Difference-based Modules: A Class-Independent Module Mechanism

Yuuji Ichisugi, Akira Tanaka
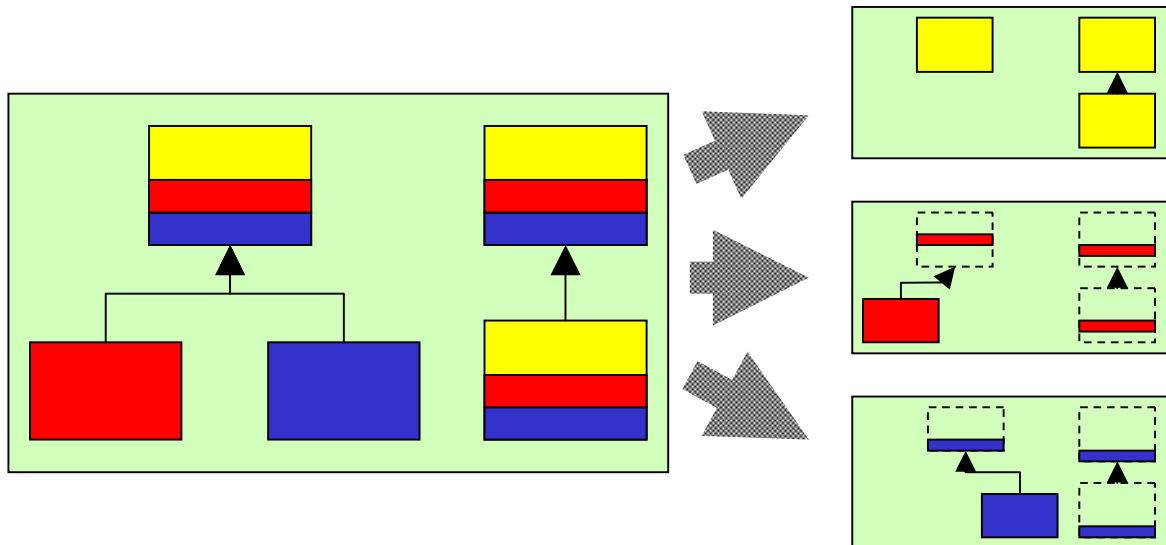
National Institute of Advanced Industrial Science and Technology

http://staff.aist.go.jp/y-ichisugi/mj/

June 12,2002

Difference-based modules

- **Simpler** than Java's module mechanism
  - "protected" and "nested classes" are no longer needed
- **Better** extensibility, re-usability
- Separation of cross-cutting concerns

Programming Language MixJuice

- An enhancement of Java with difference-based modules.

- Distributed with source-code.
  - http://staff.aist.go.jp/y-ichisugi/mj/

- We have already written more than 20,000 lines of code.

- From my experience,
  MixJuice programming is happy!

Outline of this presentation

- Problems of current object-oriented languages
- Difference-based modules
- Collision problems
  - implementation defect phenomenon
    on extensible systems

$$m_a \; + \; m_b \; > \; m_c \; + \; m_d$$
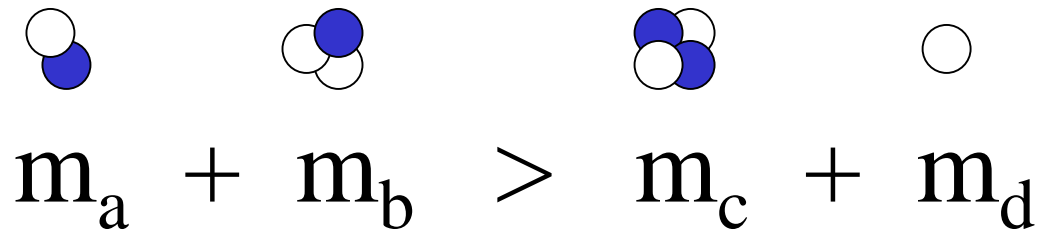
Problems of current object-oriented languages

Classes are not modules

- Classes are templates of objects.
- Modules are units of reuse and information-hiding.
- A class is inappropriate as:
  - a unit of reuse
  - a unit of information hiding
- Pointed out by many researchers.

