Difference-based modules
and MJ language

National Institute of Advanced Industrial Science and Technology
/ Japan Science and Technology Corporation

Yuuji Ichisugi
http://staff.aist.go.jp/y-ichisugi/mj/
Nov 15, 2001
Programming Language MJ

• Enhanced Java language with simple and powerful module mechanism: difference-based modules

• Target:
  – Highly extensible software
Class is not Module
Difference is Module

A module is:

- Difference between the original system and the extended system
- A unit of information hiding
- A unit of reuse
- A unit of compiling
Separation of crosscutting concerns

Codes crosscutting more than one class can be separated into modules.
Plug-and-play of modules

- End users can select and combine existing modules without writing any lines of "glue code."
Outline of this Presentation

• Problems of current object-oriented languages
• Module mechanism of MJ language
• The collision problems
• The targets of MJ language
Problems of current object-oriented language
Classes are not modules

• A class is inappropriate as:
  – a unit of information hiding
  – a unit of reuse
“class” is inappropriate as a unit of information hiding.

- class = module is only approximately true. [Szyperski ECOOP92]
  - To alleviate this problem, protected, package and nested classes are introduced, however,
  - there are still problems.
Example:
java.util.HashMap, TreeMap

- The internal of each file is encapsulated well, however, …

HashMap.java (500 lines)
- class HashMap
  - class Entry
  - class HashIterator

TreeMap.java (1000 lines)
- class TreeMap
  - class SubMap
    - class EntrySetView
  - class Entry
  - class Iterator

nested classes
Internal of HashMap, TreeMap

• Not modular at all! Classes depend on each other.
• Because of the limitation of the module mechanism.

HashMap

Entry
HashIterator

HashMap.java(500 lines)

TreeMap

SubMap

EntrySetView

Iterator

Entry

TreeMap.java(1000 lines)
“class” is inappropriate as a unit of reuse

• “Separation of crosscutting concerns” is not supported by the current OOPL.

• Various approaches
  AspectJ [Kiczales 99]
  Hyper/J [Ossher ICSE 99]
  Mixin layers [Smaragdakis ECOOP98]
  BCA [Keller ECOOP98]
  adaptive p-n-p [Mezini OOSPLA98]
  collabolation-based design [VanHilst OOPSLA96]
  ...
  Subject-oriented programming [Ossher OOPSLA92] [Ossher OOPSLA93]

• However, each system has some drawbacks
  – E.g. No information hiding, no separate compilation, no type checking, complexity of language specification
Module mechanism of MJ
Difference-based modules

- “module” describes the difference between “original system” and “extended system”
- Like “patch” files which can be type-checked and separately compiled
The way to add the differences

• A module can:
  – Add new classes
  – Add fields and methods to existing classes
  – Override existing methods

• All classes and methods are “hooks” for extension
Module definition

module m2
  extends m1 // super module of module "m2"
  {
    // body of this module (the difference)
    define class A {…} // definition of new class
    class B {…} // extension to existing class
  }

Syntax of the inside of "class" is almost the same as Java
module s {
  define class S {
    define int m() { return 1; }
  }
}

module a extends s {
  class S {
    // Overriding existing method
    int m() { return original() + 2; }
  }

  // Add subclass A
  define class A extends S { … }
}

module b extends s {
  class S {
    // Overriding existing method
    int m() { return original() + 3; }
  }

  // Add a subclass B
  define class B extends S { … }
}
Linking of modules

End-user selects their favorite modules

```
class S {
    int m() { return (1 + 2) + 3; }
}
```

class A extends S {
    ...
}

class B extends S {
    ...
}

Link result is the same as conventional object-oriented languages (Java).
Separation of crosscutting concerns

- Modularization of code independently of classes
- Cf. Aspect-oriented programming [Kiczales 97]
High re-usability

Current OOP:
Programmers can only make subclasses

MJ:
Modules written by third-party are composable (like mixins)

Framework
- Modules written by user A
- Modules written by user B
- Modules written by user C
- ...

System of user A
System of user B
Plug-and-play module

- End-user can select and combine their favorite modules
  - Not need to specify how to link the modules
Multiple inheritance of name space

- Names are visible from sub-modules
- Flexible name space control mechanism

nested name spaces

overlapping name spaces
HashMap.java written in MJ

No cyclic dependency between modules

Class name of HashMap

External interface of HashMap

Fields of HashMap, class name, fields, external interface of HashMapEntry

Class name and external interface of HashMapIterator

Methods of HashMap, Modules of HashMapEntry

Methods of HashMap using HashMapIterator

Methods and fields of HashMapIterator
Summery of module mechanism of MJ

• High extensibility
  – All classes and methods are hooks for extension.

• High re-usability
  – Composable modules like mixins.

