CONTI'2006
The 7th INTERNATIONAL CONFERENCE ON TECHNICAL INFORMATICS, 8 - 9 June 2006, TIMISOARA, ROMANIA

Reverse Inheritance Features Applied in Coding Java Mobile Applications

Smaranda-Claudia Chirila[*] and Monica Ruzsilla[*] and Ciprian-Bogdan Chirilă[*]

[*] Department of Computer Science, University "Politehnica" of Timişoara,
Faculty of Automation and Computer Science, Vasile Parvan no 2, Timişoara, Romania
Phone: (40-256) 403261, Fax: (40-256) 403216, E-mail: smaranda.chirila@siemensvdo.com,
monomi7@yahoo.com, chirila@cs.utt.ro WWW: http://www.cs.utt.ro/~chirila

Abstract ***– Reverse inheritance is a kind of inheritance where the subclasses are created first and the superclass afterwards. Reverse inheritance can be used as a composing mechanism for classes in the context of mobile coding. Some features of reverse inheritance can be used to achieve systematic coding and restricted reusability. The foster class, which is the superclass of the reverse inheritance class relationship, can be used to encapsulate particularities related to a certain mobile vendor.***

Keywords: ***reverse inheritance, class hierarchy reorganization, composing mechanisms, aspects, mobile technology***

I. INTRODUCTION

One of the most important factors on which the software quality depends is reusability. Inheritance is one way to achieve class reusability in object-oriented systems. A very close concept to the concept of inheritance is the reverse relationship, namely reverse inheritance. In this paper we present several use cases in which reverse inheritance can be used in writing Java code in the context of mobile phones programming. In our days anybody can write applications on their mobile phone. This facility is possible because the operating system of the mobiles have features that allow downloading and running Java applications. So if one has the know how to write an application for his mobile he can easily transfer it from his computer to his cell phone.

A. Reverse Inheritance

The basic idea of reverse inheritance class relationship is the generalization abstraction [8], which enables a set of individual objects to be thought generically as a single named object. It is considered to be the most important mechanism for conceptualizing the real world. Generalization helps the goal of uniform treatment for objects in models of the real world. Generalization can be defined in terms of intension and extension of a class. The intension of a class is the set of properties that defines it. The extension of a class we mean all the objects that include those properties. A class C_{general} is a single generalization of a class C if all members of $C^{\text{extension}}$ are members also in or belong to $C^{extension}_{general}$ [6]. A class is a multiple generalization of a set of other classes if it is a single generalization of every class in the set. A definition of reverse inheritance given by Pedersen [6] states that a class G can be defined as a generalization of $A_1,A_2,...,A_n$ previously defined classes. If the value of n is 1 then we discuss about single generalization, otherwise about multiple generalization. Informally, it can be defined as another model of inheritance where the subclass exists and the superclass is constructed afterwards.

The source class of reverse inheritance is called generalizing class [7] or as foster class [5]. Reverse inheritance should have an appropriate symmetrical semantics in order to produce the same class hierarchy structure having the behavior as if was defined by direct inheritance. So, this class will include all the features (attributes and methods) that are common to these classes.

Reverse inheritance is a more natural way for designing class hierarchies [6]. When modeling classes, it is considered that it is more natural to design each class with its own features and only then to notice commonalities and factor them in a common superclass. This will help avoiding data and code duplication. In some applications classes belonging to different contexts need to be reused together. They can have even common functionalities which could be factored in one place to avoid duplication. Some classes in object-oriented systems exhibit a great quantity of behavior. Maybe in some contexts only a subset of them needs to be reused. This can be achieved with the help of reverse inheritance very easy.

B. Aspect Oriented Programming

Aspect oriented programming [4] is a separation of concerns model based on object-oriented paradigm. It deals with crosscutting concerns which can not be well separated by pure object technology. The majority of object-oriented systems are composed out of crosscutting concerns dispersed over several modules. By concern it is meant a concept, a goal in the context of a given domain. For example a concern in the context of debugging a software system would be the logging operations. Another functionality, which can be viewed as a crosscutting concern, needed in the context of objects, is persistence.

There are several concepts of object-oriented programming which facilitate the separation of concerns. First, the abstraction principle implies the creation of separate classes for each concept from the real world [1]. On the

other hand the information hiding principle allows interface separation from implementation. Inheritance and delegation are ways of composing behavior. In the context of inheritance, the behavior of the subclass is composed with the behavior of the superclass [1].
Each application has a main part where the basic functionality is captured. This part is supposed to be written in a language that suits better to the application domain. Then each cross-cutting aspects are described using several specialized languages. All these programs are taken by the weaver and it produces the output code. The main property of this methodology is aspectual decomposition. Thus, the aspectually decomposed program is easier to develop and to maintain.

C. Paper Structure

The paper structure is presented next. In section two the anatomy of a foster class is described. In the third section we discuss the way reverse inheritance can improve the coding process for mobiles at class level. In forth section we present the possibilities given by reverse inheritance in coding methods. In section five conclusions are drawn and future works are stated.

