THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-04-03

Efficient Implementation of Design Patterns in Java Programs

Alonso Marquez

February 2004

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

Efficient implementation of design patterns in Java Programs *

Alonso Marquez

Department of Computer Science
Australian National University
Canberra ACT 0200 Australia

Email: Alonso.Marquez@cs.anu.edu.au

Abstract

Designing and implementing generic software components using *design patterns* like *proxy* and *facade* [Gamma et al. 1994] is commonly advocated in most application areas. Design patterns and genericity are language design techniques created to provide better semantic abstractions boundaries. However, most programming language implementations have give very little support to techniques that remove these abstraction boundaries to the extent necessary to permit efficient execution.

We present a general technique for optimization of Java programs that use any sort of data and/or behavior delegation. We define a group of basic program transformations that preserve the program semantics. They provide a foundation for more complex program transformations such as *class inlining* and *class proxying* with a broad range of applications in program optimization and semantic extension. We present several real applications including *Orthogonal Persistence* and automatic optimization of design patterns implemented using these techniques.

1 Introduction

Design patterns [Gamma et al. 1994] are becoming increasingly popular as a way to describe solutions to general design problems. They provide an abstract model of object behavior. Most design patterns are based on some sort of delegation: we wrap objects to modify or enhance or their behavior. However, any delegation approach involves an abstraction overhead. It is frequently to implement methods with only trivial behavior, e.g. forwarding a message to another object. The necessary downside of data and behavior delegation is an additional overhead in memory and runtime. Object allocation costs increase because every service object is allocated separately from the container. This dispersed allocation also affects the cache performance. Access to every field or method inside the service variables requires one more *pointer dereference*. This simple pointer dereference can increase by more than ten percent the number of byte-code instructions executed in methods like `get` in the class `java.util.Vector`.

We present a group of algorithms that `inline` and `specialize` a class hierarchy with respect to its usage in a program and create distinct classes for fields and variables that use the original class in different ways.

The benefits of *class inlining* and *class specialization* can be manifold:

- Removal of object and/or data delegation overhead.
- The space requirements of a program are reduced at run-time, because objects no longer contain unnecessary members. It also tends to cluster object data together and reduce network overhead.
- class inlining and class specialization can create new opportunities for general-purpose optimizations such as method inlining or application specific optimizations like optimization of read and write barriers in orthogonally persistent programs [Hosking et al. 1998].

Although we expect user defined pattern classes to be the main use of these class specialization techniques, they could be also applied to optimize Java libraries like Swing. Many of them use data and behavior delegation heavily to create portable libraries.

* This work was carried within the Cooperative Research Centre for Advanced Computational Systems established under the Australian Government's Cooperative Research Centres Program.

1.1 Motivating examples

A common design pattern is proxy [Gamma et al. 1994]. A proxy class receives method calls on behalf of another service class. For example, the class `java.util.Vector` receives `fetch` and `put` operations that are delegated to an internal array `elementData`.

```
public class java.util.Vector extends java.util.AbstractList implements
             java.util.List, java.lang.Cloneable, java.io.Serializable  {
    protected java.lang.Object elementData[];
    protected int elementCount;
    ...
    public synchronized final void addElement(java.lang.Object);
    ...
    public synchronized final java.lang.Object get(int);
}
```

Figure 1: The Vector class as a Proxy of an array of objects

Though the `Vector` class doesn't directly provide the storage service, the proxy class manages this service, given the capability to automatically relocate a bigger array when needed. Another example is the class `javax.swing.ImageIcon` that could be described as a proxy of the class `java.awt.Image`. Inheritance is not appropriate in this case because the service class possesses methods that conflict with the intended use of the new class. In general, inheritance is not appropriate when you attempt to extend a final method or class or you want to hide or modify the type of inherited attributes. A proxy can use the attributes of a service object while ensuring that the service object and its attributes remain private. This hiding property is common to most of design patterns and will give us the opportunity to make many useful optimizations.

