A Small Extension to Java for Class Refinement

Muga Nishizawa
Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
2-12-1 Ohkayama, Meguro-ku
Tokyo 152-8552, Japan
muga@csg.is.titech.ac.jp

Shigeru Chiba
Dept. of Mathematical and Computing Sciences
Tokyo Institute of Technology
2-12-1 Ohkayama, Meguro-ku
Tokyo 152-8552, Japan
chiba@is.titech.ac.jp

ABSTRACT

This paper presents an extended Java language in which users can refine a class definition to a certain degree. They can statically or dynamically redefine methods and append a new method, field, and interfaces to the class like dynamic languages. A unique feature of this language, named *GluonJ*, is that users can use a standard Java IDE (Integrated Development Environment) to exploit coding support by the IDE. This is significant for the industrial acceptability of a new language. A GluonJ program is written in standard Java with additional Java annotations. GluonJ was carefully designed so that the IDE can recognize a GluonJ program and reflect it on the coding support such as the code assist of Eclipse. Moreover, a GluonJ program never throws a runtime exception reporting that an undefined method is called. Guaranteeing this property is not straightforward because GluonJ allows users to refine a class definition at runtime.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Design

Keywords

Java, Class refinement, Programming transformation

1. INTRODUCTION

Software evolution is one of the most significant topics in the software industry. To react to altering and evolving requirements at a rapid pace, software must be extended quickly. To minimize this effort, the extensions should be implemented in a modular fashion as much as possible.

Refinement is one of the promising technologies for this. The concept of refinement is similar to mixin layers, virtual classes, aspect-oriented programming (AOP), feature-oriented programming (FOP), and so on. It is a language mechanism for extending an existing class[1]. Unlike subclassing and mixin mechanisms, it directly modifies the definition of an existing class. Thus, a client program can use a method overridden by refinement without explicitly creating an instance of the extended version of that class.

This paper proposes our extended Java language, named *GluonJ*, in which users can statically or dynamically refine the definition of an existing class. Our contribution is the pragmatic design of GluonJ's language construct for refinement. Our extension to Java is small. GluonJ uses Java annotations and thereby does not extend the lexical syntax of Java. It exploits Java's type system as much as possible. Thanks to these, a GluonJ program can be developed on a standard Java IDE (Integrated Development Environment) such as Eclipse and NetBeans. Particularly, users can enjoy the coding supports by a standard IDE even for GluonJ programming. The IDE can recognize methods appended by refinement and shows them as candidates when its users are typing a method name. A GluonJ program is compiled by a standard Java compiler. Only a special runtime system is needed to run a GluonJ program. We introduced these features for industrial acceptability.

Although GluonJ enables appending/removing a method to/from an existing class according to dynamic contexts, a GluonJ program never throws a NoSuchMethodException if it is successfully compiled and loaded. A naive implementation of dynamic-refinement might wrongly allow a client to call an unavailable method, that would be appended later by refinement but that does not yet exist. To avoid such an invalid call, which would throw a runtime exception, GluonJ requires users to follow some programming conventions. A GluonJ program satisfying these conventions never calls an unavailable method. If a program does not satisfy the conventions, they are statically detected before the program starts running. To do this, GluonJ exploits Java's type system and GluonJ's custom class loader.

2. GLUONJ

This paper proposes our extended Java language, named *GluonJ*, in which users can statically or dynamically refine the definition of an existing class. The users of GluonJ can redefine existing methods and append new methods, fields, and interfaces to an existing class. Since these changes are described in a separate component (or module), this language mechanism is useful for separation of concerns. Refinement is described with annotations within the standard Java syntax. Thus, a GluonJ program can be compiled by a standard Java compiler although compiled bytecode is transformed by GluonJ's custom class loader.

[1]The term "refinement" is generally used in the context of formal methods. However, in this paper, it is used as a language mechanism for the extension to an existing class.

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil

GluonJ allows applying and removing refinement to/from a class on demand. Naively designed dynamic-refinement may allow a call to a method that has not been appended yet or that has been already removed and then the call may cause a runtime type error. However, a GluonJ program never causes such a runtime type error as a NoSuchMethodException although it may fail an explicit type cast and throw a ClassCastException. To guarantee this property, GluonJ does not use a custom type checker. It exploits the type checking by a standard Java compiler and the verification by GluonJ's custom class loader.

Java's annotations are convenient language constructs for implementing a language extension without changing the lexical syntax of Java. For example, AspectJ5 [1] allows users to describe aspects in standard Java. An extended language construct of AspectJ5, such as a pointcut, is described by using annotations.

