several inheritance paths, which may cause conflicts. The same name prefix augmentation solution may be applied. Eiffel class hierarchies may have features subject to the renaming mechanism which facilitates overriding.

3 Factoring Out Behavior Signatures - (FOBS)

This transformation is specialized in dealing with factored behavior. It should be used when several classes from different hierarchies contain behavior with the same semantics, which is intended to be used in a uniform manner. The transformation will adapt the exherited class interfaces to a common interface. In order to preserve the superclasses of the exherited classes we are restricted only to factorization of methods but not attributes. Using this transformation we can not add new features to exherited classes. The implementation involves the creation of a new Java interface holding the factored behavior from exherited classes. The transformed hierarchy grants uniform access to the factored behavior. The preconditions for the initial class hierarchy are the same as the ones described in the transformation dealing with the setting of a new superclass, presented in section 2. In the next sample such a transformation sample is presented:

```
// before transformation
class Square {
 void print() {// square implementation}
}
class Circle {
 void display() {// circle implementation}
}
class Triangle {
 void list() {// triangle implementation}
}
class PrintableObject exherits Square, Circle, Triangle {
 factored void print()= {
  Square.print(),Circle.display(),Triangle.list()};
}
// after transformation
interface PrintableObject {
 void print();
}
class Square implements PrintableObject {
 void print() {
 // print method implementation from class Square}
}
class Circle implements PrintableObject {
 void print() {
  // call to display method
  display();}
 void display() {}
}
class Triangle implements PrintableObject {
 void print() {
  // call to list method
  list();}
 void list() {}
}
```

Next, the transformations involved are analyzed step by step. The first step of the code transformation is the creation of a new Java interface, which in fact behaves like an abstract class having no implementation at all, but just abstracts methods. The advantage of the Java interface is that it can be implemented by multiple classes and one class can implement multiple interfaces. The second step is to add into the interface all the abstract methods which are involved in the factoring mechanism of the foster class. All the factored classes will have an extra interface to implement, naming the newly added interface in the class hierarchy. In the exherited class, delegation methods should be added. These delegating methods have the advantage of giving the opportunity of adding adaptation code around the factored method. Using this technique, the number of methods will increase. An important issue regarding delegation of methods is the mapping of parameter which should be made according to the information embeded in the foster class.

4 Abstraction Layer Insertion in a Class Hierarchy (ALI)

This transformation intends to adapt a class hierarchy by adding into it a new abstraction layer. This transformation can be used when the design of a class hierarchy was made quickly or it was not forseen the need for such a layer. In practice, this means that between two classes having an inheritance relationship we want to insert the third class, in order to obtain a more rafined class hierarchy. When several such class insertions are made at the same level in a hierarchy we can consider that those classes create an abstraction layer. The preconditions for this transformations are given by the following rules: i) classes $B_{1,1}$, $B_{1,2}$,...,$B_{1,n}$ extend class A by ordinary inheritance; ii) class $B_{2,1}$ extends $B_{1,1}$, $B_{2,2}$ extends $B_{1,2}$, ..., $B_{2,n}$ extends $B_{1,n}$, there is ordinary inheritance between pairs of classes ($B_{2,i}$ and $B_{1,i}$, where i=1,2,...,n); iii) ...; iv) class $B_{m,1}$ extends B_{m-1},1, $B_{m,2}$ extends $B_{m-1,2}$, $B_{m,n}$ extends $B_{m-1,n}$ there is ordinary inheritance between pairs of classes ($B_{m,i}$ and $B_{m-1,i}$, where i=1,2,...,n); v) class C_1 inherits from $B_{k-1,1}$, C_2 from $B_{k-1,2}$, ...,C_n from $B_{k-1,n}$; vi) class C_1 exherits $B_{k,1}$, C2 exherits $B_{k,2}$, C_n exherits $B_{k,n}$, there is reverse inheritance between C_1, C_2,...,C_n and $B_{k,1}$, $B_{k,2}$,...,$B_{k,n}$; vii) foster classes C_1, C_2, ...,C_n do not exist already in the target project, due to reasons like avoiding name clashes; viii) foster classes C_1, C_2,...,C_n are the only ones that exherit classes $B_{k,1}$, $B_{k,2}$,...,$B_{k,n}$; ix) factored features of foster classes C_1, C_2,...,C_n exist in $B_{k,1}$, $B_{k,2}$,...,$B_{k,n}$; x) types of factored features in foster classes C_1, C_2,...,C_n and $B_{k,1}$, $B_{k,2}$,...,$B_{k,n}$ are compatible; xi) features of C_1, C_2,...,C_n will not mask any inherited features from $B_{k-1,1}$, $B_{k-1,2}$,... ,$B_{k-1,n}$ into $B_{k,1}$, $B_{k,2}$,... ,$B_{k,n}$.

```
// before transformation
class AbstractShape {}
class Shape extends AbstractShape {}
class Square extends Shape {}
class Circle extends Shape {}
class Rectangle extends AbstractShape exherits Square {}
class Ellipse extends AbstractShape exherits Circle {}

// after transformation
class AbstractShape {}
class Shape extends AbstractShape {}
class Rectangle extends Shape {}
class Square extends Rectangle {}
class Ellipse extends Shape {}
class Circle extends Ellipse {}
```