We can reduce the proxy overhead if we inline the attributes of the service class inside the proxy class. For example, suppose we have a proxy class `Test` that use a service class `StringStack`. `StringStack` needs a proxy over `java.util.Stack` to modify the return type of some common methods (See figure 2).

```
public class Test {
    private StringStack myStack=new StringStack();
    String pop(int n) { ..myStack.pop()... }   // pop the element n objects down ths stack
    void push(int n,String newValue) { ... }   // put a new value n elements down the stack
    Test scramble(int n) {
        Test result=new Test();
        for(int i=0;i<n;i++) {
            result=pop(n-i);
            test.push(i,result);
        }
        return result;
    }
}
final class StringStack {
    private java.util.Stack stack=new java.util.Stack();
    public synchronized String pop() {
        return(String)stack.pop();
    }
    public void push(String s) {
        stack.push(s);
    }
}
```

Figure 2: A Test program that use a stack of Strings

A more efficient version of `Test` is obtained by folding the attributes of the field `myStack` inside class `Test`. Although we can inline all attributes of service field `stack` as well, it is simpler to extend `StringStack` from `java.util.Stack`. Methods `push` and `pop` are renamed to solve *name conflicts*(two methods with same name and signature). As result, fields `stack`, `myStack` and pointer dereferences in renamed methods `push(`**`int`**`)`, `pop(`**`int`**`)`, `pushNew` and `popNew` disappear (See figure 3). This simple transformation improves speed by 25 percent.

```
public class Test extends java.util.Stack {
    final String popNew() {        // Renamed to solve name conflict
        return(String)pop();
    }
    final void pushNew(String s) { // Renamed to solve name conflict
        push(s);
    }
    String pop(int n) { ..pop(). }
    void push(int n,String newValue) { ... }
    Test scramble(int n) {
        Test result=new Test();
        for(int i=0;i<n;i++) {
            result=pop(n-i);
            test.push(i,result);
        }
        return result;
    }
}
```

Figure 3: An optimized version of class Test using class inlining

```
public class Test {
    private static int capacityIncrement=0;
    private Object elementData[]=new Object[10];
    private int elementCount;
    final Object elementAt(int pos) throws ArrayIndexOutOfBoundsException { ..}
    private final void removeElementAt(int pos) throws ArrayIndexOutOfBoundsException { ..}
    private final void addElement(Object o) {..}
    final String popNew() {
        ... removeElementAt() ...
    }
    final Object pushNew(String s) {
        addElement(s);
        return s;
    }
    String pop(int n) { ..pop()... }
    void push(int n,String newValue) { ... }
    Test scramble(int n) {
        Test result=new Test();
        for(int i=0;i<n;i++) {
            result=pop(n-i);
            test.push(i,result);
        }
        return result;
    }
}
```

Figure 4: A more optimized version of class Test using class specialization

A more elaborate class optimization (class specialization) combined with class inlining useful of `java.util.Stack` and `java.util.Vector` attributes allows the removal of unused fields like `modCount` (See figure 4). This optimized version runs twice as fast as the original one.

1.2 Related work

Although the concept of attribute inlining is new in Java, it is quite common in languages likes Fortran, C, C++ and ML. For example, the 'Concert' C++ Compiler [Dolby and Chien 1998] and the *object folding* transformation [Jones and Rundensteiner 1997] define limited *field inlining* techniques over class fields. The Concert compiler can not handle structure assignment well (handling explicit references to the service object), and their technique can not be easily applied in a language like Java which does not have multiple inheritance. The object folding transformation consists of the repeated application of field inlining over non-recursive object structures. Proposal of *class specialization* over C++ programs have been made by Tip and Sweeney [1997] and Volanshi, Concel and Muller [1997]. Other papers about program transformations include extensions to the Java language such as genericity [Thorup 1997; Agesen et al. 1997; Bokowski and Dahm 1998] and role related extensions [Boyland and Catsagna 1997].

Most of these proposals are based on source code transformation [Boyland and Catsagna 1997], or modification to the Java compiler [Thorup 1997]. Finally, some proposals of type parameterization for Java [Agesen et al. 1997; Bokowski and Dahm 1998] make use of byte-code transformations at class loading time. While Bokowski and Dahm [1998] achieve this entirely through a custom class loader, Agesen et al. [1997] use both a custom class loader and a modified compiler.

1.3 Paper organization

The formal definition of basic class optimizations (class folding, class unfolding and specialization) is described in section 2. In section 3 we present some useful transformations built using the basic set of transformations. Section 4 provides some details about the actual implementation. Section 5 discusses several examples of the application of these techniques. Section 6 concludes the paper.