However, language extensions implemented with annotations are not understood by a standard Java IDE. Users must use an extended IDE for those language extensions, for example, the AspectJ plugin for the Eclipse IDE if they want to enjoy IDE support for aspects. Although GluonJ uses Java annotations for implementing a refinement mechanism, our implementation is quite compatible with an unmodified Java IDE. We have carefully designed GluonJ so that users can exploit the coding support of an unmodified Java IDE. For example, a new method appended by a refinement is listed in the available methods shown by the code assist of the Eclipse IDE.

2.1 Design overview

In GluonJ, a refinement, which represents modifications to an original class, is described as a subclass of the original class. The subclass is annotated with @Refine and called *a refinement class*. Both the original class and the refinement class are compiled by a standard Java compiler. Then GluonJ's runtime system actually modifies the definition of the original class at load time according to the specification given in the refinement class.

2.1.1 Changing an existing method

To change an existing method in an original class, users override it in a refinement class described as a subclass of the original class. For example, to change the implementation of a getPrice method in Book class, users must write the following refinement class BookLogger.

```
@Glue class Logging {
  @Refine public static class BookLogger
                      extends Book {
    @Override public int getPrice() {
      System.out.println(price);
      return super.getPrice();
} } }
```

The getPrice method and price field have been declared in the Book class already. The BookLogger class is a standard Java class that extends the original class Book and overrides the getPrice method. Note that the refinement class is a static nested class in the Logging class annotated with @Glue. Such a class is called *an @Glue class* and can group refinement classes related to each other. A refinement class can also override static methods in its original class.

A method in a refinement class can use super for invoking the methods directly declared in its original class. The semantics of super are the same as for subclassing.

Once the two classes are compiled and loaded, GluonJ's runtime system automatically modifies the original Book class. Thus, during runtime, a Book object is an instance of the modified version of Book, the behavior of which is equivalent to that of an instance of BookLogger in standard Java. The call to getPrice in the original Book class invokes the method declared in BookLogger.

From the viewpoint of a Java IDE, the definition of a refinement class is just a standard subclass. Hence, users can fully exploit coding support provided by the IDE. For example, they can add @Override to a method in a refinement class. If the name or the signature of that method is wrong and the method does not override a method in the original class, the IDE will report an error. Recall that @Override in standard Java declares that the method with this annotation overrides a method in a super class.

Although a refinement class looks like a standard subclass, to explicitly create an instance of a refinement class is prohibited in GluonJ. Creating an array of instances of a refinement class is also prohibited. On the other hand, using a refinement class as a type name is allowed. We will later discuss details of this issue.

2.1.2 Appending new members

To append a new method to an original class, users declare the new method in a refinement class extending that original class. For example, the following refinement class BookPrinter appends a new method print to the Book class:

```
@Glue class Printing {
  @Refine public static class BookPrinter
                      extends Book {
    public void print() {
      System.out.println(
        "Book[" + title + "," + price + "]");
} } }
```

The original Book class is automatically modified by GluonJ's runtime system at load time according to Printing. The behavior of instances of the modified Book class in GluonJ is equivalent to that of instances of BookPrinter in standard Java.

To call a method appended by refinement, a reference to the target object must be cast to the type of the refinement class. For example, the following code calls the print method appended by the refinement class BookPrinter above:

```
public static void printBook(Book b) {
  ((BookPrinter)b).print();
}
```

The type cast from Book to BookPrinter always succeeds while the refinement class BookPrinter is applied to the Book class. Users can understand this programming convention by analogy to down-casts in Java. To call a method in a subclass, a reference to the target object must be down-cast to the subclass's type. This type cast succeeds only if the target object is an instance of that subclass. Although a refinement class is not equivalent to a subclass, this analogy would help users understand GluonJ's semantics.

