# Foster Class Model

**Ciprian-Bogdan Chirila, Dan Pescaru, Emanuel Țundrea**

chirila@cs.utt.ro, dan@cs.utt.ro, emanuel@emanuel.ro

*Abstract: Inheritance is a class relationship that enables the extension of object-oriented systems. We use a reverse inheritance class relationship [1] in order to address the goals of limited adaptation and restricted reuse of object-oriented class hierachies. Reverse inheritance provides class hierarchy reorganization and class composition facilities. The reverse inheritance semantics are based a new class model, designed as an extension of the "classic" model of class.*

*Keywords: reverse inheritance, foster class model, class hierarchy reorganization*

## 1 Introduction

We intend to extend Java programming language with a reverse inheritance class relation, also known as exheritance [7], in order to achieve the goals of limited adaptation and restricted reusability. As implementation we chose to regenerate hierarchies containing reverse inheritance class relations into pure Java code. The implied code transformations are out of our paper's scope, so they will not be discussed here.

In object-oriented programming each class relationship has one or more sources and one or more targets. The reverse inheritance class relationship, presented in [1], makes no exception and may have one or more source classes, also named foster classes, and one or more target classes.

Because of the factoring mechanism, feature adding mechanism and the other mechanisms detailed in [1], we conclude that a new class model for the source class is needed. Part of the semantics of these mechanisms will be embeded in the model of the foster class. For prototyping, we selected the extend the Java class model because of the programming language's simplicity and popularity.

The main goal of the paper is to express part of the reverse inheritance semantics [1] with the help of foster class proposed model.

The paper is structured as follows. In the second section we propose the foster class model structure, emphasizing on the elements which differ from a regular Java class model. In the third section we set the rules of consistency for the

reverse inheritance's foster class model. In the fourth section we discuss about a CASE tool implementation, which helps the programmer to build foster classes.

# 2 Foster Class Model

The goal of this section is to define the model of the foster class with it's components. The model is presented using UML notations together with the corresponding syntax. Samples using the proposed syntax are provided and commented.

The first syntactical element which is part of the foster class definition, is the optional use of the "foster" keyword:

foster class Shape exherits Rectangle, Ellipse {}

In the sample above class "Shape" is the source of the reverse inheritance class relationship and classes "Rectangle" and "Ellipse" are the target of the same class relationship. Class "Shape" is the foster class. Keyword "exherits" denotes the reverse inheritance class relationship between the classes.

## 1.1 Anatomy

In this subsection we present the foster class main components. From now on, they will be named entities and they are depicted in figure 1.

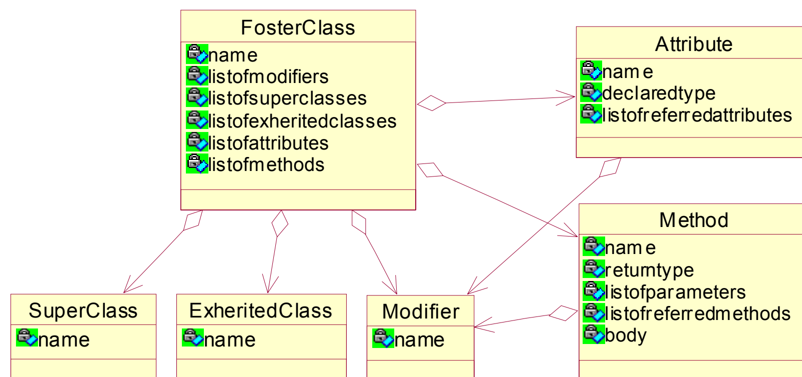


Figure 1
Foster Class Anatomy

By name, we mean a fully qualified name including the package the class lives in (fully qualified class names are specific to Java). The modifier list of a foster class

includes all the modifiers that come with the definition of that class and may have one of the following values: "foster", "public", "abstract". In the definition of a foster class, the "foster" modifier is optional, the reverse inheritance “exherits” keyword is enough to mark a class as being foster.

From the architectural point of view, the foster class has a list of superclasses (in Java the list consists in only one element due to the fact that multiple inheritance is not supported) and a list of subclasses (or exherited classes, which come along with the reverse inheritance class relationship). In the following sample class “Shape” is a foster class involved in two class relationships:

foster class Shape extends AbstractShape exherits Rectangle, Ellipse {}

i) normal inheritance: “extends AbstractShape” and ii) reverse inheritance: “exherits Rectangle, Ellipse”.

The next two entities in the composition list of the foster class are features: attributes and methods which will be detailed further on.

In the next sections we discuss about the regular and factored features that exist in our model.

