Difference-Based Modules: A Class-Independent Module Mechanism

Yuuji Ichisugi[1] and Akira Tanaka[2]

[1] PRESTO, Japan Science and Technology Corporation(JST) / National Institute of Advanced Industrial Science and Technology(AIST),
y-ichisugi@aist.go.jp
[2] National Institute of Advanced Industrial Science and Technology(AIST),
akr@m17n.org

Abstract. We describe a module mechanism, which we call *difference-based modules*, and an object-oriented language we call *MixJuice*. MixJuice is an enhancement to the Java language that adopts difference-based modules instead of Java's original module mechanism. Modules are units of information hiding, reuse and separate compilation. We have completely separated the class mechanism and the module mechanism, and then unified the module mechanism and the differential programming mechanism. Although this module mechanism is simpler than that of Java, it enhances ease with which programs can be extended, reused and maintained. Collaborations that crosscut several classes can be separated into different modules. Modules are composable in the same way as mix-ins. The composition of modules sometimes causes name collision and an interesting phenomenon, which we call *implementation defects*. We describe solutions to these problems.

1 Introduction

Modules are units of information hiding and reuse. Classes are templates of objects. These two notions are inherently different. However, in current object-oriented languages such as C++ and Java, the language construct "`class`" has the functions of a module. We call this type of module mechanism *class-based modules*. In large-scale programs, various problems occur when classes are used as modules.

One problem, pointed out by Szyperski[30], is that classes are inappropriate as units of information hiding. A class is appropriate as a unit of information hiding only if it is a simple abstract data type such as a stack. If one or more classes collaborate closely to realize a function, these classes are not appropriate as units of information hiding. In order to alleviate this problem, mechanisms such as packages and nested classes[10] have been introduced into Java. Even though these mechanisms have been introduced, class-based modules suffer a major problem. If the number of functions possessed by the software increases, the fields and methods needed for each class will also increase. This enlarges the size of the class, more specifically its scope, thus making system maintenance

more difficult. For example, the size of the source file of the class `TreeMap`, which is in the standard Java library, is about 1,000 lines, not including comment lines. Because all lines share a single name space, it is difficult to predict which parts of the source file will be influenced if a part of it is modified.

Another problem is that classes are inappropriate as units of reuse. Over the past few years, several studies have been made on this problem. The source-code related to a concern may crosscut more than one class[17]. In order to increase the reusability of the programs, such *crosscutting concerns* should be separated from the other parts of the program. Some systems such as AspectJ[16], Hyper/J[26], Demeter/Java[19] and DJ[25] have been proposed to support *separation of crosscutting concerns*.

Other than these, some studies have focused on *collaborations* instead of classes as units of reuse[11, 31, 27, 14, 22, 23]. A collaboration is a set of the fields and methods of two or more classes in relation to a certain function. In order to make collaborations into reusable units, the programming language should feature a *mixin*[3] or similar mechanism.

Mixins are fragments of classes. The programmers can define a new class by composing existing mixins. The use of mixins is a common programming technique used in programming languages that support multiple inheritance with class linearization, such as CLOS[28]. Mixins increase the reusability of programs because each mixin can be used as a part of more than one class.

VanHilst and Notkin have proposed a programming technique to implement mixins using C++ template mechanisms, in order to support collaboration-based design[31]. Mixin layers[27], however, are an improved programming technique that make the composition of the reusable parts much easier than in the VanHilst and Notkin method. Mixin layers are sets of mixins belonging to certain collaborations.

Independently, we have designed and implemented a mechanism named SystemMixins[14] on top of Java. SystemMixins are similar to mixin layers, which are sets of mixins belonging to certain collaborations. We have implemented an extensible Java pre-processor(EPP)[14, 12] using this mechanism. The user can extend the language specification of Java by adding new collaborations to the pre-processor using the SystemMixin mechanism. A wide variety of language extensions have been implemented, including a data-parallel language[14], thread migration[2], parameterized types[12] and SystemMixin mechanism itself.

As a result of our experience with EPP implementation, we are convinced that collaborations are appropriate as units of reuse, especially for applications with extremely high extensibility, such as EPP.

As pointed out in [11, 31], in collaborations, groups of objects cooperate to perform a task or to maintain an invariant. Therefore, collaborations must be suitable for not only units of reuse but also units of information hiding. However, both mixin layers and SystemMixins lack the function of information hiding.

In this paper, we propose a module mechanism which we call *difference-based modules*. We have designed and implemented an improved version of Java, which we call the *MixJuice* language[13], which adopts difference-based mod-

ules instead of Java's original module mechanism. We first completely separated the class mechanism and the module mechanism, and then unified the module mechanism and the differential programming mechanism. By applying difference-based modules, we have resolved the problems associated with the above-described conventional class-based modules. Using this module mechanism, collaborations can become units of information hiding and reuse instead of classes. This module mechanism is based on the three simple design principles: *difference definition*, *name-space inheritance* and *name-collision avoidance*.

The rest of this paper is organized as follows. In Section 2, we describe differential programming using this module mechanism. In Section 3, we describe the other feature of this module mechanism, information hiding. In Section 4, we explain an implementation defect phenomenon that may occur in highly extensible systems. In Section 5 we describe an application of MixJuice. Section 6 covers related work. We conclude with Section 7.

2 Differential Programming Using Difference-Based Modules

2.1 Principle and Merits

This module mechanism is based on the following design principle.

> **The principle of difference definition:** A module is the difference between the original program and the extended program. The difference is a set of definitions of new names and modifications of definitions of existing names[1].

Modules are units of reuse, information hiding and separate compilation. The executable application is constructed by linking of modules. In the case of difference-based modules, linking of modules means adding all differences defined by the modules to the empty program.

Difference-based modules can be applied to various programming languages. In many programming languages, a program consists of names and their definitions. For example, in the case of imperative languages, a program is a set of definitions of procedures and data structures. In the case of Java, a program is a set of definitions of classes, fields and methods. The MixJuice language is a modified Java language, which adopts difference-based modules instead of Java's original module mechanism. In other words, in MixJuice, a module is a set of additions and modifications of classes, fields and methods.