The method print appended by the refinement class is included in the list of the available methods shown by Eclipse (see Figure 1) when a user types the period at the end of the following sequence:

```
((BookPrinter)b).
```

This is because Eclipse recognizes BookPrinter as a standard class, which declares the print method under the interpretation for standard Java. Although GluonJ extends Java's semantics by using annotations, the lexical representation of a GluonJ program can be read as a standard Java program and the types recognized by this reading are almost equivalent to the types under GluonJ's semantics. We have carefully designed GluonJ to preserve this property

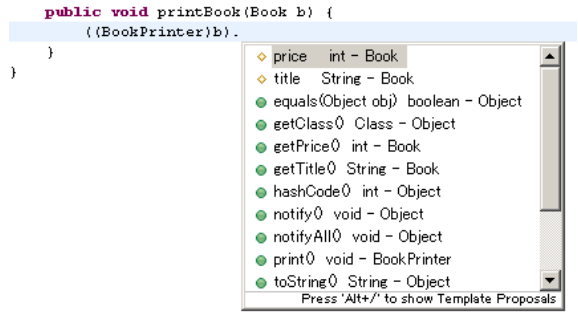

Figure 1: The code assist of Eclipse pops up a list of available methods and fields on the b variable. Not only existing methods in Book but also print() newly appended by BookPrinter are included in that list.

so that the coding support by Java IDEs can be used when writing a GluonJ program.

In GluonJ, the type cast from the type of an array of an original class to the type of an array of its refinement class succeeds while the refinement class is applied to the original class. For example, the following cast succeeds.

```
Book[] books = new Book[8];
  :
BookPrinter[] printers = (BookPrinter[])books;
```

Also, in GluonJ, to append a new interface to a class, the interface is declared in a refinement class. To call the interface method, users must cast a reference to the target object to that refinement class or its interface. Moreover, to append a new field to a class, the field can be described in its refinement class like as when subclassing. Note that the field is visible only within methods declared in the refinement class.

2.2 Dynamic refinement

GluonJ allows applying a refinement class to its original class during only limited time. Recall that a refinement class is a static nested class in a class annotated with @Glue, which can group multiple refinement classes related to each other. If the @Glue class is annotated with @Cflow, then all the refinement classes included in that @Glue class are activated during only the time specified by the @Cflow annotation. The extension by the refinement classes is effective during only that time.

The argument to @Cflow is the time during which refinements are effective. For example, the following refinement class appends the print method only while the getBookPrice method in the BookStore class is being executed.

```
@Cflow("void BookStore.getBookPrice(Book)")
@Glue public class CflowBasedPrinting {
  @Refine public static class BookPrinter
                      extends Book {
    public void print() {
      System.out.println(
          "Book[" + title + "," + price + "]");
} } }
```

The signature of a method is given as the argument to @Cflow. An @Glue class with @Cflow is active only while the specified method is executed. After that, the @Glue class is inactivated. A refinement class included in the @Glue class is applied to its original class only during the active time of the @Glue class.

A method appended by a refinement class with @Cflow is never invoked when the refinement class is inactive. Recall that, to call a method appended by a refinement class, a reference to the target object must be cast to the type of the refinement class. The type cast from the type of an original class to the type of the refinement class with @Cflow succeeds only when the refinement class is active. If an overriding method in a refinement class with @Cflow is invoked when the refinement class is inactive, the overridden method in its original class is invoked instead of the overriding one.

The following code calls the print method appended by the refinement class BookPrinter shown above:

```
public class BookStore {
  public int getBookPrice(Book b) {
      ((BookPrinter)b).print();
      return b.getPrice();
  }
  public String getBookTitle(Book b) {
      // throw ClassCastException
      ((BookPrinter)b).print();
      return b.getTitle();
} }
```

The refinement class BookPrinter is effective only during the execution of the getBookPrice method. Thus, it is not effective while the getBookTitle method is executed. Thus, the cast from Book to BookPrinter fails and causes a ClassCastException. The print method is never invoked on b.

The extension to a class by a refinement class with @Cflow is applied to all the instances of that class during a specified time. It is applied to not only existing instances but also instances created while the @Glue class is active. Note that this application is on a per-thread basis. The effect of the @Glue class is within the thread that activated the @Glue class. When other threads access the instances, the extension by the refinement class is not effective.

An @Glue class without an @Cflow annotation is always active at runtime. A refinement class included in that @Glue class is statically applied to its original class.

2.3 Restriction by @Cflow

To prevent a method appended by dynamic-refinement from being called when the method is not available, GluonJ requires users to follow the programming conventions shown below.

Let G is an @Glue class associated with @Cflow and let G include a refinement class R. R appends an interface I to its original class. If a type T is either R, I, or an array type of R or I, such as $R[]$ and $I[]$, then the following rules must be satisfied:

1. A field of the type T appears only within G.
2. The type T is not the return type or one of the parameter types of the method specified as the argument to @Cflow.
3. T is not an exception type (*i.e.* a subclass of Throwable). It is not the parameter to a catch clause.
4. The refinement class R does not override a static method in its original class.