• Flexible modularization
  – Classes and modules are completely orthogonal
  – Separation of crosscutting concerns

• Flexible name space control
  – Multiple inheritance of name space
  – Arbitrary name space boundary
Simple and powerful module mechanism
Classes and methods are completely orthogonal

Java, C++

Function
- Templates of objects
- Type definition
- Name space
- Separate compilation
- Reuse
- Differential programming
- Conditional compilation

MJ
- class
- module

(Security)
(Mutual exclusion)
Implementation

module m1

class A

class B

module m2

class B1

class A

class A2

class B1

class B

Source-code Translation (compile time)

Byte-code Translation (link time)

Super classes are changed by the linker. ²⁶
Third party problems
“Third party problems”

- General problems in extensible systems

Correct program + Correct program = Sometimes not works!
The cause of third party problems

• Three categories of problems:
  – Name collision
  – Implementation defect
  – Semantic collision
Name collision problem

• Solved by “full qualified name of methods”

Separately defined names are distinct
Implementation defect problem

module m0{
    define class S {}
    define class A extends S {}
}

module m1 extends m0{
    define class B extends S {}
}

module m2 extends m0{
    class S { define abstract void m();}
    class A { void m() {…}}
}

Add new subclass

Add new abstract method

Link time error if both m1 and m2 are selected because method m of class B is not implemented.
Implementation defect problem (contd.)

- Two different directions of extensions will produce implementation defects.
- Someone need to know exact specification of both m1 and m2 to complement the defect.
Previous Solutions

• Limit the way of extension to the only one direction
  – Avoids implementation defect
  – Restricts extensibility

• Enforce the definition of “default behavior” [Millstein, Chambers ECOOP99]
  – Prohibit adding abstract method to super class
  – Pass through type checking at link-time
  – Difficult to define “correct default behavior”
An example of implementation defect in the real world

- Someone should complement the defect
  (= device drivers)

<table>
<thead>
<tr>
<th>Devices</th>
<th>PC</th>
<th>Windows</th>
<th>Linux</th>
<th>Solaris</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mouse</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Video card</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAN card</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>…</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OS
Complementary module

- m3 (implemented by someone) will be found from CLASSPATH and automatically linked if both m1 and m2 are selected.

```
module m0{
  define class S {}
  define class A extends S {}
}

module m1 extends m0{
  define class B extends S {}
}

module m2 extends m0{
  class S { define abstract void m() {...} }
  class A { void m() {...} }
}

module m3 complements m1,m2 {
  class B {
    void m() {...}
  }
}
```

Complement “defect” between m1 and m2.
Semantic collision problem

• Assumption: Design by Contract [Meyers ’92]
  – Implement modules to meet the specifications
  – Use modules only assuming the specifications of them

• The module which doesn’t meet specification causes problems
  – assertion checking (not implemented yet)
  – “Design by Contract” in languages which support Mixins has not studied.
    • Now investigating.
Summary of third party problems

• Name collision
  – Avoided by Fully qualified name of method

• Implementation defect problem
  – Detected by the linker
  – Complemented by complementary modules

• Semantic collision problem
  – Detected by assertion checking at run time.
  – Detected by the linker: Future work
Target of MJ
Target 1: Highly extensible software

- Ex) Extensible pre-processor EPP 1.1
  - Practical software realized by difference-based modules (Currently not written in MJ)
  - [http://staff.aist.go.jp/y-ichisugi/epp/](http://staff.aist.go.jp/y-ichisugi/epp/)
Target 2: PDA Application

• Applications on PDA
  – Cellular phone, PIM, mail, Remote-controller, game, …

• Contradiction of demands
  – Various needs depending on users
  – Limit of hardware resource

• Using MJ, the vendors can:
  – Provide infinite variation of applications
  – Minimize the down-load size
Target3: Software development in distributed environment

- Software development in a Open Source Community
  - linux, GNU

- The problem in integrating various versions
  - Low level tools (diff/patch, cvs)

- MJ supports integrating software at language-level
Related work (1/2)

• Mixin Layers [Smaragdakis ECOOP 98]
  – Programming style using Template of C++.

• AspectJ [Kiczales 99]
  (Aspect oriented programming)
  – Support “separation of concerns.”

• Hyper/J [Ossher ICSE 99]
  – Tools for multi-dimensional separation of concerns.

• BCA [Keller ECOOP 98]
  – Support “delta files” by byte-code translation

• JAM [Ancona ECOOP 00]
  – Java with mixins. Static type-checking.
Related work (2/2)

- Cecil [Chambers TOPLAS 95]
  Dubious [Millstein ECOOP 99]
  - Multi-method and open-classes.
  - Module mechanism without link time error.
- MultiJava [Clifton OOPSLA 2000]
  - Java extension with multi-method and open-classes
Research schedule

2000  10  |  2001  4  7  10  |  2002  1  4  7  10

MJ beta version

Prototype framework of source code translator

EPP Ver.2 on MJ

MJ on EPP Ver.2
Summary

• The difference is module, class is not module
• Separation of crosscutting concerns
• plug-and-play module
• Multiple inheritance of name spaces
• Fully qualified name of fields and methods
• Implementation defects and complementary modules