## 1.2 Attributes

The attribute is part of the foster class structure. It can be of two types: regular and factored. It contains the entities depicted in the following UML diagram:

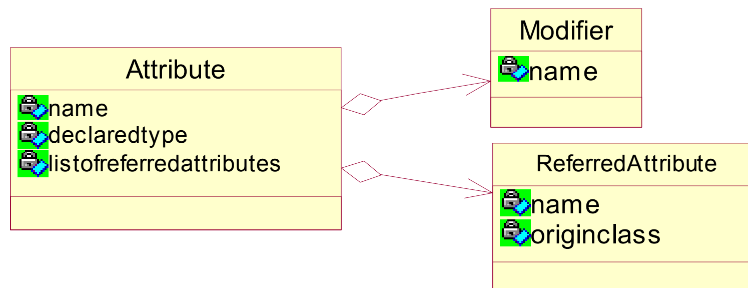


Figure 2
Attribute Anatomy (factored or non-factored)

Name is the simple name of the attribute, declared type is the type used in it's declaration. The type can be a primitive one or a user defined one (like the name of class or interface). The list of modifiers may contain the modifiers found in the declaration statement and may have values like: "factored", "public", "protected", "private", "static". In a consistent modelled foster class, the list of referred attributes is not empty if the attribute is factored, otherwise it is. A static attribute

can not be also factored, so the combination of "factored" and "static" is forbidden.

## 1.3 Referred Attributes

The last component of an attribute is the referred attribute list. A referred attribute must be declared using the “factored” keyword. The list is empty in the case of a regular, non-factored attribute. Otherwise, it contains all the names of the referred attributes along with the names of the classes they live in. In the next sample, we present such a situation:

foster class Shape exherits Rectangle, Ellipse

{factored int area={Rectangle.surface, Ellipse.Area};}

class Rectangle { int surface;}

class Ellipse { int Area; }

The single attribute in class “Shape” is a factored one: "area" and has a list o references. The elements in that list refers to attributes located in classes “Rectangle” and “Ellipse”. Due to the fact that classes may belong to different class hierarchies and they were developed independently an attribute with the same semantics may have different names: "surface" and "Area". The names of the classes that belong to this list must belong also in the list of exherited classes of the foster class, otherwise the foster class is inconsistent. The references from the list of referred attributes, maye be omitted in case they have the same name and the same declaration type. They are considerred to be implicit.

If the type of the factored attribute is incompatible with the type of any referred attribute, the model is inconsistent. In the case of compatible types, type adaptation is necessary.

## 1.4 Methods

The method is the most complex part in the foster class structure. It can be of two types like attributes: regular and factored. It contains the following elements:

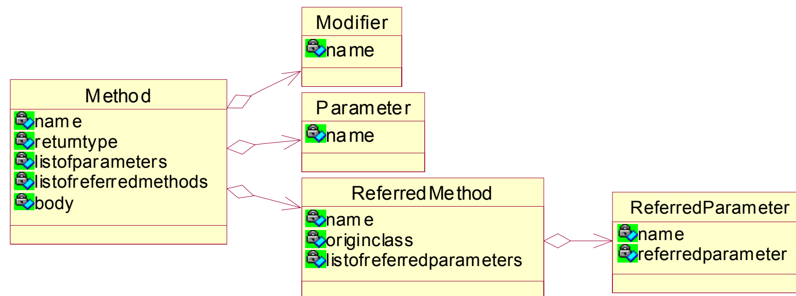


Figure 3
Method Anatomy (factored or non-factored)

## 1.5 Parameters

The list of parameters is a common entity which belongs to the structure of the method. The parameter contains information like: it's name and it's declaration type.

## 1.6 Referred Methods

The same reason of referred classes independent development, discussed in 2.3 in the context of attributes, can be mentioned also in the context of referred methods. In the next sample, we find the same referring mechanism used for attributes, but with a small adaptation for methods:

foster class Shape exherits Rectangle, Ellipse

{factored int draw()={Rectangle.paint(),Ellipse.Draw()};}

class Rectangle { public void paint() {// Rectangle implementation}}

class Ellipse {public void Draw(){ // Ellipse implementation}}

The factored method "draw()" of class “Shape” is declared as referring to method "paint()" of class “Rectangle” and to method "Draw()" of class “Ellipse”. A factored method is useful for renaming or adaptation purposes. On the other hand, it has the role of factoring the features from all exherited classes. The considered methods in this sample have the same semantics but they have different names. If a referred class has a method having the same name with a factored one from the foster class, the corresponding referred method may be omitted from the list. We consider that omitting a method reference in a foster class, implicitly a method with the same signature must exist in the exherited class.