Modules may inherit other modules. In MixJuice, both the module-inheritance mechanism and the traditional class-inheritance mechanism can be independently available. Class inheritance and module inheritance are different, as described next. Class inheritance is a mechanism for describing the difference between classes. Module Inheritance is a mechanism for describing the difference

[1] Currently, the difference includes neither the renaming nor deletion of names.

between two programs consisting of one or more classes. Using class inheritance, the programmers can only define a new class which has a different name from that of the original class. By module inheritance, the programmers can modify the definitions of existing classes and methods without changing their names. Class inheritance is a mechanism for subtyping and safe late binding. Module inheritance is a mechanism for static reuse and information hiding as described in Section 3.

Classes no longer have the functions of modules. In other words, classes are no longer units of reuse, information hiding, or separate compilation.

Difference-based modules have the following merits compared with class-based modules.

- **High extensibility of applications**
  It is easy to write highly extensible applications. There are two reasons for this. One is that all class and method names act as "hooks" for programmers of extension modules. The other reason is that each extension module is composable as a mixin, using multiple inheritance of modules. (Details are described in Section 2.3.)
- **Class-independency of units of reuse**
  Programmers can define the units of reuse completely independently of boundaries of classes. The programmers can make codes that crosscut some classes, namely collaborations, units of reuse (Figure 1).

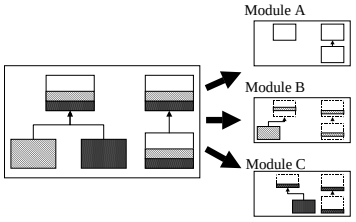

Fig. 1. Separation of crosscutting code.

- **Extensibility by third party programmers**
  Third party programmers can provide extension modules to extend existing applications. The programmers do not need to have the source-code of the original programs. (We give a more detailed account of this process in Section 2.5.)
- **Module-composability by end-users**
  End users can compose existing modules that provide selected functions to create their own customized applications. The composition of modules does

not require any lines of "glue code". It only requires a set of module names. (Details are included in Section 2.5.)

– **Flexibility of module grouping**
  The programmers can make groups of modules and give names to them to simplify their use. In the case of Java, a certain degree of grouping is possible due to the package mechanism and the use of an "`import`" declaration in the form of "`import p.*;`". For difference-based modules, however, more flexible grouping is possible. (Details are included in Section 2.3.)

The rest of this section describes differential programming using difference-based modules in greater detail.

2.2 Syntax of Module Definitions

```
module m1 {
   define class S {
     define S(){}
     define int foo(){ return 1; }
   }
   define class A extends S {
     define A(){}
     int foo(){ return original() + 10; }
   }
   class SS {
     void main(String[] args){
       A a = new A();
       System.out.println(a.foo());
     }
   }
}
module m2 extends m1 {
   class S { int foo(){ return original() + 2; } }
   class A { int foo(){ return original() + 20; } }
}
```

Fig. 2. Definitions of module `m1` and module `m2`.

Modules are defined as illustrated in Figure 2. The modules `m1` and `m2` are defined in Figure 2.

An "`extends`" declaration at the top of the module definition specifies the module to which the difference is intended to be added. The declared module is called a *super-module*. In Figure 2, the module `m2` declares the module `m1` to be a super-module of `m2`. At this time, we can say that "`m2` is a sub-module of `m1`"; or "`m2` inherits `m1`".

```
class $S1$ { int foo(){ return 1; } }
class S extends $S1$ { int foo(){ return super.foo() + 2; } }
class $A1$ extends S { int foo(){ return super.foo() + 10; } }
class A extends $A1$ { int foo(){ return super.foo() + 20; } }
class SS {
  void main(String[] args){
    A a = new A();
    System.out.println(a.foo());
  }
}
```

Fig. 3. A Java program almost equivalent to the program defined by module `m2`.

A module definition without an "`extends`" declaration, like module `m1`, denotes that the difference is assumed to be added to the empty program.

The module body, enclosed by braces, is the definition of the difference between the original program and the extended program. Specifically, a module can modify the program defined by its super-module as follows:

- Addition of new classes[2].
- Addition of fields to existing classes.
- Addition of methods to existing classes.
- Modification of existing methods by overriding.

In Figure 2, the module `m2` extends the behavior of the method `foo` of class `S` and the method `foo` of class `A` by overriding those methods originally defined in the module `m1`.

The syntax of the inside of the module body is closely similar to Java; however, it differs from Java in the following ways.

The definitions of new names require the keywords "`define`". More accurately, the declarations of classes, fields, constructors and methods preceded by "`define`" denote they are the new definitions. The declarations of classes, constructors and methods without "`define`" denote them to be modifications of existing definitions. (Because the class `SS` and its method `main` are pre-defined names, they do not require "`define`".)

An expression "`original()`" is used when an overriding method invokes the overridden method. This is a similar mechanism to the method invocation of "`super`" in Java. In MixJuice, there are two kinds of method overriding. One is overriding by class inheritance, and the other is by module inheritance. In case of Figure 2, an "`original()`" in the module `m1` is for the former, and two "`original()`" in the module `m2` are for the latter.

[2] In this paper, we do not mention "interfaces". Actually, the current implementation of MixJuice allows both extension of existing interfaces and addition of super interfaces to existing classes.

In MixJuice, there is no use of package mechanisms or access modifiers (`public`, `protected` or `private`). How information hiding is achieved in MixJuice is described in Section 3.

The MixJuice program defined by module `m2` is closely equivalent to the Java program as seen in Figure 3.

2.3 Multiple Inheritance of Modules

```
module m3 extends m1 {
   class S { int foo(){ return original() + 3; } }
   class A { int foo(){ return original() + 30; } }
}
module m4 extends m2, m3 {
   class S { int foo(){ return original() + 4; } }
   class A { int foo(){ return original() + 40; } }
}
```

Fig. 4. Multiple inheritance of modules.

A module can inherit more than one super-module.

Figure 4 is an example of multiple inheritance of modules. The module `m3` defines, as well as `m2`, the difference between the extended program and the program defined by `m1`. The module `m4` inherits both `m2` and `m3`. In this case, the modules form a so-called "diamond inheritance" because both `m2` and `m3` inherit `m1`.

All modules are linearized by topological sort. This is similar to the class linearization done in some object-oriented languages with multiple-inheritance mechanisms such as CLOS[28].