"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,

  – language specification becomes complex and non-intuitive !

    - Especially Java's information hiding mechanism is complex.

"class" is inappropriate as a unit of reuse

- "Separation of crosscutting concerns" is not supported by the current OOPL.

- Various approaches which *extend* OOPL.

    AspectJ[Kiczales 99]

    Hyper/J [Ossher ICSE 99]

    Mixin layers[Smaragdakis ECOOP98]

    BCA[Keller ECOOP98]

    AP&PC[Mezini OOSPLA98]

    collabolation-based design[VanHilst OOPSLA96]

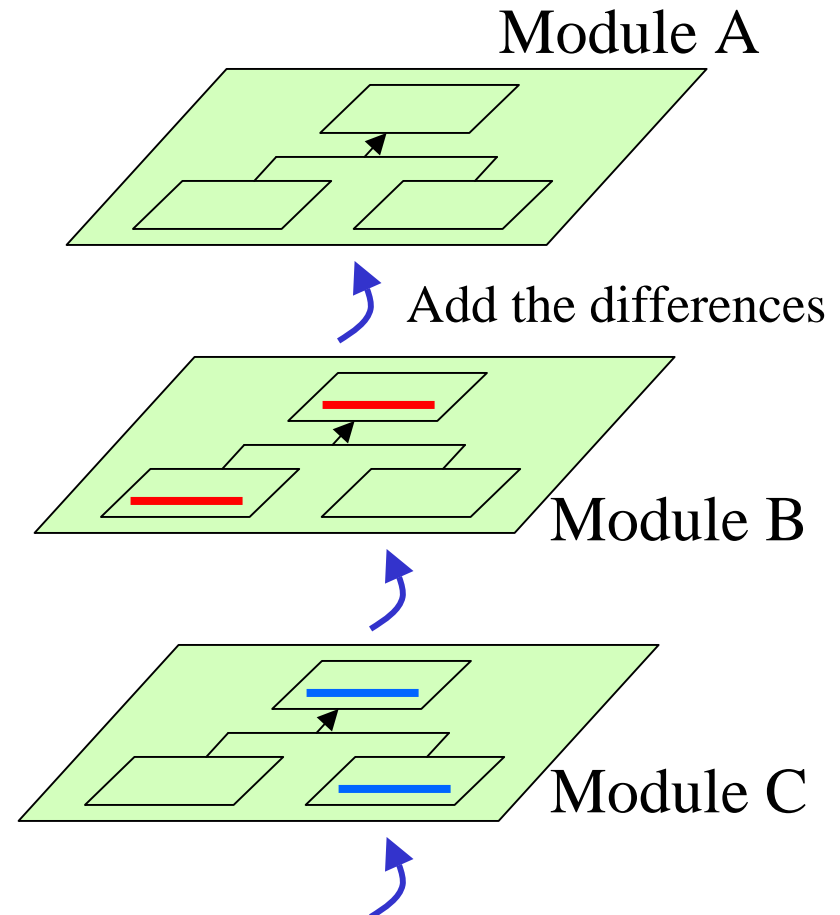   Subject-oriented programming [Ossher OOPSLA92] [Ossher OOPSLA93]

    etc.

Difference-based modules

Class is not Module
Difference is Module

- Simple design principle: "module" describes the difference between "original program" and "extended program"
- Like "patch" files
- separately type-checked and compiled

Module A

Add the differences

Module B

Module C

What is "difference" ?