2 Basic Transformations

2.1 'JavaBean' compliance

This transformation provides a significant decoupling of the implementation from the form of an object. The 'JavaBeans' specification [Sun Microsystems 1997] describes the form and style of a JavaBeans compliant object. The form includes a description of the methods for retrieving and setting the properties (fields) of a JavaBeans compliant object (section 8.3 of [Sun Microsystems 1997]). By limiting direct access to fields to methods executing over the object owning those fields, the JavaBeans specification enhances the level of object encapsulation. We define a similar transformation to facilitate the definition of other program transformations.

Definition 1 (JavaBean transformation) *Each non private field declaration of the form:*

```
<protection> [static] <type> <field>;
```

It is replaced by

```
private [static] <type> <field>;

<protection> [static] final <type> get$<classname>$<field>() { return <field>; }
<protection> [static] final void set$<classname>$<field>(<type> arg$<field>) {
  <field> = arg$<field>;
}
```

To accommodate this transformation all operations over fields need to be transformed. All getfield *and* putfield*instructions are replaced by an* invokevirtual *to either the get or set access-or method (except inside get and set access-or methods). In a similar way,* getstatic *and* putstatic *instructions are replaced by* invokestatic *to either the get or set access-or method associated with a static field. This replacement (along with the correct constant pool method descriptor) is stack neutral.*

2.2 class unfolding

This transformation introduces data and/or behavior delegation that can be used for complex semantic extensions moving a group of attributes from a class to another. It can reduce memory overhead of usually null fields but introduces an additional reference overhead over all unfolded attributes.

Definition 2 (Class unfolding) *Given a 'JavaBean' compliant, non-abstract class C, a possible new class 'C$Service' and a set of attributes Lattrib,*

- *Unfolded attributes are added to 'C$Service'. If unfolded attributes reference non-unfolded attributes, a variable 'proxy$' of type C is created in 'C$Service' to invoke those attributes.*
- *A private field 'service$' of type 'C$Service' is created and initialized in every local constructor. New proxy methods are created for each unfolded method. These proxy methods call the corresponding method over field 'service$'.*

Definition 3 (Hierarchical unfolding) *Given a 'JavaBean' compliant class C with non-abstract superclass SC, a private field variable 'super$' of type CS is created and initialized in every local constructor using the previous super constructor invocation. New proxy methods are created for each inherited method. These proxy methods call the corresponding method over field 'super$'. Any reference to any inherited attribute is replaced by an explicit invocation of new proxy methods.*

Hierarchical unfolding can be seen as a particular case of class unfolding where the implicit variable **`super`** is unfolded. It may be used to simulate multiple inheritance because the hierarchical unfolded class keep all the inherited attributes but inherit directly from `java.lang.Object`. In this paper is used to enable the inlining of inherited attributes.

2.3 Class specialization

This generates a specialized clone of a class to be used in a specific context (specializing methods and taking away useless attributes). Class specialization can lead to a considerable performance gain, by eliminating from the specialized class useless attributes and bytecode instructions.

```
public class C {
    private Service f=new Service(10);
    void m(int inc) {
        f.setValue(new Integer(f.getValue().intValue()+inc));
    }
}
class Service extends SSuper {
    private Integer value;
    private int initialValue;
    public Service(int iv) {
        initialValue=iv;
        value=new Integer(iv);
    }
    public Integer getValue() { .. }
    public void setValue(Integer value) { ... }
    OTHER_SERVICE_ATTRIBUTES;
}
```

Figure 5: Class Service before specialization

Definition 4 (Visible object instance) *An instance of a class P is visible if it may be returned by a function, cast to a class, tested as instance of a class or passed as a parameter.*

Definition 5 (Visible Attribute or method variable) *A private field or method variable is visible with respect to a class P if may receive or return a reference to a visible object instance of P. For example, the field f in class C is not visible even its subfield value is visible (See figure 5).*

A field can be non-visible even if its subfields are visible. We can only invoke instance methods over a non-visible field or method variable. A class P is non-visible with respect to a group of attributes of another class if each attribute in the group is non-visible.

Definition 6 (Useless attribute) *An attribute A is useless with respect to a field F if it is not required by any use of F (does not exists a sequence of calls that invoke or get a value from the the attribute).*

A similar definition applies to method variables. A field or method variable can be useless even if there exists useful method that updates its value. Useless methods and fields are quite common when a subclass overwrites the semantics of an inherited method or field. The hierarchical unfolding and class folding process may make some of these useless attributes private, given the opportunity to remove them.