For example, the program defined by the module `m4` is constructed as follows. First of all, the set of the module `m4` itself and the ancestor modules of `m4` are found. The set is $\{\texttt{m1}, \texttt{m2}, \texttt{m3}, \texttt{m4}\}$. The set is then linearized by topological sort so that it preserves the order between super-modules and sub-modules. The result of this topological sort is called a *linearized list*. In this case, the linearized list may be (`m1` `m2` `m3` `m4`). Finally, all differences defined by the modules are applied to the empty program ϕ, from the beginning of the linearized list to the end. That is, if the notation "$a \triangleleft b$" expresses the result of addition of a difference "b" to "a", the constructed program is expressed as:

$$((((\phi \triangleleft \texttt{m1}) \triangleleft \texttt{m2}) \triangleleft \texttt{m3}) \triangleleft \texttt{m4})$$

A serious problem incurred by multiple inheritance is name collision. In MixJuice, this problem is completely resolved. The details are described in Section 3.7.

The multiple-inheritance mechanism of modules can be used as a grouping mechanism. To make the utilization of the group of modules more convenient, the programmer can define a group of modules and name it. For example, the following is the definition of a group named "`m_x`".

```
module m_x extends m_a, m_b, m_c, m_d {}
```

This mechanism is more flexible than Java's grouping mechanism using the declarations like "`import p.*;`", which is grouping based on packages. In addition, MixJuice allows definitions of groups of groups, which are not possible in Java.

2.4 Programming Styles Specific to MixJuice

Using difference-based modules, programmers can write programs in modular style, even if the programs cannot be written in modular style using traditional class-based modules.

The programmer can add a new traversing code for the tree structure without modifying the original source-code, because modules can add new methods to existing classes. In traditional object-oriented languages, it is possible to add a new traversing code using Visitor pattern[8]; however, the use of Visitor pattern disables the addition of the new kind of nodes to the tree structure unless the source-code is modified.

The programmer can split a nested if-statement as in Figure 5 into the modules shown in Figure 6, if each condition is disjoint. This programming style enables the programmers to add new conditional clauses without modifying the source-code. With this style, for example, recursive-descent parsers become modular and highly extensible[14]. Another application of this style is in programs processing XML that normally have nested if-statements.

```
class F {
   void branch(String s){
     if (s.equals("a")){ ... }
     else if (s.equals("b")){ ... }
     else { throw new Error(); }
   }
 }
```

Fig. 5. Nested if-statements.

The programmer can split initialization codes, such as initialization of tables, into modules. Moreover, when initializing tables, the programmer can add new entries to a table without needing to modify the source-code, because the initialization method acts as a hook for the extension modules. These initialization codes tend to be concentrated in a single method in traditional object-oriented languages.

```
module framework {
  define class F {
    define void branch(String s){ throw new Error(); }
  }
}
module case_a extends framework {
  class F {
    void branch(String s){
      if (s.equals("a")){ ... } else { original(s); } }
  }
}
module case_b extends framework {
  class F {
    void branch(String s){
      if (s.equals("b")){ ... } else { original(s); } }
  }
}
```

Fig. 6. Modularized nested if-statements.

2.5 Execution Environment

This section describes the characteristics of MixJuice with respect to compiling, linking and execution.

Separate Compilation of Modules Each module can be separately compiled. Although each module contains fragments of classes, such fragments are type-checked by the compiler. When compiling a module, the compiler requires the ancestor modules of the modules. More accurately, the source-codes or the compiled binaries[3] of the ancestor modules should be accessible by the compiler.

Because units of separate compilation are independent of class, MixJuice is ideal for realistic application development, which sometimes uses collaborations as development and testing units. In MixJuice, each collaboration can be written and tested by an independent development team.

Linking and Execution of Modules In order to execute a program, all the modules that make up the program must be linked together. Current implementation of MixJuice features the `mj` command which links and executes the modules.

To execute a program defined by a module, the module name should be specified as an argument of the `mj` command. The `mj` command links the specified module and produces a executable Java program[4]. The `mj` command then loads

[3] In the current implementation, the result of compilation of a module is represented by a set of class files of the Java language.

[4] The current implementation performs byte-code translation to compose fragments of classes.

the Java program to the JavaVM and executes it. The `mj` command invokes a `main` method of the class `SS` by default.

The `mj` command automatically links all the ancestor modules of the specified module[5]. For example, if the module `m2` is specified as an argument, the module `m1` is automatically linked. Below is an example of execution of the modules `m1`, `m2`, `m3` and `m4`.

```
% mj m1
11
% mj m2
33
% mj m3
44
% mj m4
110
```

Actually, the `mj` command automatically links modules of a type other than ancestor modules, *complementary modules*, which are described in Section 4.

Composition of Selected Modules The end-users of an application can select specific modules and construct their own configured applications without having to write any lines of code.

For example, end-users can compose the module `m2` and `m3`, which might be independently developed modules. To compose selected modules, the "`-s`" option of the `mj` command is used as follows.

```
% mj -s m2 m3
66
```

The `mj` command makes a virtual module named "`_bottom`" which extends all the selected modules specified as arguments. In the above case, the definition of the module "`_bottom`" would look like the following.

```
module _bottom extends m2, m3 {}
```

The `mj` command then executes the program defined by the module "`_bottom`". That is, the program expressed as

$$(((\phi \triangleleft \texttt{m1}) \triangleleft \texttt{m2}) \triangleleft \texttt{m3})$$

is executed.

End users can select more than two modules by specifying the "`-s`" option more than once. (The current version of MixJuice does not permit the addition of a difference defined by a module more than once.)

[5] In the current implementation, the linker finds the required modules from the `CLASSPATH`.

In this way, end-users can compose existing modules that provide chosen functions in order to build their own customized applications. Traditionally, this type of customization is achieved by the mechanism of conditional compilation, such as "`#ifdef`", or patching onto the source-code. These mechanisms are processed at string level; difference-based modules, conversely, are more reliable because they are processed at the language level. In addition, difference-based modules have the advantage of not requiring the source-codes of extension modules to be available to the public.

3 Information Hiding Using Difference-Based Modules

In this section, we describe how the module mechanism of MixJuice is more powerful than that of Java with respect to information hiding.

3.1 Principle and Advantages

The module mechanism of MixJuice is based on the following design principle concerning information hiding.