We later discuss why these rules guarantee that a method appended by dynamic-refinement is never invoked when the method is not available. For more details, see Section 3.5.

GluonJ's runtime system checks whether or not these rules are satisfied in a given GluonJ program when the classes are loaded.

If the rules are not satisfied, the runtime system throws an exception. Otherwise, the program is successfully loaded and it will never throw a NoSuchMethodException, which reports that an unavailable method was called. NoSuchMethodException is an exception type included in the standard Java library. It is thrown when a client calls a method that does not exist.

Note that a GluonJ program may throw a ClassCastException when an object reference is cast from an original class to its refinement class. However, once the cast operation succeeds, a GluonJ program never throws a NoSuchMethodException.

2.4 Other functions

To extend a target class, GluonJ's refinements have several functions other than the above-mentioned functions. The following is a list of the other functions.

1. If there are several refinement classes for modifying the same original class, each refinement class is applied to the original class by GluonJ's runtime system in turn in the order of the priority that the users specify.

2. A refinement class allows overriding the initial value of a field in its original class. When a constructor in the original class is called, the field is set to the overridden value.

3. A refinement class allows changing and calling private methods in its original class only if it is annotated by @Privileged. A refinement with @Privileged also allows accessing private fields in its original class.

3. RUNTIME SYSTEM

To parse the arguments of annotations such as @Refine and @Cflow at runtime and then extend a GluonJ program according to the annotations, a GluonJ program runs on the top of a runtime system provided by GluonJ. The runtime system transforms the bytecode of a GluonJ program at load time. A GluonJ program must be compiled by a standard Java compiler in advance. Our implementation of refinements generates an original class definition that is the union of all refinements for that class at the bytecode level. The runtime system uses Javassist [8], which is a toolkit for transforming Java bytecode.

3.1 Appending new methods

When GluonJ finds a method that a refinement class appends to an original class, it copies the appended method to the original class at load time. A call to a method in the original class on super within the copied method is replaced with the method call on this.

When there are several refinement classes that append a new method with the same name to an original class, each appending method is copied to the original class in turn in the order of their priority. GluonJ first copies an appending method with the highest priority to its original class. Next it renames the appended method with the highest priority into a unique name and then copies a newly appending method with a lower priority to the original class. The priority of refinement classes was mentioned in Section 2.4.

3.2 Changing methods in an original class

If a refinement class includes a method changing a method existing in its original class, GluonJ first renames the overridden (changed) method in the original class into a unique name, and then copies the overriding (changing) method in the refinement class to the original class. A call to the original method on super within the overriding method is replaced with a call on this to the renamed method. A call to a non-overridden method in the original class on super within the overriding method is also replaced with the method call on this. When there are several refinement classes change the same method in its original class, each overriding method is copied to the original class according to their priority.

3.3 @Glue classes with @Cflow

When GluonJ copies an overriding method in a refinement class with @Cflow to its original class, it inserts some check codes at the beginning of the overriding method. These codes check whether or not the refinement class with @Cflow is active, and if it is not, it forwards the call to the overridden method in its original class. Since methods in multiple refinement classes with @Cflow may override a method in its original class, if the refinement class with @Cflow is not active, the check code forwards to the next method overridden by another refinement class according to their priorities. When there are several refinement classes with @Cflow that append a method with the same name to an original class, the check code is also inserted in each appending method like changing original methods.

3.4 Type names, type cast, and instanceof

GluonJ's runtime system searches compiled Java class files (Java bytecode) and replaces all occurrences of the type name of refinement classes with their original class names. The replaced type names are ones used as a return type, a parameter type, or a thrown exception type. The runtime system substitutes the bytecode generated by the runtime system for virtual machine instructions that perform a runtime type check. These instructions are checkcast (type cast operator) and instanceof.

If the destination type of the checkcast instruction is a refinement class with @Cflow, the bytecode generated by the runtime system is inserted before the instruction. The bytecode checks whether or not the refinement class with @Cflow is active. If the refinement class is inactive, a ClassCastException is thrown. Also, similar bytecode is inserted if the destination type of the checkcast instruction is an array type of a refinement class with @Cflow.

If the right-hand type of the instanceof instruction is a refinement class with @Cflow, the instanceof instruction is replaced with the bytecode generated by the runtime system. The bytecode first checks whether or not the refinement class with @Cflow is active. If not, the bytecode pushes 0 (false) onto the operand stack. Otherwise, it really executes the instanceof instruction and pushes the result onto the stack. Also, the instanceof instruction is replaced if the right-hand type is an array type of a refinement class.