- A module can:
  - Add new classes
  - Add fields and methods to existing classes
  - Override existing methods
- All classes and methods are "hooks" for extension

Original system
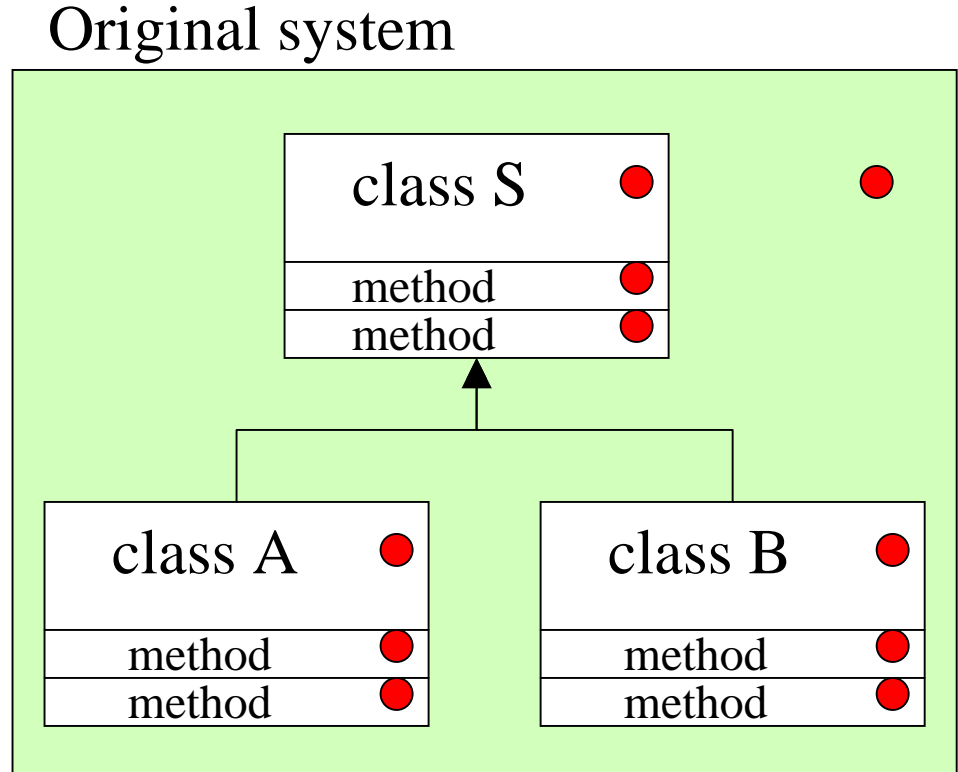
● "hook" for extension

Module definition

```
module m2 extends m1
{
  define class A {…}   // Addition of new class
  class B {…}   //Extension of existing class
}
```