> **The principle of name space inheritance:** All names that are defined at a module are visible from the module itself and its descendant modules, and are invisible from the other modules.

More specifically, "names" means the class, field and method names. The module mechanism of MixJuice enables more flexible name space management than that of Java by means of this simple rule concerning visibility.

Classes are no longer the units of information hiding in the source-code. All fields in a class are accessible from the defining module and the descendant modules of the module, even if the accessor class is different from the owner of the fields. In MixJuice, there are no access modifiers (`public`, `protected` or `private`), package mechanisms or nested class mechanisms[6].

Difference-based modules have the following advantages with respect to information hiding compared with Java.

- **Class-independency of units of information hiding**
  Programmers can make the boundaries of information hiding independent of class boundaries. For example, programmers can make collaboration units of information hiding. (Details are described in Section 3.6.) In addition, to improve the maintainability of the source-code, a programmer can minimize the size of the name space on which their source-code depends. This is especially effective if the number of functions of classes increases and the size of the classes thus becomes bigger and bigger.

[6] To be precise, MixJuice supports a kind of nested class, anonymous classes which are often used for GUI programming in Java.

– **Flexibility of name space structures**
  The name spaces can form nested structures and, giving a more general structure than nesting, overlapping structures. This characteristic makes Java's nested class mechanism unnecessary, with the result that the language specification is radically simplified. (Details are described in Section 3.4 and Section 3.5.)
– **Ease of code-moving**
  Programmers can easily move code between modules. This is due to a characteristic of difference-based modules: moving code between a super-module and a sub-module does not affect the semantics of the linked modules. As a result, the programmer can perform a kind of refactoring[7] with a high degree of flexibility and without changing the structure of the classes. For example, inter-dependent classes can be split into non-inter-dependent modules without changing the structure of the classes. (Details are described in Section 3.6.) Ease of code-moving enables smooth shifting from a monolithic prototyping source-code to a modular and extensible source-code.
– **Simplicity**
  Names are inherited only by one mechanism: module inheritance. On the other hand, the specification of Java concerning names is extremely complex. For example, four kinds of classes can be referred to by simple names: (1) Classes belonging to the same package. (2) Classes declared by `import` declarations. (3) Member classes of the outer classes. (4) Member classes of the ancestor classes. The relation between these four mechanisms is far from intuitive.

The rest of this section describes the details of information hiding using difference-based modules.

3.2 Black-box Reuse

In MixJuice, the programmer can utilize existing classes in a manner called *black-box reuse*. We define black-box reuse as a style of utilization of existing classes only depending on the external interface of the classes.

A class can be defined as two separate modules. One module is called the *specification module*, which only defines the external interface of the class using *abstract constructors* and *abstract methods*; and the other is called the *implementation module*, which defines the internal implementation of the class. Abstract constructors, which do not appear in Java, are introduced to separate the interface and the implementation of constructors.

The program in Figure 7 is a definition of a class `Point` which consists of two modules: a specification module, `point` and an implementation module, `point.implementation`[7]. The module `point` defines the interfaces of the con-

[7] The character "`.`" contained in the module name `point.implementation` is merely a punctuator. In MixJuice, the hierarchical structure of module names does not affect the semantics of the program.

```
module point {
  define class Point {
    // abstract constructor:
    define abstract Point(int x, int y);
    // abstract methods:
    define abstract void move(int dx, int dy);
    define abstract int getX();
    define abstract int getY();
  }
}
module point.implementation extends point {
  class Point {
    define int x;
    define int y;
    Point(int x, int y){ this.x = x; this.y = y; }
    void move(int dx, int dy){ x += dx; y += dy; }
    int getX(){ return x; }
    int getY(){ return y; }
  }
}
```

Fig. 7. The specification module and the implementation module.

structor and the methods of the class `Point`. The module `point.implementation` implements the constructor and methods.

Other modules can utilize the class `Point` in the style of black-box reuse, by means of inheriting the specification module of the class `Point`. In Figure 8, the module `point.test` is an example of black-box reuse. The module `point.test` inherits the module `point`, and does not inherit `point.implementation`.

```
module point.test extends point {
  define class Test {
    define void test(){
      Point p = new Point(1, 2);
      p.move(10, 10);
      ...
    }
  }
}
```

Fig. 8. An example of black-box reuse.

3.3 White-box Reuse

In MixJuice, programmers can utilize existing classes in the manner of *white-box reuse*, in addition to black-box. We define white-box reuse as a style of utilization of existing classes depending not only on the external interface of the classes, but also on the internal implementation of the classes.

In Java, a class can serve for both black-box reuse and white-box reuse due to the `protected` access modifier. By defining the internal implementation of a class with the `protected` modifiers, a programmer can make internal implementation accessible by the subclasses of the class, but inaccessible by classes other than the subclasses.

```
module colorPoint extends point {
  define class ColorPoint extends Point {
    define abstract ColorPoint(Color c, int x, int y);
  }
  define class Color {...}
}
module colorPoint.implementation extends colorPoint, point.implementation{
  class ColorPoint {
    define Color c;
    ColorPoint(Color c, int x, int y){ super(x, y); this.c = c; }
    ...
  }
}
```

Fig. 9. The definition of the class `ColorPoint`.

In MixJuice, the programmer utilizes a class in the style of white-box reuse by inheriting the implementation module of the class. `protected` modifiers are no longer used in MixJuice. Figure 9 is an example of white-box reuse. The `ColorPoint` class is a subclass of the class `Point` defined in the program illustrated in Figure 7. The module `colorPoint.implementation` inherits not only the module `colorPoint`, but also the module `point.implementation`. The inheritance graph of these modules is illustrated in Figure 10. This inheritance structure enables the programmer to utilize the internal implementation of the class `Point` when implementing the class `ColorPoint`.

In MixJuice, the programmer can choose either black-box reuse or white-box reuse when implementing a class, independently of the inheritance relation of classes. For example, if the module `point.test` in Figure 8 inherits the module `point.implementation`, the style is white-box reuse; if the module `colorPoint.implementation` in Figure 9 does *not* inherit the module `point.implementation`, the style is black-box reuse.

With white-box reuse, the programmer can utilize the internal implementation of other classes; however, white-box reuse has the following disadvantages.