The virtual machine instructions new, anewarray (array creation), and aastore (assignment to an array element) are not replaced by the runtime system. In GluonJ, creating an instance of a refinement class or an array of it is prohibited. Thus, since the new and anewarray instructions for a refinement class cannot appear in the whole program, they do not need to be replaced by the runtime system. Also, aastore is not replaced even though it performs a runtime type check. See the following example:

```
R[] r = new R[1];
C[] c = r;
c[0] = new C(); // throws ArrayStoreException
```

Here, R is a subclass of C. The last assignment, which is compiled into the aastore instruction, performs a runtime type check and throws an exception because of the covariant rule of Java. If R is a refinement class and C is its original class, then the behavior of the aastore instruction would be extended to perform a type check considering dynamic activation. However, since creating an

array of a refinement class is prohibited in GluonJ, there is never an array that can only contains R objects. Thus, replacing the aastore instruction is unnecessary.

3.5 Why is NoSuchMethodException never thrown?

A GluonJ program that is successfully compiled and loaded never throws a NoSuchMethodException. This exception is included in the standard Java library and it is thrown when an undefined method is called. Suppose that a client class calls a method m declared in a class C. After the client class is compiled, if the method m is removed from the class C and the class C is separately compiled again, then the client class will throw a NoSuchMethodException at runtime.

It is not straightforward to guarantee that a GluonJ program does not throw a NoSuchMethodException. If @Cflow is specified, a refinement class is effective during only limited time. Thus, a method appended by the refinement class is not always available. If a client class calls that method when it is not available, then an exception will be thrown.

Our idea is the following. (1) To call a method appended by a refinement class, a reference to the target object first must be cast to the type T of that refinement class. Then, (2) a value of this type T is alive at most while the refinement class is effective. Therefore, a method appended by a refinement class is never called when it is not available.

The property (1) is enforced by a standard Java compiler and the cast operation extended by GluonJ's runtime system. Since Java is a statically typed language, it is not possible that a client calls a method appended by a refinement class if the target reference is not the type of the refinement class. The type cast to the refinement class is extended by GluonJ and it succeeds only when the refinement class is effective.

(2) is guaranteed by the programming conventions shown in Section 2.3. A value obtained by the type cast to a refinement class may be stored in a local variable or a field. However, if the field is not declared in the refinement class, the value stored in the field will survive after the refinement class becomes ineffective. A method call on that field may throw a NoSuchMethodException. To avoid this, a GluonJ program must follow the programming conventions, which guarantee that a field of the type of a refinement class is declared only in its @Glue class. When the @Glue class becomes inactive, that field also becomes invisible from the program. The programming conventions also guarantee that a value of the type of a refinement class does not leak out of its @Glue class as a return value or an exception.

4. AN APPLICATION

Dynamic-refinement is useful for writing unit tests. Suppose that we are describing the test program for the contains method in the following BookTitles class.

```
public class BookTitles {
  protected String[] readDatabase() {
    // reads the book titles from a database
    // and returns them
  }
  public boolean contains(String title) {
    for (String s : readDatabase()) {
      if (s.equals(title)) return true;
    }
    return false;
} ... ... }
```

BookTitles is a program used to access the data of books that have been stored in a database. contains is the method for checking whether or not a book with the title passed as a parameter has been stored in a database. The readDatabase method accesses the database.

By using dynamic-refinement provided by GluonJ, we can test the behavior of contains without accessing a database. The following program allows switching the implementation of readDatabase only while a test method testContains is executed.

```
@Cflow("BookTitlesTest#testContains()")
@Glue class GlueForBookTitlesTest {
  @Refine static class MockTitles
              extends BookTitles {
    protected String[] readDatabase() {
      return new String[] { "Proc. SAC 08" };
} } }
public class BookTitlesTest extends TestCase {
  BookTitles table = new BookTitles();
  public void testContains() {
    assertTrue(table.contains("Proc. SAC 08"));
    assertFalse(table.contains("Proc. SAC 07"));
} ... ... }
```

5. RELATED WORK

Most AOP languages such as AspectJ [11], CaesarJ [5], PROSE [17], JAsCo [19], Steamloom [7] for Java more or less have a mechanism for changing the definition of an original class at runtime. However, they require extensions of the standard Java syntax. Although some of them provide Eclipse plugins for customizing the Eclipse IDE, the plugins cannot extend other IDEs because the specification of plugin mechanisms that other IDEs provide is different from Eclipse's. To use coding support of various IDEs when users write a program in those languages, those languages must provide the plugins for customizing those IDEs.