Definition 7 (Class specialization) *Given a 'JavaBean' compliant class C with a non-visible field or method variable F,*

- *A new final class 'P$clone' of P is created with same local instance fields, local instance methods, local inner classes and same parents. Static references to local attributes in P continue to reference P.*
- *The local variable F is made of type 'P$clone'.*
- *All useless local attributes and method variables are erased from the class. Any instruction that only modifies a useless field or method variable is taken away as well. Other program specialization technique such as [Volanshi, Concel and Muller 1997] can be applied as well.*

```
public class C {
    private Service f=new Service$clone();
    int m() {
        f.setValue(new Integer(f.getValue().intValue()+inc));
    }
}
final class Service$Clone extends SSuper {
    private int value=new Integer(10);
    public final int getValue() {... }
    public final void setValue(Integer value) { ... }
}
```

Figure 6: Class Service after specialization

The general idea of this transformation is, given the class C in figure 5, we will generate the class in figure 6. The `value` field and its accessor methods are the only non useless attributes in class Service.

2.4 Class folding

Replace a non-visible field for its attributes (fields and methods).

Definition 8 (Class folding) *Given a 'JavaBean' compliant class, C, with a non-visible instance field S such that S is only assigned with non new instances of class CS and CS inherits directly from `java.lang.Object`,*

- *We generate a specialized class CloneCS of class CS. If neither C nor CloneCS inherit directly from `java.lang.Object`, we apply hierarchy unfolding over CloneCS.*
- *Any signature conflict between C and CloneCS attributes is resolved, renaming CloneCS attributes with new names. Any reference to renamed attributes in C and CloneCS is renamed as well.*
- *All CloneCS constructors are added to class C as private instance methods. All other CloneCS methods are added to class C as private attributes. All references to class CloneCS and CloneCS attributes in C and CloneCS are replaced with class C and new class C attributes. All instance fields of CloneCS are inlined as well.*
- *All references inside C to service field S are replaced with references to actual class **`this`**. Any initialization of field S with a new instance of CloneCS is replaced by invoking the constructor over the actual class. Any access to the service field before the first assignment must throw a null pointer exception.*

If the service variable is a static field, a similar transformation can be defined where instance fields of service class become new static fields. If the service variable is a method variable, we can inline class attributes as new method variables. In this case, only service classes with non-primitive fields can be inlined and the new method variables will be passed as additional arguments to all inlined methods. In a similar way, a folding operation over arrays can be defined.

Definition 9 (Class hierarchy folding) *Given a 'JavaBean' compliant class C with an non-visible instance field S of type CS such that C inherit directly from `java.lang.Object`, make inherit C from CS and replace field S by **`super`**.*

Can be seen as a particular case of class folding where the service variable becomes the new **`super`**. The service class may be a subclass of another class, but the proxy class must inherit directly from `java.lang.Object`. It can not be applied if introduce name conflicts. Figure 3 show the result of apply class folding over non-visible field `myStack` and Hierarchical folding over non-visible field `stack` of classes `Test` and `StringStack` in figure 2.

2.5 Justification

'JavaBean' compliance, class unfolding, class specialization and class folding of non-visible fields and method variables are all semantic-preserving program transformations. Additional constraints are necessary to preserve the semantic in hierarchical folding and unfolding. Due to space limitations, we only state the properties here, without formal proof.

3 Issues in automatic generalization and specialization of OO Programs

We are interested in program transformations that preserve the original program semantics. Therefore, a transformed class must respect the API and the semantics of the original class' attributes (it must maintain the API contract). However, a transformation could add new attributes so long as they do not create conflicts. We are interested in two sorts of transformations: global transformations that are applied over the whole program and local transformations inside a class that become invisible for the rest of the program. In many cases, a transformation can be applied either locally or globally.

A classical example of local transformation is the inlining of final methods. A similar but less common transformation is method unfolding, where a code segment is replaced by an invocation of a new private method with the semantics of unfolded code. Both transformations preserve the API and the semantics. However, inlining and method unfolding can be applied globally as well.

3.1 Class Inlining

The class inlining is based on the inlining of instance attributes of a service field. This transformation can be applied over more of one field in a class. It can even be applied to array or method fields. The attribute inlining can include new proxy fields. This inlining process is usually repeated with the new included fields as well if the new fields are not instances of the service class.