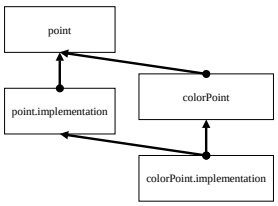

Fig. 10. The module diagram of the class `Point` and `ColorPoint`.

If the internal implementation on which a module written by a programmer depends is modified, the programmer must rewrite the module. In addition, if the programmer accesses the internal implementation of other classes, the programming needs to be done more carefully in order to preserve the class invariants of the classes.

Currently, MixJuice does not have a mechanism for preventing careless inheritance of implementation modules: such mistakes are prevented by the naming convention, in which the implementation modules are given long names such as `point.implementation`.

3.4 Nested Name Spaces

Nested name spaces, which are expressed by nested classes in Java, are expressed by module inheritance in MixJuice.

The Java program illustrated in Figure 11 is an example of nested name space expressed as nested classes. The field `x` of class `A` is not accessible from outside of class `A` because it is not a public field; it is, however, accessible from class `B` which is a member of class `A`.

```
public class A {
  protected static int x = 0;
  public static class B {
    public int getX(){ return x; }
  }
}
```

Fig. 11. A Java program with nested classes.

The program illustrated in Figure 12 is almost the same program written in MixJuice. The module `A_B` defines the public names of class `A` and class `B`. The module `A_B.implementation` is the implementation module, which defines the protected names of classes `A` and `B`. The method `getX` in class `B` directly

```
module A_B {
  define class A { }
  define class B {
    define abstract int getX();
  }
}
module A_B.implementation extends A_B {
  class A { define static int x = 0; }
  class B { int getX(){ return A.x; } }
}
```

Fig. 12. A MixJuice program that represents nested name spaces.

accesses the static field of class `A`, using the expression "`A.x`". In this way, all names defined in a module can be accessed from the inside of the module (and the descendant modules of the module) even if the accessor class is a different class.

Similarly, n-levels of nesting of name spaces can be expressed using n-levels of module inheritance(Figure 13).

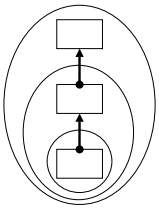

Fig. 13. The nested name spaces formed by inheritance of modules.

3.5 Overlapping Name Spaces

With multiple inheritance of modules, overlapping name spaces can be expressed, which have a more general structure than nested name spaces.

For example, the programs in Figure 2 and Figure 4 form overlapping name spaces as seen in Figure 14. That is to say, the lowest module, `m4`, is inside the name spaces defined by the modules `m1`, `m2` and `m3`.

In the case of class-based modules, programmers are often forced to make names public to a greater extent than actually necessary because of the low degree of flexibility of name spaces. For example, some method names are often defined as public methods even though they are accessed by only members of a collaboration. A programmer can sometimes encapsulate such names using

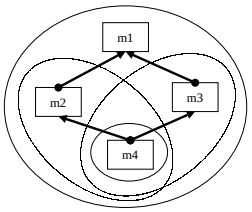

Fig. 14. The overlapping name spaces formed by multiple inheritance of modules.

nested classes; however, the use of nested class mechanisms damages the maintainability of the classes because the scope of the field names defined by the classes expands.

In the case of difference-based modules, the programmers can minimize the size of scope of names. In addition, programmers can easily uncover the dependency relation between the modules because it is explicitly declared by the "`extends`" declaration.

3.6 Collaboration-Based Modularization

```
class A {  // class A uses class B
  void m1(B b){ ... b.m3(); ...}
  void m2(){...}
}
class B {  // class B uses class A
  void m3(){...}
  void m4(A a){ ... a.m2(); ...}
}
```

Fig. 15. Inter-dependent classes containing two collaborations.

The programmers can define a collaboration that crosscuts more than one class as a separate module, since classes and modules are completely orthogonal in MixJuice.

Consider the program in Figure 15 written in Java. Two classes, classes A and B, depend on each other; however, these classes actually contain two independent collaborations.

The program can be modularized as in Figure 16. The program contains two unrelated modules, `collaboration_m1_m3` and `collaboration_m2_m4`.

Modularization based on collaborations has the following advantages:

– The volume of the source-code on which each module depends decreases. In general, this leads to increased maintainability.

```
module A_B {
  define class A {}
  define class B {}
}
module collaboration_m1_m3 extends A_B {
  class A { define void m1(B b){ ... b.m3(); ...} }
  class B { define void m3(){...} }
}
module collaboration_m2_m4 extends A_B {
  class A { define void m2(){...} }
  class B { define void m4(A a){ ... a.m2(); ...} }
}
```

Fig. 16. Modularized inter-dependent classes.

- Because `collaboration_m1_m3` and `collaboration_m2_m4` do not depend on each other, one of the two modules can be compiled and executed even if the other module does not exist. Therefore, these two modules can be developed and tested by different development teams.
- Other variations of application can be provided by means of implementing different versions of collaborations. For example, the module `collaboration_m1_m3` can be replaced by another module `my_collaboration` which contains completely different methods. In this case, existing modules, such as modules `A_B` and `collaboration_m2_m4`, need not be re-compiled.

3.7 Fully-Qualified-Names

The name-collision problem that is incurred by multiple inheritance is fully resolved in MixJuice. In Java, the name-collision problem caused by `import` declarations is resolved by the idea of *fully-qualified-names* (FQNs) of classes. In MixJuice, this idea is applied to all names including field and method names in order to resolve the problem.

In difference-based modules, all names are processed based on the following design principle.

> **The principle of name-collision avoidance:** Each name has a unique FQN. Each FQN consists of "the module name which first defined the name" and "a simple name". If a simple name is used at one point in the source-code and more than one candidate which has the same simple name is accessible at that point, the compiler will report an error because the reference is ambiguous. Two names defined at different places are never regarded as identical by the compiler. A name definition never shadows another name. If an error is reported because of an ambiguous reference to a name, the programmer can always avoid this error by using the FQN of the name instead of the simple name.

We assume that the uniqueness of the module names is guaranteed by other mechanisms or rules, such as the naming convention adding as a prefix the domain name of the vendor, as in Java.

In MixJuice, an FQN which consists "the defining module name `m`" and "the simple name `n`" is expressed as "`FQN[m::n]`"[8].

The FQNs of methods are used not only for method invocations, but also for method overriding. The program in Figure 17 is an example of overriding and invocations of the two methods `m` independently defined at the module `m2` and `m3`.