### 1.7 Referred Parameters

In the previous section 2.6 we presented a simplified sample in which parameters of methods were not taken into account. In the case of factoring methods having parameters, we have to establish a link between the formal parameters of the foster class and the formal parameter of the referred classes.

```
foster class Shape exherits Rectangle, Ellipse
{factored int draw(int x, int y)=
        {Rectangle.paint(c_x=x, c_y=y),
        Ellipse.Draw(coordinateX=x, coordinateY=y)};}
class Rectangle
{ public void paint(int c_x, int c_y)
 {// Rectangle implementation}}
class Ellipse
{ public void Draw(int coordinateX, int coordinateY)
 {// Ellipse implementation}}
```

The sample above is developed from the sample of section 2.6 by adding parameters to the factored method: "int x" and "int y". The corresponding referred methods have their own parameters with different names: "int c_x","int c_y" in method “paint()” of class “Rectangle”, respectively "int coordinateX", "int coordinateY" in method "Draw" of class "Ellipse". The links that make the correspondence between the parameters from a factored method and a referred method, are located in the factored method's declaration: "Rectangle.paint(c_x=x, c_y=y), Ellipse.Draw(coordinateX=x, coordinateY=y)".

## 3 Consistency Rules

The model of the foster class has a set of rules that must be checked before declaring a certain foster class consistent. The rules reffer mainly to semantic aspects rather than syntactical ones.

A first rule is that the classes declared in the extended and exherited class lists must exist. The extended class list and the exherited class list must be disjunctive, thus circular inheritance graph is not allowed. This rule expresses direct circular inheritance, but indirect circular inheritance implied by the combination of inheritance and reverse inheritance is forbidden as well.

Feature names obey the same rules as in a regular class. The set of regular and factored features have to be disjunctive. Factored and static features at the same time are not allowed.

References from the referred features declaration must be consistent with the list of exherited classes. In the case of omitting one reference it is considerred that the feature exist with the same name as in the referred class.

The referred entities like types of attributes, parameters, return types must be compatible with the corresponding referres ones.

# 4 Foster Class Builder – Part of a CASE Tool

The foster class model is the blue print for creating foster classes. A foster class building tool is the automated solution facilitating the creation. The tool is based on a reification in Java of the model. The tool offers facilities like: refference name assisting in foster class design, automatic linking to the referred projects, model consistency automatic verification. The Foster Class Builder Tool, together with the code transformation tool, may be part of a CASE tool. Such a CASE tool can be implemented as an Eclipse plugin, having immediate integration with an industrial IDE.

**Related Works**

The goals of class hierarchy reorganization are approached in work [5], presenting a manual approach which consists in the application of several refactorings. The factoring mechanism we are using is inspired from the mentioned work.

The works of [3] is decomposing inheritance into more basic mechanisms of object composition and message forwarding. They are targeting an object model which is based on separating messages (abstract operations) from methods (concrete operations).

Issues aboute the reverse inheritance class relationship are discussed in works like [4],[6],[7]. In our work we took the concept of reverse inheritance and we proposed a new semantics behind it, to achieve our goals.

Adaptation issues like signature matching, presented in [8] are close to the reference mechanism for attributes and methods used in our model.

**Conclusions and Future Work**

In this paper we extended the design of a regular class model in order to be able to achieve the goals of limited adaptation and small evolution of class hierarchies. In the structure of the foster class we encapsulated parts of the semantics of reverse inheritance class relationship. The proposed model of foster class imply a set of

possible code transformations towards our goals of adaptation and reuse. By creating a foster class using this model, we specify indirectly a certain set of transformations that will be applied to the reengineered class hierarchy. There are several possible transformations that can be performed, such aspects are not covered in this paper, considerring it as future work.

**References**


[1] Ciprian-Bogdan Chirila, Pierre Crescenzo, and Philippe Lahire. Towards reengineering: An approach based on reverse inheritance. Application to Java. Research report, Laboratoire Informatique, Signaux et Systmes de Sophia-Antipolis (UNSA / CNRS), France, July 2003

[2] Michel Dao, Marianne Huchard, Therese Libourel, and Cyril Roume. Evaluating and optimizing factorization in inheritance hierarchies. In Proceedings of the Inheritance Workshop at ECOOP 2002, Malaga, Spain, June 2002

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

[4] 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

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

[6] 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

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

[8] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: A tool for using software libraries. ACM Transactions on Software Engineering and Methodology, 4(2):146–170, 1995