II. FOSTER CLASS MODEL IN A NUTSHELL

The foster class is the equivalent of the superclass in the context of reverse inheritance. The presented model is designed for Java programming language [3].

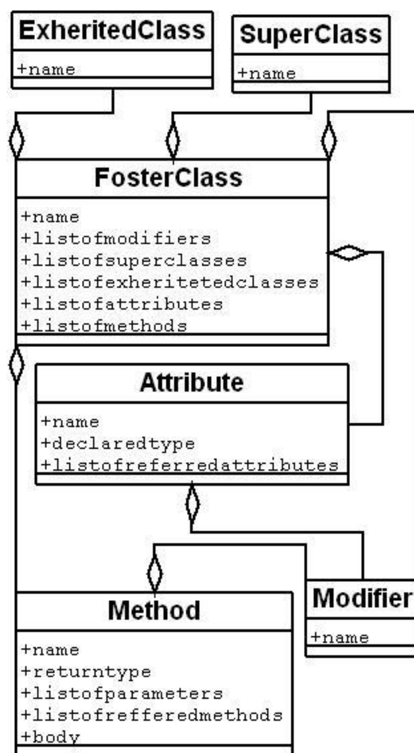

Fig. 1. Foster Class Anatomy

In figure 1 the structure of a foster class is presented. Following all the components included by the anatomy of a foster class will be explained. The name presumes a fully qualified name including the package the class lives in. The list of modifiers must include 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 because 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).
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 UML diagram of figure 2:

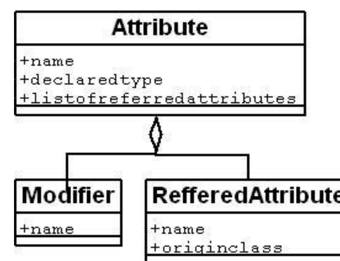

Fig. 2. Attribute Anatomy (factored or non-factored)

Name is the simple name of the attribute, declared type is the type used in its 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. 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. 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, may be omitted in case they have the same name and the same declaration type. They are considerred to be implicit.
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:

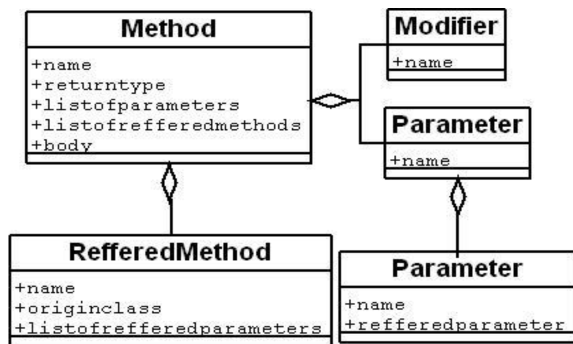

Fig. 3. Method Anatomy (factored or non-factored)

The list of parameters is a common entity which belongs to the structure of the method. The parameter contains information like: its name and its declaration type. The same reason of referred classes independent development, in the context of attributes, can be mentioned also in the context of referred methods. 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. 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. The model of the foster class has a set of rules that must be checked before declaring a certain foster class consistent. The foster class model is the blue print for creating foster classes.

III. COMPOSING CLASSES

In coding mobile phone applications, there are **common operations** which imply the same code in several similar projects. For example in gaming applications, such operations are: the loading of a frame image by pieces, the loading or playing sounds. In this kind of applications classes are not written in the real sense of object-oriented programming paradigm. Practically, due to memory restrictions, classes are used to group altogether sets of methods and constants which contain the logic of the application.

Sometimes, in an application, a part of a class could be used to create future new classes. In this use case it is proposed to facilitate better class design by decomposing existing classes and creating new ones by recomposing with the decomposed parts.

For example taking three classes: class *A* with attribute *att1*, method *meth1()*, class *B* with attribute *att2*, method *meth2()* and class *C* with attribute *att3* and method *meth3()*. In such a hierarchy features from classes *A* and *B* are combined in class *C,* the hierarchy being equivalent with a class which includes all features from all the three classes. This way we can create new classes with exactly the features we want from already existing classes. In figure 4 this example is presented.

In this case the semantics of reverse inheritance includes the semantics of multiple inheritance between classes which is not present in Java [2].

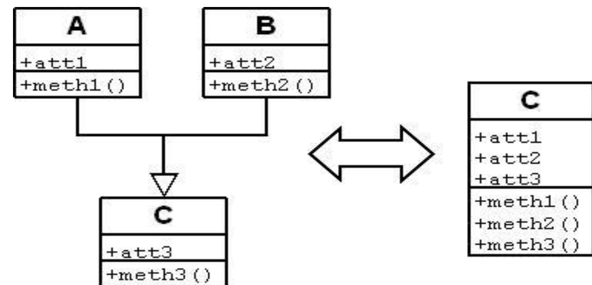

Fig. 4. Composing Classes

IV. COMPOSING METHODS

There are software companies that are specialized in designing mobile applications so there are invented special techniques to develop them. These techniques were required because in order to program applications for a wide range of phones (from a wide range of vendors) which have different particularities.