Although a similar notation, "`c::n`" is used in C++[29] to resolve name-collision, its semantics is quite different from that of MixJuice. In C++, the expression "`a.c::n()`" is not a virtual function call. In MixJuice, the expression "`a.FQN[m2::m]()`" is a normal method invocation with late binding. In addition, in C++, it is impossible to override two virtual functions with the same name as in Figure 17 because separately defined virtual functions with the same name are regarded as identical by the C++ compiler.

4 Implementation Defects and Complementary Modules

4.1 Implementation Defects

When the end user composes two modules, an interesting phenomenon, which we call an *implementation defect*, may occur. Consider the program in Figure 18. The module `m1` defines an abstract class `S` and its subclass `A`. The module `m2` adds a new subclass `B`. On the other hand, the module `m3` adds a new abstract method `m` to the class `S` and an implementation of the method `m` to the class `A`. Both `m2` and `m3` are the complete program which will not result in a link-time error; however, if the end user selects both modules simultaneously, the linker reports a link-time error because the method `m` in the class `B` is not implemented. As in this example, the phenomenon where the composition of two correct modules produces un-implemented abstract methods is called an implementation defect.

4.2 Complementary Modules

In general, it is impossible to complement an implementation defect automatically. Someone who understands the specification needs to implement abstract methods to make the program executable. Modules that complement an implementation defect between other modules are called *complementary modules*.

The program in Figure 19 is an example of a complementary module `m23` which complements the implementation defect between `m2` and `m3`. The complementary module is defined as a module that has a "`complements`" declaration

[8] The syntax of FQN shown in this paper is ugly. The reason is the parsing problem caused by the character "`.`" used as both the punctuator of module names and the access operator for fields and methods in Java. In actual programming in MixJuice, FQNs are seldom used because the scope of names becomes smaller than that in traditional object-oriented languages.

```
module m1 {
  define class A { define A(){} }
}
module m2 extends m1 {
  class A { define int m(){ return 1; } }
}
module m3 extends m1 {
  class A { define int m(){ return 2; } }
}
module m4 extends m2, m3 {
  class A {
    int FQN[m2::m](){ return original() + 3; }
    int FQN[m3::m](){ return original() + 4; }
  }
  class SS {
    void main(String[] args){
      A a = new A();
      //System.out.println(a.m()); // ambiguous
      System.out.println( a.FQN[m2::m]() ); // 4
      System.out.println( a.FQN[m3::m]() ); // 6
    }
  }
}
```

Fig. 17. An example of FQNs.

```
module m1 {
  // S and a subclass A.
  define abstract class S { }
  define class A extends S { }
}
module m2 extends m1 {
  // Add a new subclass of S.
  define class B extends S { }
}
module m3 extends m1 {
  // Add a new method of S.
  class S { define abstract int m(); }
  class A { int m(){ return 1; } }
}
```

Fig. 18. An example of an implementation defect between two modules.

```
module m23 complements m2, m3 {
  class B { int m(){ return 2; } }
}
```

Fig. 19. The complementary module.

that declares the modules that cause the implementation defect. The compiler processes the "`complements`" declaration in the same way as an "`extends`" declaration, except that the compiler adds information of module names to be complemented to the compiled binary.

The linker of MixJuice supports automatic linking of complementary modules in order to enhance the usability of end users who compose existing modules. Suppose that the complementary module as in Figure 19 is implemented by someone and the compiled binary is placed where the linker is able to access it. If an end user tries to compose `m2` and `m3` as follows, the linker automatically finds the complementary module `m23` and links the complementary module together with `m2` and `m3`.

```
% mj -s m2 m3
```

As shown above, if the complementary modules are properly installed, the end users do not have to be aware of the implementation defect problem when composing modules. By this mechanism, the end users can customize applications easily without requiring detailed knowledge of implementation of the modules.

4.3 Implementation Defect and Complementary Modules in Other Systems

The implementation defect problem occurs not only in MixJuice, but also in other highly extensible systems. To be precise, it occurs in extensible systems that have two or more directions of extension (Figure 20).

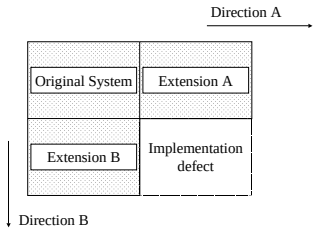

Fig. 20. Two directions of extension and their implementation defect.

One example can be seen in the extensible interpreter using monad transformers[18], which are composable extension modules. Monads are abstract data types which become extensible by a technique called monad transformers in order to extend their data-structure and applicable operations. When composing two monad transformers, "lifting of operators" are sometimes required to make the monad complete. This is what we call a complementary module.

Another more familiar example can be seen in personal computers (PCs). The users of a PC can choose their OS and peripherals; however, the device drivers corresponding to the selected OS and the peripherals need to be obtained and installed in order to make them work. These device drivers are what we call complementary modules.

The latter case is a good example that shows the extensible systems which cause implementation defects phenomena are not necessarily impractical. Not all possible implementation defects have to be complemented by the vendors of the extension modules. By implementing complementary modules between relatively popular modules, vendors can satisfy most of the demand for customization by the end users.

5 Application

As an example of a typical object-oriented application, we now describe a drawing tool [9]. The source-code of the tool contains a class hierarchy: an abstract class `Figure` and subclasses of the `Figure` corresponding to each kind of figure such as circles and rectangles.

This program is extensible by adding extension modules. Extension modules can add new kinds of figures or new kinds of operations to the figures.

The module that adds a new kind of figure contains the following code: (1) A definition of a new subclass of the class `Figure`. (2) An extension of the method that displays the buttons to select the figure to draw. The programmer can add a new button without modifying the source-code because the method that displays buttons is a "hook" for extension.

The module that adds a new kind of operation to the figures contains the following code: (1) The definition of an abstract method of the class `Figure` and the implementation of the method of all the subclasses of the class `Figure`. (2) An extension of the method that displays the buttons to select the operation to perform.

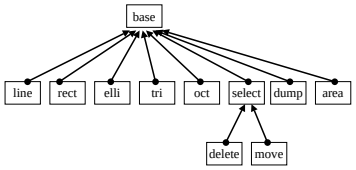

Fig. 21. The module diagram of the drawing tool.