Definition 10 (Class inlining) *For each 'JavaBean' compliant user class P, for each non-visible field or method variable F such that F is only assigned with non new instances of class CF,*

- *If P directly inherits from `java.lang.Object`, apply hierarchy folding over F,*
- *else if the result of clone CF ('Clone$CF') directly inherits from `java.lang.Object`, apply class folding over F.*
- *else apply hierarchical unfolding over 'clone$CF' and class folding over F.*

3.2 Class proxying

This transformation allows the introduction of complex cases of data or behavior delegation. It is a global transformation where the behavior of class is replicated in a new class. It is applied when the results of semantic extensions using inheritance are inadequate.

Definition 11 (Class proxying) *Given a 'JavaBean' compliant non-abstract class C, a list of C methods ProxyMethods and a possible empty class ProxyClass,*

- *If ProxyClass does not have an attribute 'service$' of type C, we add a new one. Any constructor of ProxyClass is modified to include the initialization of 'service$' field.*
- *For each method m in ProxyMethod without name conflicts with ProxyClass, a method with same name, signature and return type is generated in ProxyClass. This new method will delegate the call to class C using 'service$' field.*

The transformation ensures that any new attribute, superclass or interface introduced in ProxyClass will replace any previous definition in C. Examples of use are the automatic generation of Corba or 'Enterprise JavaBeans' interfaces.

3.3 semantic extension transformation

The semantic extension transformation can be see as a particular case of proxying a class where all references to a class are replaced by a new proxy class. Additional transformations over class references and introspection invocation are necessary.

Definition 12 (Semantic extension of a class) *Given a 'JavaBean' compliant non-abstract class, C, and an extension class ExtClass that extends C superclass,*

- *Let C' be the result of proxying class C using ExtClass and C attributes.*
- *Every reference to class C is replaced by C'.*
- *Every invocation to system class methods like `getClass()` is delegated to a similar call over the 'service$' field..*

The semantic extension transformation can be used as a change definition language for schema evolution, where the ProxyClass specifies any new or redefined attribute. Other examples are the definition of semantic extensions over final system classes like `java.lang.String`.

4 The Program Transformation library

We have implemented a library of filters over the Java representation of a class. Rules that match patterns over attribute declarations and byte-code can be easily defined. Transformations like JavaBeans compliance and proxy generation have been implemented this way. The 'filter' pattern used in the library enables the generation of more complex transformations like Class Inlining by composition of program transformations. We have implemented the framework using the 'CFParse' library that provides facilities for class file parsing and basic class transformations (See [alphawork 1999]). In order to avoid any modification to actual Java compilers, we defined a specialized class loader that modifies the way the JVM loads byte-code files. This technique of dynamic byte-code transformation is related to the load-time expansion technique used in the proposal by Agesen et al. [1997] and Bokowski and Dahm [1998]of Sun and Stanford University for parametric polymorphism. This portable approach to program transformations in Java has several key advantages. Any Java to byte-code compiler can be used, and it does not rely on Java source being preprocessed. This contrasts with techniques that extend the Java language syntax, such as those described in [Agesen et al. 1997; Thorup 1997] et al. Because the transformations are applied at the byte-code level, the framework is not tightly coupled to a particular JVM. This gives the user the freedom to use the most suitable JVM and leverage rapid advances in Java technology immediately. Our portable approach therefore has a significant advantage over approaches to semantic extension which modify the JVM itself and so are tied-down to a particular technology.

The class transformation overhead for most applications is not significant. The initial overhead of applying the byte-code transformation to a hundred or so classes is only a few seconds. The runtime overhead is dependent upon the transformations applied. The 'JavaBeans compliance' transformation does not generate any real overhead. This is due to the accessor methods being final and the consequent inlining of these methods by the JVM. The replacement of a class for a subclass of that class adds little to the overhead. However, the generation of proxies or deeper semantic changes like modification of instructions semantic can carry an important overhead.

5 Practical use of class proxying and class inlining

5.1 Class Inlining application

This technique can be used to optimize the object version of design patterns like adapter, bridge, proxy, facade, chain of responsibility, command, mediator and iterator. A first implementation of the class inlining transformation over non-visible fields and method variables has been made. Non-visible fields are automatically detected and inlined. Preliminary statistics have shown that speed improvement as much as doubling can be achieved by applying this transformation. Moreover, memory use is reduced by 5% to 10%. Most of the speed improvement is result of the reduction in the number of objects created. This result is quite consistent with Java benchmarks that show that object creation is quite expensive in Java, up to 25 times as expensive as C++.