In the sample above, the two classes ”Rectangle” and ”Ellipse” were not foreseen at the beginning in the class hierarchy. So, using ordinary inheritance and reverse inheritance the two classes are integrated between ”AbstractShape” and ”Square”, respectively ”AbstractShape” and ”Circle”. The new resulting class hierarchy is as it should be designed from the start, with all the necessary classes.

5 State and Behavior Insertion in a Class (SBI)

In practice there are situations when we need to affect a class, which belongs to a hierarchy, but not by adding a new superclass line in SNS transformation. One approach is the manual changes in the code which are not reusable, repeatable and are very sensible to error insertion. Once we have designed a foster class containing information about the necessary changes, this foster class can be used anytime when the original class evolves independently. Of course, name conflicts may occur and they have to be avoided. The rule to do this is to disallow the newly added features from the foster class to have the same names as the ones in the exherited classes. This transformation facilitates the adding of new functionality into the targeted class. There are no intentions of reorganizing the class hierarchy. We could use this transformation when we want to adapt a class to some special needs. Adaptation is done by adding state and behavior into the class. The state and behavior from the foster classes may be reused to affect other target classes. This kind of transformation is similar to those implemented in aspect-oriented programming and traits approach. The scenario presented in this section intends to simulate a kind of multiple inheritance class relationship. It involves one or multiple foster classes targeting one subclass which is the subject of adaptation. The target class may, of course, belong to a class hierarchy. The preconditions for this class transformation are the followings: i) classes A_1, A_2,..., A_n exherit B - multiple foster classes, one exherited class; ii) class B may have superclass(es); iii) classes A_1, A_2,...,A_n may exist in the project, but is no recommended due to clarity reasons; iv) foster classes A_1, A_2,...,A_n must have no factored features; v) new added features in A_1, A_2,...,A_n do not already exist in class B locally declared or inherited; vi) features of A_1, A_2,...,A_n are distinct, in order to avoid name conflicts.

In the following sample, class “Shape” was equipped with features representing coordinates, which were encapsulated in foster class “Coordinates”. In the implementation of this transformation, all the new features are added as a part of the afected class.

```
// before transformation
class Shape {
 // original implementation
}
foster class Coordinates exherits Shape {
 private int x,y;
 public int getX(){return x;}
 public int getY(){return y;}
}
foster class Color exherits Shape {
 private int color;
 public int getColor(){return color;}
}

// after transformation
class Shape {
 // original implementation
 // received features from foster class Coordinate
 private int x,y;
 public int getX(){return x;}
 public int getY(){return y;}
 // received features from foster class Color
 private int color;
 public int getColor(){return color;}
}
```

6 Prototype Implementation

The ideas of this research are directed towards the development of an Eclipse plugin prototype which implements partially the transformations presented in this paper. The prototype developed aims at implementing the four code transformations described above. The goal is to introduce the semantics of the foster classes in an already existing class hierarchy, without altering the syntax of the Java language. Thus, the prototype will generate a new class hierarchy respecting Java’s syntax but containing the semantics of the foster classes.