[9] The source-code of this drawing tool and demonstration as Java applets are accessible from the following URL. `http://staff.aist.go.jp/y-ichisugi/mj/demo.html`

Currently, the following 11 modules have been implemented.

`base` : Framework of the drawing tool
`select` : Selection of a figure
`delete` : Deletion of a selected figure
`move` : Moving of a selected figure
`dump` : Dumping of information of the displayed figures
`area` : Display of the total area of the displayed figures
`line` : Lines
`rect` : Rectangles
`elli` : Ellipses
`tri` : Triangles
`oct` : Octagons

The module `base` contains the definitions of a class `Figure` and a class that displays a canvas, menus and buttons. The module `base` can itself be executed as an application; however, it displays no buttons for selecting the figure to draw or buttons to select the type of operation. All other modules are defined as sub-modules of this module (Figure 21).

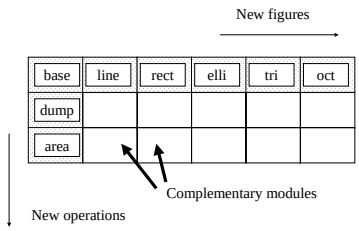

Fig. 22. Complementary modules for the drawing tool.

The combination of modules which add a figure and an operation causes implementation defects that need to be complemented. We have implemented 10 complementary modules which complement defects between the modules which add figures (`line`, `rect`, `elli`, `tri` and `oct`) and the modules which add operations (`dump` and `area`), as illustrated in Figure 22.

Although the modules `select`, `delete` and `move` are modules that add operations to the figures, they do not require complementary modules, since these operations are implemented as non-abstract methods of the class `Figure`.

More than 2^9 varieties of applications can be realized because all modules except `base` and `select` can be arbitrarily selected.

6 Related Work

Some traditional languages including Modula-3[1] and Java have *import declarations*, which incorporate names defined at other modules into the current module. In difference-based modules, super-module declarations provide similar function. By traditional import declarations, the programmer can specify individual names to be incorporated. By super-module declarations, all names defined by the super-module are incorporated together. Traditional import declarations only affect the inside of the declaring module, and do not affect the clients of the declaring module. On the other hand, effect of super-module declarations is inherited to all descendant modules. The reader may think these characteristics of super-module declarations lead to the pollution of name space. Actually it does not matter because difference-based modules can minimize the scope of names, and the name collision can be avoided by fully-qualified-names.

Fragment system of BETA language[20] is a module mechanism that is independent from the language core. Fragments are units of reuse and separate compilation. Fragments can inherit more than one fragment (with restrictions). The inheritance relation between fragments determines visibility of names. Separation of specification modules and implementation modules enables information hiding. All characteristics listed above are common with difference-based modules. In fragment system, each language construct in the original program, that is a syntactical notion, can become hook for extension. In difference-based modules, behavior of classes and methods, that is a semantical notion, is the target of extension. In fragment system, the programmer should specify names of hooks and their syntactic category explicitly. Therefore, if programmer wants to provide many hooks, description becomes complicated somewhat. In difference-based modules, the increase of hooks does not make the program complicated because existing class and method names act as hooks.

Virtual classes[21] are mechanism of BETA, which enables extension of inner classes of nested classes by means of overriding by the subclass of the outer class. In order to make applications as extensible as MixJuice does, modules in MixJuice may be expressed as outer classes in BETA and classes in MixJuice may be expressed as inner classes in BETA. Virtual classes are covariant types, which require runtime-check because it is not type safe. On the other hand, MixJuice is type safe because the original class hierarchy and the extended class hierarchy do not exist simultaneously in an application. Virtual classes in `gbeta` language enable family polymorphism[6], which is polymorphism over a group of objects, which is not supported by MixJuice.

A programming technique of mixin layers[27] supports collaboration-based design. A mixin layer is a set of mixins that is related to a collaboration. The programming technique does not require a special language, because it only uses the standard C++ template mechanism. The programming style using mixin layers is quite similar to that in MixJuice; however, mixin layers support neither information hiding nor separate compilation. In addition, the ingenious programming using the template mechanism makes debugging difficult. In MixJuice programming, there are no essential difficulties with debugging.

AspectJ[16], Hyper/J[26], Demeter/Java[19], DJ[25], Adaptive Plug-and-Play Components(AP&PC)[22] and Pluggable Composite Adapters(PCA)[23] are Java based systems which supports separation of crosscutting concerns. All these systems inherit Java's original information hiding mechanism and extend it. In contrast, MixJuice removes it and successfully makes the language specification simpler.

AspectJ[16] treats two kind of crosscutting concerns. One is about dynamic concerns related to the call graph and the other is static concern supported by a mechanism called *introduction*. The former is orthogonal to the MixJuice features and the latter is basically the same as the differential programming mechanism of MixJuice.

Hyper/J[26] is a tool that extracts more than one concern from compiled Java programs and applies them to the other compiled programs. MixJuice solves the problem of "the tyranny of the dominant decomposition", that is pointed out by [26], in a different way. If there are n-dimension of orthogonal concerns, the programmer can divide the source-code into n orthogonal directions. For example, the drawing tool in Section 5 has two dimensions of concerns: data concern and operation concern. The source code of the drawing tool is divided into two orthogonal directions as illustrated in Figure 22.

Hyper/J supports non-invasive extraction of concerns from existing applications; however, MixJuice does not. We think enhancing refactoring[7] tools is more promising approach for extracting concerns from existing applications. MixJuice is a suitable language for this approach because it makes refactoring easier than languages with class-based modules.

Demeter/Java[19] and DJ[25] enables definition of traversing concerns which is independent from concrete tree structures. As described in Section 2.4, it is possible to separate traversing code in MixJuice too.

AP&PC[22] enables definition of collaborations that are independent from concrete class structures. PCA[23] is an enhancement of AP&PC, which supports dynamic application of collaborations. In contrast, current MixJuice does not support dynamic loading of modules.

In mixin layers, Hyper/J and AP&PC, it is possible to apply a collaboration to more than one class hierarchies. On the other hand, it is not possible in the current MixJuice because module definitions include concrete class names to be applied to. This restriction of reusability simplifies definition and composition of modules compared with above systems.