5.2 Semantic Extensions

The semantic extension of classes in Java can be viewed as similar to the process of schema evolution where the semantics of some classes are modified. Using this analogy, our interest lies in a sort of 'incremental evolution' where the original semantics form the basis for the extended semantic. Our objective therefore has a similarity to proposals for schema evolution in object oriented databases where special DDL[1] operators are defined that represent high level transformations such splitting a class or modifying the class hierarchy. The need for a family of transformations that allow such incremental extensions to be readily introduced a global and consistent way is clear. We have defined a syntactic convention for specifying the most common transformations using an independently defined class (Extension Class) that specifies all semantic changes.The name of the special extension class must be terminated with a '$' and with '.'s replaced by '$'s (e.g. `java.lang.Object` become `java$lang$Object$`.

We can define or replace attributes of the original class given the definition in the Extended Class. We also have extended this name convention to define special methods that modify the language semantics itself, like the definition of instruction triggers that are fired before or after execute an instruction. This transformation can be implemented using the semantic extension transformation defined in 3.3.

5.2.1 semantic extensions framework

We have implemented the proposed semantic extension language as a framework for semi-dynamic semantic transformation of Java programs [2]. Our framework allows for both semantic extension of methods and the inclusion of special 'triggers' (similar in concept to database triggers) that are activated on the occurrence of particular events such as the execution of `getfield` or `putfield` byte-codes.

5.3 Code Profiling and Instrumentation

Another important application for the class of transformations described here is in trapping method invocations. Such a technique can be used to transparently instrument byte-codes for call profiling. This can be applied generally (trapping all method invocations) or selectively (to profile certain classes or methods).

5.4 A portable implementation of Orthogonally Persistent Java

Orthogonally persistent systems are distinguished from other persistent systems such as object databases by an orthogonality between data use and data persistence. This orthogonality comes as the product of the application of the following principles of persistence [Atkinson and Morrison 1995]: the form of a program is independent of the longevity of the data which it manipulates, all data types should be allowed the full range of persistence, irrespective of their type and the choice of how to identify and provide persistent objects is orthogonal to the universe of discourse of the system.

These principles impart a transparency of persistence from the perspective of the programming language, which obviates the need for programmers to maintain mappings between persistent and transient data. The same code will thus operate over persistent and transient data without distinction. The extension semantic framework has a natural application in extending the semantics of Java to orthogonal persistence. We have already built two implementations of Orthogonally Persistent Java (OPJ) using the semantic extension framework. The first of these is characterized by the use of object 'shells' and eager swizzling and lazy reification policies with small read barrier overhead. The second uses the concept of 'façades' to remove read barriers and has lazy swizzling and slightly eager faulting and reification policies. Both approaches are novel insofar as they represent the first portable implementations of Orthogonally Persistent Java, and each explores quite different implementation strategies [3].

[1]Data Description Language.

[2]This work has been submitted for publication.

[3]This work has been submitted for publication.

The use of class loading time program transformations lends our Orthogonally Persistent Java a number of important qualities, including simplicity, portability and a clean model of persistence. Significantly, our implementation is efficient, outperforming PJama, a well-known orthogonally persistent Java, which is based on a modified virtual machine.

5.5 Orthogonally Object Versioning (OOV) framework

The Orthogonally Object Versioning framework is based on orthogonally object versioning and it was built on top of our OPJ prototype. The versioning system is independent of the object type. All the object types are versioned except inmutable objects such as Strings. An object version that can be reached from any version of any persistent root is made persistent. The form of a program is the same whether it manipulate the last version or any previous version of an object. It is also possible to access and manipulate multiple object versions at the same time using the new version methods.

6 Conclusions and Future Work

We have introduced four program transformations: JavaBean compliance, class specialization, class and hierarchical folding and unfolding. We have showed how many useful transformations like proxy generation and class inlining can be decomposed using these atomic transformations. We believe that these elementary transformations can be used to generate many other interesting program transformations. We implemented a library of class filters that can be used for the automatic generation of proxies, inlining of attributes and class specialization in Java. A first implementation has been built using byte-code transformation at class loading time using a specialized class loader. This solution is portable and independent of JVM or compiler. We have used this library for building a Semantic Extension Framework that implements a declarative class manipulation language that allows any class to be semantically extended, even if the class is final or is a system class. This framework represents a high level, powerful and portable means of semantically extending Java classes. It can be applied for implementing any behavior that is orthogonal to the purpose of the class and can be specified independently. We finally show several practical applications of this framework to program optimization and complex semantic extensions. Several complex semantic extensions, like a portable implementation of Orthogonally Persistent Java and an Orthogonally Versioning framework have been built using the semantic extension framework.