For example, the mobile phone Nokia S40 class allows only 5 songs to be pre-fetched because of the memory restrictions while Nokia S60 class has no such limitations. In this context some specific instructions are needed in order to manage the memory. So, it is necessary **to combine vendor specific code with general use code** for several types of mobile phones. An immediate solution implies the use of compiler directives to separate the code specific to each mobile vendor from the general code of the application. This technique makes the code very hard to understand, maintain and reuse.

Reverse inheritance can help in this matter. Special classes containing vendor specific code organized in methods can be combined by reverse inheritance with the core of the application. For example a snippet of code using compiler directives used to mix implementations for more vendor types is the following:

```
class Application {
m() {
        #ifdef  VENDOR:Nokia
                InstructionSet1;
        #endif
        InstructionSet2;
        #ifdef  VENDOR:Sharp
                InstructionSet3;
        #endif
}}
```

At compile time if we choose the vendor Nokia method *m()* will be composed by *InstructionSet1* and *InstructionSet2*, otherwise when choosing Sharp it will contain *InstructionSet2* and *InstructionSet3*. This is a way to

manage multiple projects starting from the same source code.
If we would like to reuse only the code dedicated to Nokia or Sharp in this way, it is impossible. But if we would have two classes Nokia and Sharp, having a method *m()* including *InstructionSet1* in class Nokia and a method *m()* including *InstructionSet3* in class Sharp, this would be very much possible. Starting from the two vendor specific classes Nokia and Sharp, and using reverse inheritance we show that we can compose those classes to obtain the same code like in the snippet having preprocessor directives.
The reverse inheritance solution implies the following design: the common code *InstructionSet2* has to be factored in the *m()* method of class *Application*, each specific vendor code should be encapsulated in the *m()* method of the *Nokia* and *Sharp* classes. The specific vendor *m()* method should call the common code from the subclass (see figure 5). This is facilitated by the *inferior()* call feature of reverse inheritance. The *inferior()* call is the opposite of the *super()* call in the context of ordinary inheritance.

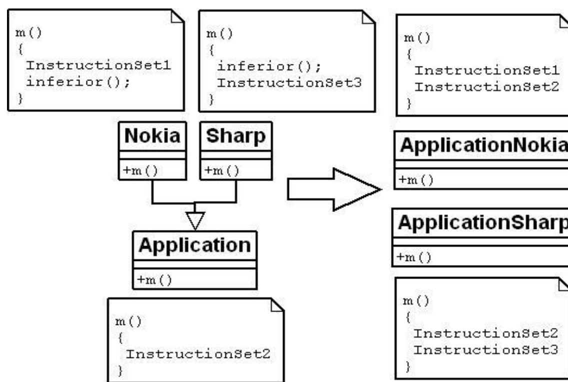

Fig. 5. Composing Methods

V. CONCLUSIONS AND FUTURE WORKS

We can conclude that each foster class will contain all the functionalities required by the corresponding vendor, thus foster classes represent a specific concern, like in AOP. So using foster classes, the code will be more readable and reusable.
There are also some limitations of the reverse inheritance based solution in coding mobile software. A first restriction is given in the following example:

```
m()
{
    InstructionSet1
    #ifdef  VENDOR:Nokia
    ...
    #endif
    InstructionSet2
    #ifdef  VENDOR:Sharp
    ...
    #endif
    InstructionSet3
}
```

The main problem is that *InstructionSet1* comes before the specific part for Nokia and *InstructionSet3* comes after the specific part for Sharp vendor. In this case where the vendor code is enclosed before and after with application code, reverse inheritance can not be applied. The *inferior()* mechanism can not be used to provide the desired behavior. One possible solution is to split method *m()* so the application code to be able to be invoked by the *inferior()* mechanism.
Another problem arises in the situation of code containing nested compilation directives, like in the following example:

```
#ifdef ...
    ...
    #ifdef ...
     ...
    #endif
    ...
#endif
```

In this case reverse inheritance based solutions are possible, but the implied semantics of the class relationship it would be too complex and hard to use in practice.

ACKNOWLEDGMENTS

This paper is a natural consequence of the cooperation between Politehnica University of Timişoara and University of Nice, France. We would like to thank M.C. Philippe Lahire and M.C. Pierre Crescenzo for the opportunity of developing this collaboration.

REFERENCES

[1] M. Aksit. Separation and composition of concerns in the object oriented model. ACM Comput. Surv., 28(4es):148, 1996.
[2] K. Arnold and J. Gosling. The Java Programming Language. Sun Microsystems, 3rd edition, USA, 2000.
[3] 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.
[4] Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, Proceedings European Conference on Object-Oriented Programming, volume 1241, pages 220-242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
[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] 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, Malaga, Spain, June 2002.
[8] John Miles Smith and Diane C.P. Smith. Database Abstractions: Aggregation and Generalization. In ACM Transactions on Database Systems, volume 2, pages 105-133, June 1977.