AspectJ5 [4], AspectWerkz [2], and JBoss AOP [3], which are annotation-based AOP frameworks, allow users to write aspects using standard Java syntax. They provide language constructs for dynamically changing the behavior of an existing method on an object with pointcut-advice. Unlike GluonJ, however, they do not provide language constructs for dynamically adding and removing new fields and methods to and from an object according dynamic contexts.

Hyper/J [16], eJava [20], MultiJava [10], mixin layers [18], virtual classes [14], nested inheritance [15], and so on, are similar to static refinement. To statically extend an original class and an original class hierarchy, these mechanisms have been proposed before. Also Classbox/J [6], Darwin/Lava [13], and delegation layers [12] allow extending an original class at runtime. All of them make extensions to Java's syntax.

ContextJ [9] enables switching the extension of original classes according to dynamic contexts. It allows users to override an existing method in its original class and to dynamically change multiple overriding methods. However, it does not allow appending a new method to an original class according to dynamic contexts. Thus it cannot change the behavior of an object of its original class according to dynamic contexts.

6. CONCLUSION

This paper proposed GluonJ, in which the users can statically or dynamically modify the definition of an existing class according to refinement at runtime. A main contribution of our work is the pragmatic design of a language construct for refinement. To make refinement available, GluonJ uses Java annotations and thereby does

not extend the lexical syntax of Java. Unlike other naive implementations, however, our implementation is quite compatible with an unmodified Java IDE. GluonJ has carefully been designed so that the users can exploit coding support by a standard Java IDE. Also, a GluonJ program can be edited with an unmodified Java editor and compiled with a standard Java compiler. Only a special runtime system is needed to run a GluonJ program. This feature of GluonJ is good for industrial acceptability.

Also, this paper explained that a GluonJ program never occurs a runtime type error NoSuchMethodException if the program is successfully compiled and loaded. To prevent a method appended by dynamic-refinement from being called when the method is not available, GluonJ requires users to several programming conventions. When a GluonJ program satisfies the conventions, an unavailable method is never invoked. To guarantee this property, GluonJ does not use a custom type checker. It exploits the type checking by a standard Java compiler and the verification by GluonJ's custom class loader. This paper, moreover, presented the implementation of GluonJ's refinement and illustrated a regression test program using GluonJ. At load time, GluonJ merges an original class and its refinement classes into one class definition.

7. REFERENCES

[1] *aspectj project — aspectj crosscutting objects for better modularity*. URL `http://www.eclipse.org/aspectj/`.

[2] *AspectWerkz - Plain Java AOP*. URI `http://aspectwerkz.codehaus.org/`.

[3] *JBoss Aspect Oriented Programming (AOP)*. URL `http://labs.jboss.com/portal/jbossaop`.

[4] *The AspectJ 5 Development Kit Developer's Notebook*. URL `http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html`.

[5] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ.

[6] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proc. of OOPSLA '05*.

[7] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *Proc. of AOSD '04*.

[8] S. Chiba. Load-Time Structural Reflection in Java. In *Proc. of ECOOP '00*.

[9] P. Costanza, R. Hirschfeld, and W. D. Meuter. Efficient Layer Activation for Switching Context-dependent Behavior.

[10] Curtis Clifton and Gary T. Leavens and Craig Chambers and Todd Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proc. of OOPSLA '00*, pages 130–145, New York, NY, USA, 2000. ACM Press.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. of ECOOP '01*, volume 2072, pages 327–355. Springer-Verlag, 2001.

[12] Klaus Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proc. of ECOOP '02*.

[13] G. Kniesel. Type-Safe Delegation for Run-Time Component Adaptation. In *Proc. of ECOOP '99*, pages 351–366, Lisbon, Portugal, June 1999. Springer-Verlag.

[14] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *Proc. of OOPSLA '89*, pages 397–406, New York, NY, USA, 1989. ACM Press.

[15] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. In *Proc. of OOPSLA '04*.

[16] H. Ossher and P. Tarr. Hyper/J: Multi-dimensional Separation of Concerns for Java. In *Proc. of ICSE '00*.

[17] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proc. of AOSD '02*.

[18] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[19] D. Suvee, W. Vanderperren, and V. Jonckers. JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. In *Proc. of AOSD '03*.

[20] A. Warth, M. Stanojevic, and T. Millstein. Statically Scoped Object Adaptation with Expanders. In *Proc. of OOPSLA '06*.