Code Example

```
module m1 {
  define class S {
    define int foo(){ return 1; }
  }                      // 1
  define class A extends S {
    int foo(){ return original() + 10; }
  }                      // 11
}
```
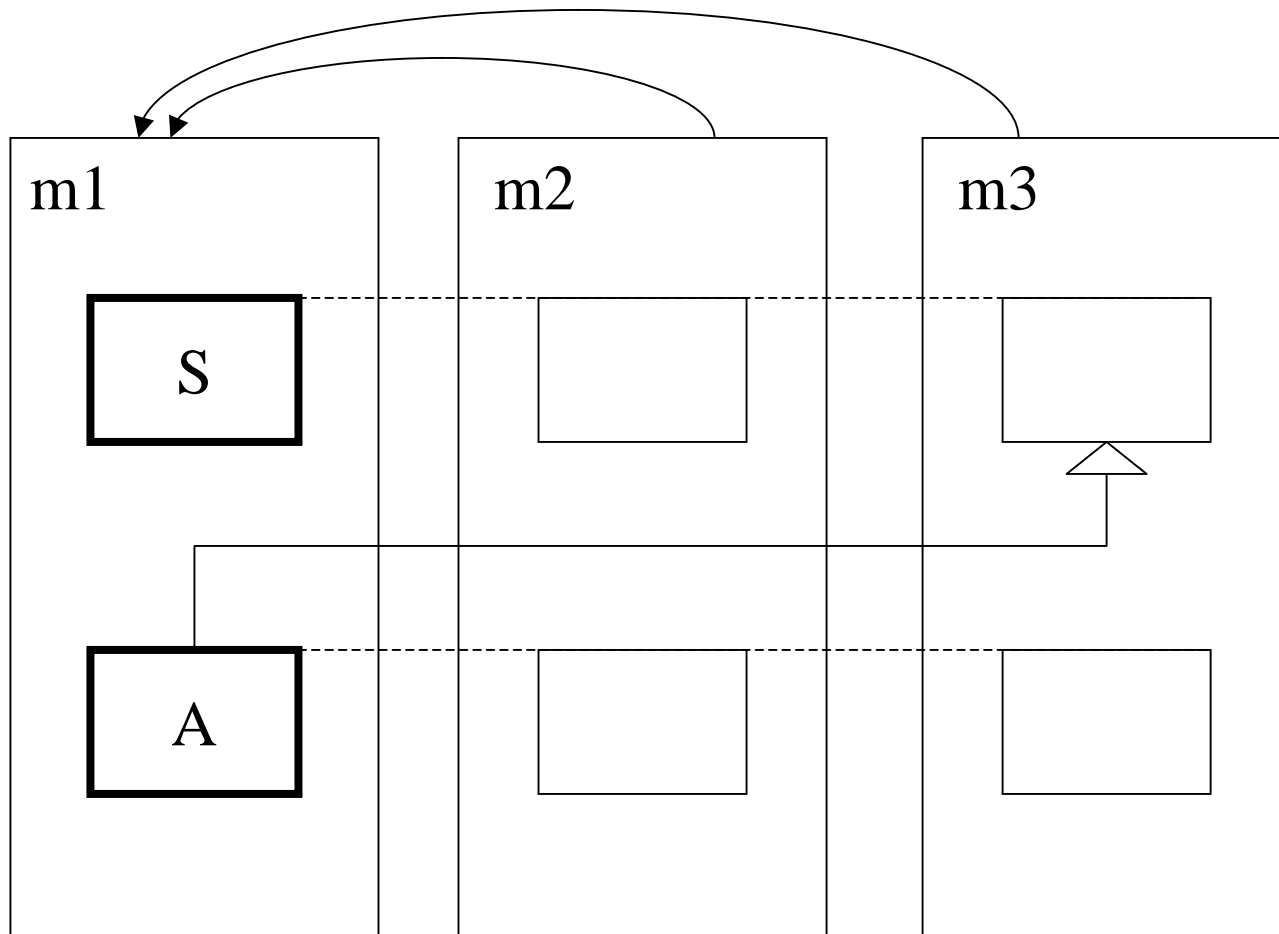
m2 and m3 are separately type-checked and compiled

```
module m2 extends m1 {
  class S {
    int foo(){ return original() + 2; }
  }              // 3
  class A {
    int foo(){ return original() + 20; }
  }              // 33
}
```

```
module m3 extends m1 {
  class S {
    int foo(){ return original() + 3; }
  }              // 4
  class A {
    int foo(){ return original() + 30; }
  }              // 44
}
```
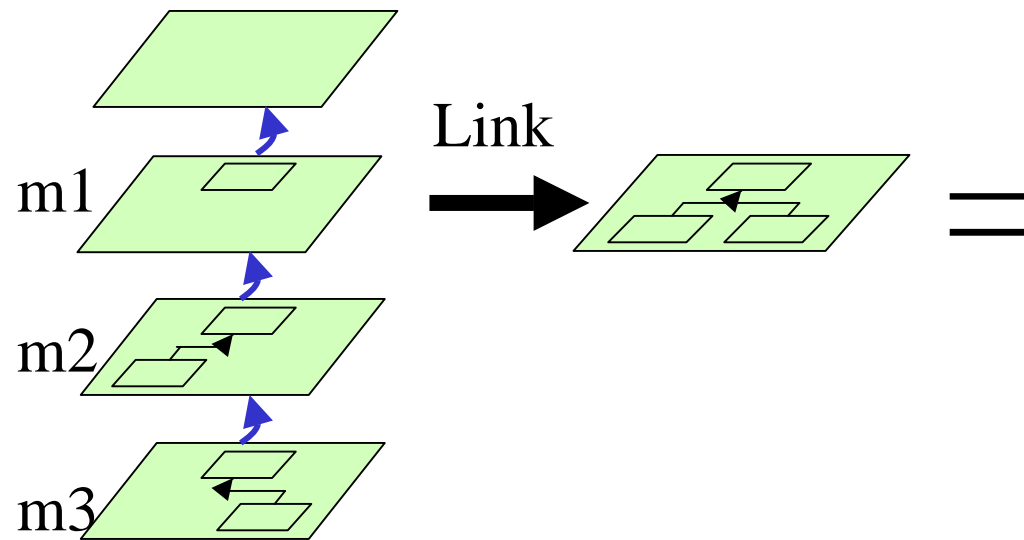
13