For implementing the code transformations, we choose the Eclipse platform [4], due to its great flexibility and popularity. Eclipse is a highly extensible workbench, feature obtained through the use of plug-ins. The plug-in takes as input the description of several foster classes, described in an XML file, and a Java project opened in the Eclipse workspace and applies the code transformations implied by the structure of each foster class, creating a new Java project in the workspace. Analyzing the structure of each foster class, the plug-in chooses the type of code transformation to be applied.

For describing the foster class model we used the XML technology. An XML file contains a list of foster class definition. For each foster class it has to be defined the name, the superclass, a list of inferred classes, new and referred features. For each new feature it has to be specified a reference to the feature in the inferred class. The XML file containing the definitions of the foster classes is parsed within the plug-in using the JAXB (Java Architecture for XML Binding) framework. In the design of the application we tried to obtain flexibility and reusability of code. Thus each transformation is decomposed in several more simple transformations, which can be reused between the four types of transformations. We used the Composite design pattern to model a transformation. Also each transformation has defined a precondition and a postcondition. The transformation is initiated only if the precondition evaluates to true, and the changes are made definitive only if the postcondition remains true. When defining the conditions we used also the Composite design pattern, each condition being defined as a composite of more simple conditions. For implementing the code transformations implied, we used the JDT (Java Development Tooling) API which comes with the Eclipse platform. This API offers the possibility to apply changes to the ASTs used within Eclipse to model Java compilation units.

7 Related Works

In [6] there are presented several steps in creating manually an abstract class starting from several subclasses. Some of the steps are used also in our SNS and FOBS transformations. In [9] classes are composed out of mixins, which are reusable units of behavior. In the SBI transformation the idea of composition was used but for both state and behavior. In [8] there are presented several semantical elements related to the integration of reverse inheritance in the object-oriented programming languages. In [10] reverse inheritance conflicts are analysed and solutions based on renaming are proposed. Several ideas from these works are used in our approach.

Conclusions

In principle foster classes are designed with the intention to be reusable. They should be applied to each release of the evolving project and even to different projects. The only inconvenients in the way of reusability are the conflicts due to the mechanisms which refer to particular entities in subclasses. This aspect could be improved by using a foster class design wizard which facilitates referring different entities. Code transformations are not analysed thoroughly in all possible situations. A different problem is that adaptations in foster classes may conflict the reengineered hierarchies. At the end of the analysis made, we can conclude that code transformations are highly dependent on the programming language, there is no place for general transformation rules to fit to all object-oriented programming languages.

References

[1] Ciprian-Bogdan Chirila, Dan Pescaru, Emanuel Tundrea. Foster Class Model, SACI 2005 2nd Romanian-Hungarian Joint Symposium on Applied Computational Intelligence, ISBN 963-7154-39-6, pp. 265-272, Timisoara, Romania, May 12-14, 2005.

[2] Jean-Marc Geib Bernard Carre. The point of view for multiple inheritance. October 1990.

[3] 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.

[4] www.eclipse.org

[5] 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.

[6] William F. Opdyke and Ralph E. Johnson, Creating Abstract Superclasses by Refactoring, 1993.

[7] 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.

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

[9] Nathanael Scharli, Stephane Ducasse, Oscar Nierstrasz and Andrew Black, Traits: Composable Units of Behavior, Proceedings of the Inheritance Workshop at ECOOP 2003, Darmstadt, Germany, July, 2003.

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