BCA[15] is a system using byte-code translation to enhance the reusability of existing class libraries. The programmers can modify the existing classes without source-code by describing the difference, called *delta files*. This implementation technique is similar to that of MixJuice. Separate type-checking of the delta files has not yet been implemented.

In some object-oriented languages such as CLOS[28] and Smalltalk[9], it is a common programming practice to add methods to existing classes; however, differential extension of existing methods, as in MixJuice, is not common practice. In addition, these languages do not support static type-checking.

Cecil[4], Dubious[24] and MultiJava[5] are object-oriented languages that support multi-methods. They have class-independent module mechanisms that support separate compilation. These languages support a feature called *open class* that enables modules to add methods to existing classes[24]; however, they do not support differential extension of existing methods.

7 Conclusion

We have described a module mechanism, which we call difference-based modules, and an object-oriented language MixJuice. MixJuice is an enhancement of the Java language which adopts difference-based modules in preference to Java's original module mechanism. We have completely separated the class mechanism and the module mechanism, and then unified the module mechanism and the differential programming mechanism. This module mechanism enhances the extensibility, reusability and maintainability of programs. In particular, collaborations, which crosscut several classes, can be separated into different modules that can be developed and tested by independent development teams.

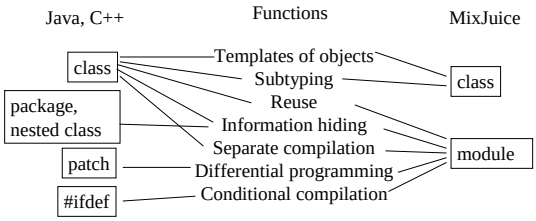

Fig. 23. The functions of classes and modules.

Figure 23 shows how the functions in object-oriented languages are supported by class-based modules and difference-based modules. As shown in Figure 23, the responsibility of classes and modules is clearly separated in difference-based modules. In addition, difference-based modules support differential programming and conditional-compilation features, which are not supported by the language-core of the traditional languages.

We have already written more than 20,000 lines of code in MixJuice. We have not found any major problems with difference-based modules. Their only disadvantage might be readability problems when the code of a class is split into several modules. This problem is the same as the readability problem in the current object-oriented languages that is incurred by code splitting into super-classes and subclasses. Sometimes there is a tradeoff between readability and

reusability. These readability problems should be alleviated by documents using UML or similar, as in current object-oriented languages. Actually, we found that if the roles of classes and methods are made clear, the code becomes more readable, because related codes are located in the same module.

Because the design principles of difference-based modules are very simple, they can be applied to languages other than Java. In addition, because the MixJuice language is still simple, there are numerous possibilities for language extensions, such as introducing parameterized modules.

References

1. Modula–3 home page. `http://research.compaq.com/SRC/modula-3/html/`.
2. Hirotake Abe, Yuuji Ichisugi, and Kazuhiko Kato. An implementation scheme of mobile threads with a source code translation technique in Java. In *IPSJ:PRO*, volume 41, pages 29–40. IPSJ, March 2000. in Japanese.
3. Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. of the OOPSLA/ECOOP '90*, pages 303–311, October 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.
4. Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, November 1995.
5. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *Proc. of the OOPSLA2000*, pages 130–145, October 2000. Published as ACM SIGPLAN Notices, volume 35, number 10.
6. Erik Ernst. Family polymorphism. In *Proc. of the ECOOP'2001*, LNCS 2072, 2001.
7. Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Welsley, 1995.
9. A. Goldberg and D. Robson. *Smalltalk–80: The language and its implementation*. Addison-Wesley, 1983.
10. James Gosling, Bill Joy, Guy L. Steele, and Gilad Bracha. *The Java language specification*. Java series. Addison-Wesley, second edition, 2000.
11. R. Helm, I. Holland, and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object oriented systems. In *Proc. of the ECOOP/OOPSLA'90, Ottawa*, pages 169–180, October 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.
12. Yuuji Ichisugi. EPP home page. `http://staff.aist.go.jp/y-ichisugi/epp/`.
13. Yuuji Ichisugi. MixJuice home page. `http://staff.aist.go.jp/y-ichisugi/mj/`.
14. Yuuji Ichisugi and Yves Roudier. The extensible Java preprocessor kit and a tiny data-parallel Java. In *ISCOPE'97, California*, LNCS 1343, pages 153–160, December 1997.
15. R. Keller and U. Höelzle. Binary component adaptation. In *Proc. of the ECOOP'98*, LNCS 1445, pages 307–329, 1998.
16. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proc. of the ECOOP2001*, 2001.

17. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of the ECOOP'97*, LNCS 1241, pages 220–242, 1997. Invited Talk.
18. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proc. of the POPL'95*, pages 333–343, January 1995.
19. Karl J. Lieberherr and Doug Orleans. Preventive program maintenance in Demeter/Java (research demonstration). In *International Conference on Software Engineering*, pages 604–605, 1997.
20. M. Löfgren, J. Lindskov Knudsen, B. Magnusson, and O. Lehrmann Madsen. *Object-Oriented Environments - The Mjølner Approach*. Prentice Hall, 1994.
21. Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes - a powerful mechanism in object-oriented programming. In *Proc. of the OOPSLA'89*, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.
22. M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proc. of the OOPSLA'98*, pages 97–116, October 1998.
23. Mira Mezini, Linda Seiter, and Karl Lieberherr. Component integration with pluggable composite adapters. In Mehmet Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer Academic Publishers, 2000. University of Twente, The Netherlands.
24. Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *Proc. of the ECOOP'99*, LNCS 1628, pages 279–303, 1999.
25. Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, LNCS 2192, pages 73–80, Kyoto, Japan, September 2001.
26. H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proc. of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer*, 2000.
27. Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *Proc. of the ECOOP'98*, LNCS 1445, pages 550–570, 1998.
28. G.L. Steele. *Common Lisp the Language, 2nd edition*. Digital Press, 1990.
29. Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, third edition, 1997.
30. C.A. Szyperski. Import is not inheritance – why we need both: Modules and classes. In *Proc. of the ECOOP'92*, LNCS 615, 1992.
31. Michael VanHilst and David Notkin. Using role components to implement collaboration-based designs. In *Proc. of the OOPSLA'96*, October 1996. Published as ACM SIGPLAN Notices, volume 31, number 10.