Layered class diagram

- Extension of UML class diagram
- Useful notation for AOP languages

Linking of modules

Linearize modules
and add differences
defined by the modules

m1

m2

m3

Link

```
class S {
  int foo(){
    return (1 + 2) + 3;
  }
}
class A extends S {
  int foo(){
    return ((super.foo() +
            10) + 20) + 30;
  }
}
```

Link result is the same as conventional
object-oriented languages(Java).

Advantages of difference-based modules

- High extensibility of applications
- Class-independency of modules
- Extensibility by third party programmers
- Module-composability by end-users
- Flexibility of module grouping
- Flexibility of name space structures
- Ease of code-moving
- Simplicity

Separation of crosscutting concerns

- Like AOP languages [Kiczales 97]
- Naturally achieved because class and module are completely orthogonal in MixJuice.
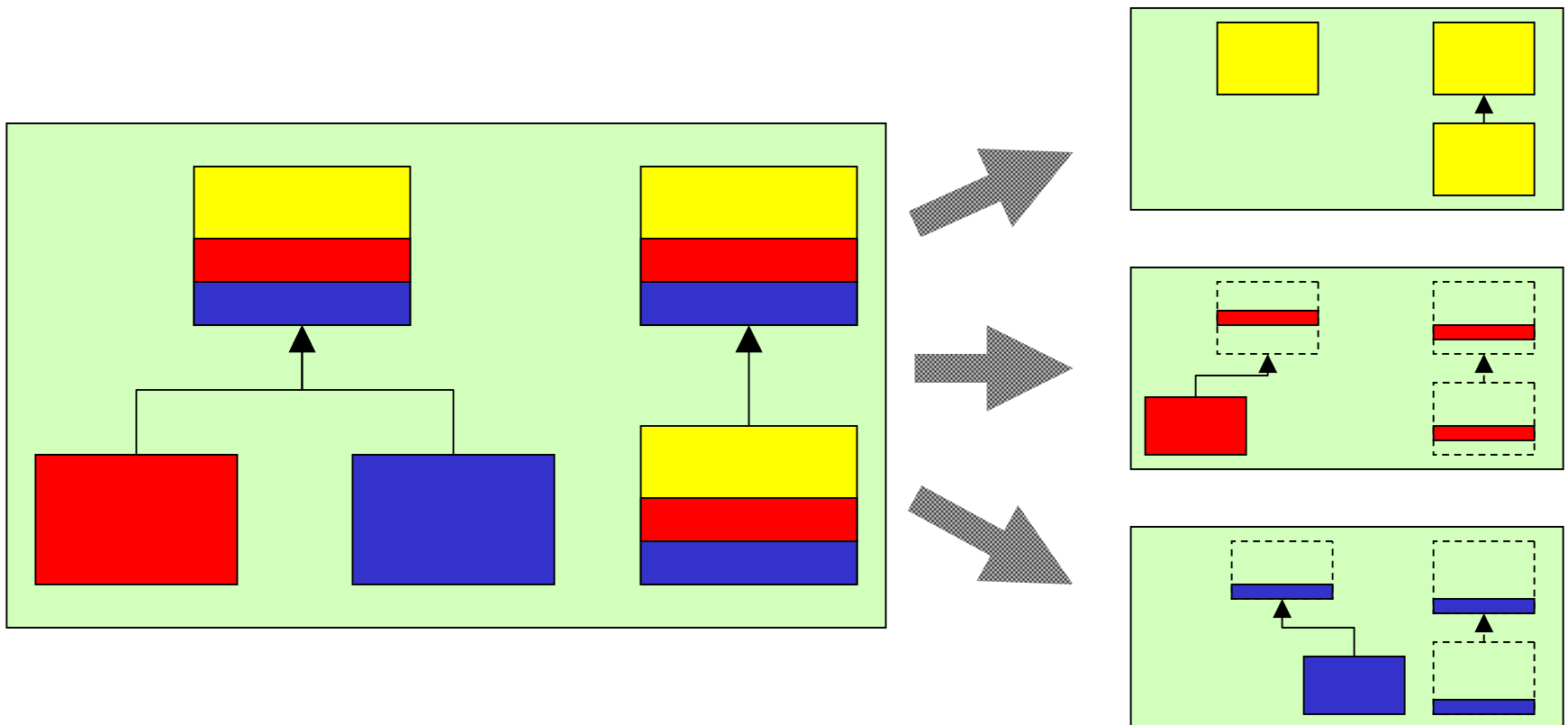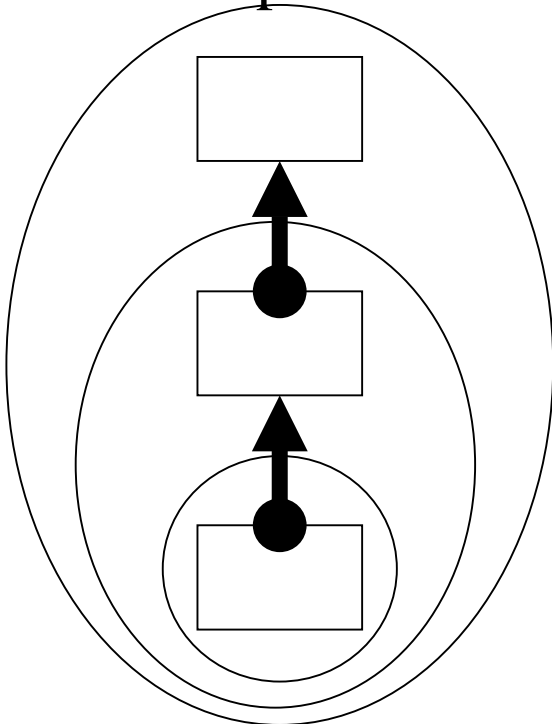
No "glue code" needed

- End-users of applications can select and link modules to build their own customized applications without detailed knowledge.
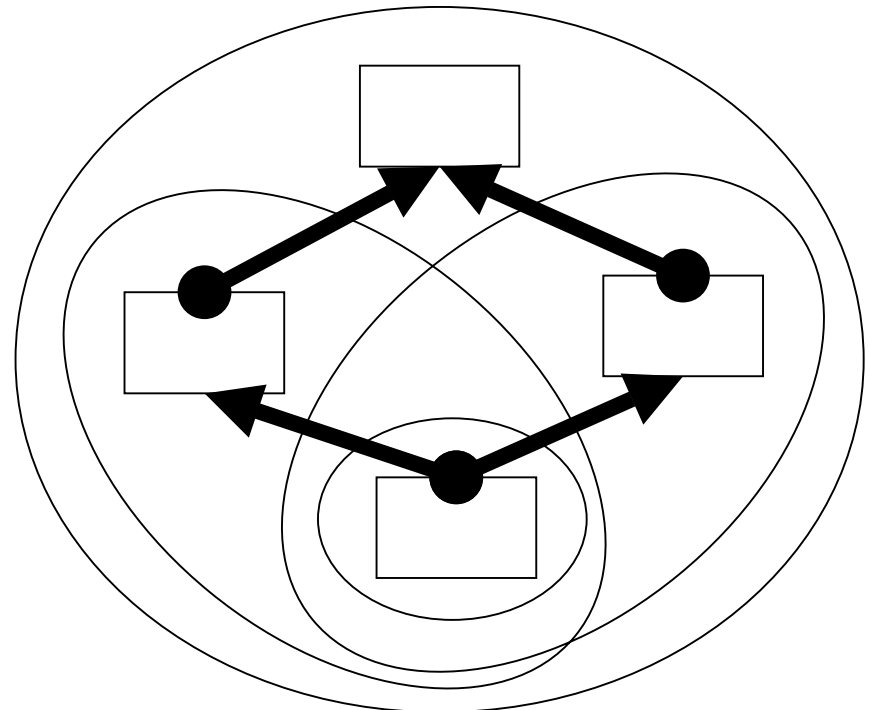