The class folding, class unfolding and class specialization transformations provide a foundation for a complete family of advanced program transformations like class inlining and proxying with a broad range of applications in program optimization and semantic extensions. Significantly, a number of real applications like Orthogonal Persistence, object versioning, schema evolution and automatic optimization of programs has been presented and implemented using these techniques. A Preliminary benchmark has show that automatic class inlining can provide good improvements in performance and memory allocation. In some cases speed improvements of more of 100 % have been obtained through inlining and specializing JDK library classes.

We intend to continue the practical application of the semantic extension framework. We have started to implementing a declarative rule oriented language for defining more complex program transformations like partial evaluation of programs. We expect to finish the implementation of the automatic class inlining system and to test performance and memory usage improvements in real programs.

Bibliography

ALPHAWORK. 1999. AlphaWork. http://www.alphaWorks.ibm.com/tech/cfparse.

AGESEN, O., FREUND, S. N., AND MITCHELL, J. C. 1997. Adding type parameterization to the Java language. In *OOPSLA'97, Proceedings on the 1997 Conference on Object-Oriented Programming Systems, Languages, and Applications*, Volume 32 of *SIGPLAN Notices* (Atlanta, GA, U.S.A., October 5–9 1997), pp. 49–65. ACM.

ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent systems. *The VLDB Journal 4,* 3 (July), 319–402.

BOKOWSKI, B. AND DAHM, M. 1998. Poor man's genericity for Java. In *Proceedings of JIT'98* (Frankfurt am Main Germany, November 12–13 1998). Springer Verlag.

BOYLAND, J. AND CATSAGNA, G. 1997. Parasitic methods: An implementation of multi-methods for Java. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages &*

Applications (OOPSLA '97), Atlanta, Georgia, October 5-9, 199, Volume 32 of *SIGPLAN Notices* (October 1997), pp. 66–76. ACM Press.

DOLBY, J. AND CHIEN, A. 1998. An Evaluation of Automatic Object Inline Allocation Techniques In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, B.C., Canada, October ?, 199*, Volume ? of *SIGPLAN Notices* (October 1998), pp. ?–?. ACM Press.

GAMMA, E., HELM, E., JOHNSON, R., AND VLISSIDES, J. 1994. Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, 1994

HOSKING, A., NYSTROM, N., CUTTS, Q., AND BRAHNMATH, K. 1998. Optimizing the read and write barriers for orthogonal persistence. In R. MORRISON, M. JORDAN, AND M. ATKINSON Eds., *Advances in Persistent Object Systems: Proceedings of the Eighth International Workshop on Persistent Object Systems, Tiburon, CA, U.S.A., August 30–September 1, 1998* (San Francisco, 1998), pp. 37–50. Morgan Kaufmann.

JONES, M., AND RUNDENSTEINER, E. 1997. View Materialization Techniques for Complex Hierarchical Objects In *Proceedings of the 1997 ACM Conference on ... (CIKM '97), Las Vegas, Nevada, ?, 199* (? 1997), pp. 222–229. ACM Press.

SUN MICROSYSTEMS. 1997. JavaBeans. API specification (24 July), Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, CA 94043.

TIP, F., AND SWEENEY, P. 1997. Class Hiearchy Specialization In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97), Atlanta, Georgia, October 5-9, 199*, Volume 32 of *SIGPLAN Notices* (October 1997), pp. 271–285. ACM Press.

THORUP, K. K. 1997. Genericity in Java with virtual types. In M. AKSIT AND S. MATSUOKA Eds., *ECCOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9–13, 1997*, Number 1241 in Lecture Notes in Computer Science (LNCS) (1997), pp. 444–471. Springer-Verlag.

VOLANSHI, E., CONCEL, C., AND MULLER, G. 1997. Declarative Specialization of Object Oriented Programs In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '97), Atlanta, Georgia, October 5-9, 199*, Volume 32 of *SIGPLAN Notices* (October 1997), pp. 286–300. ACM Press.