Selected modules

Linked Application
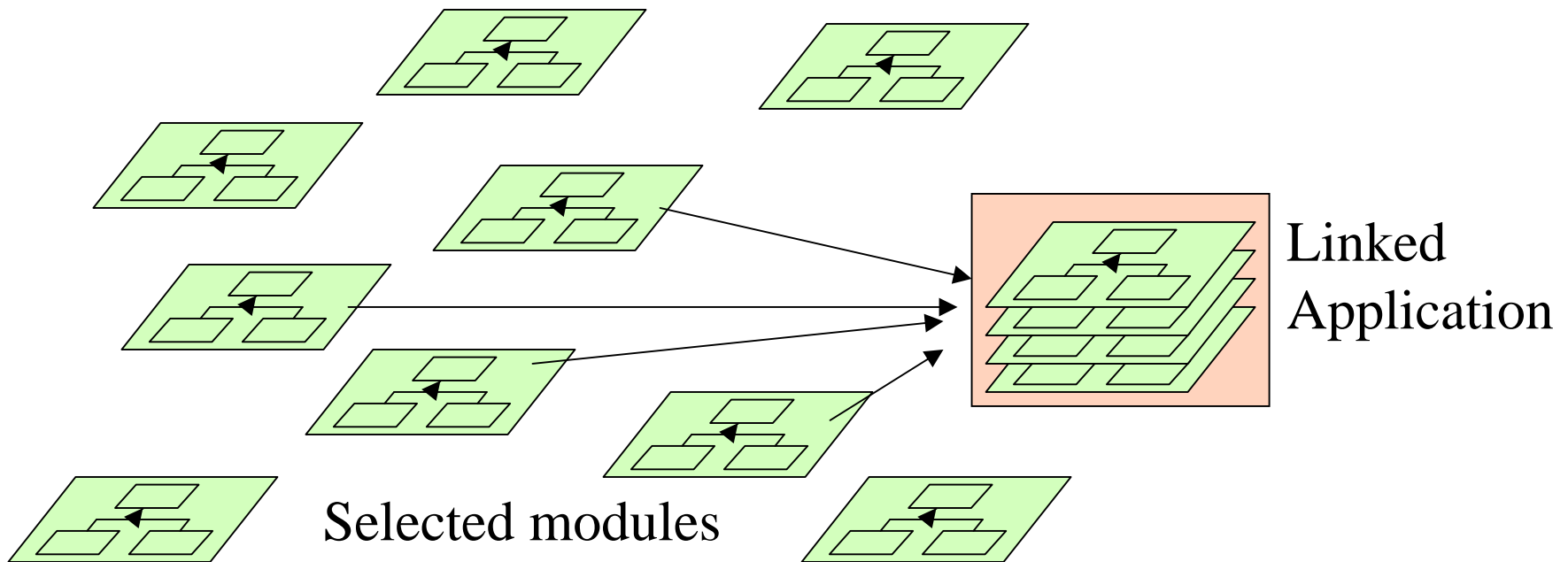
"nested classes" are no longer needed

- Names are visible from all descendant-modules
- More flexible name spaces than nested classes.
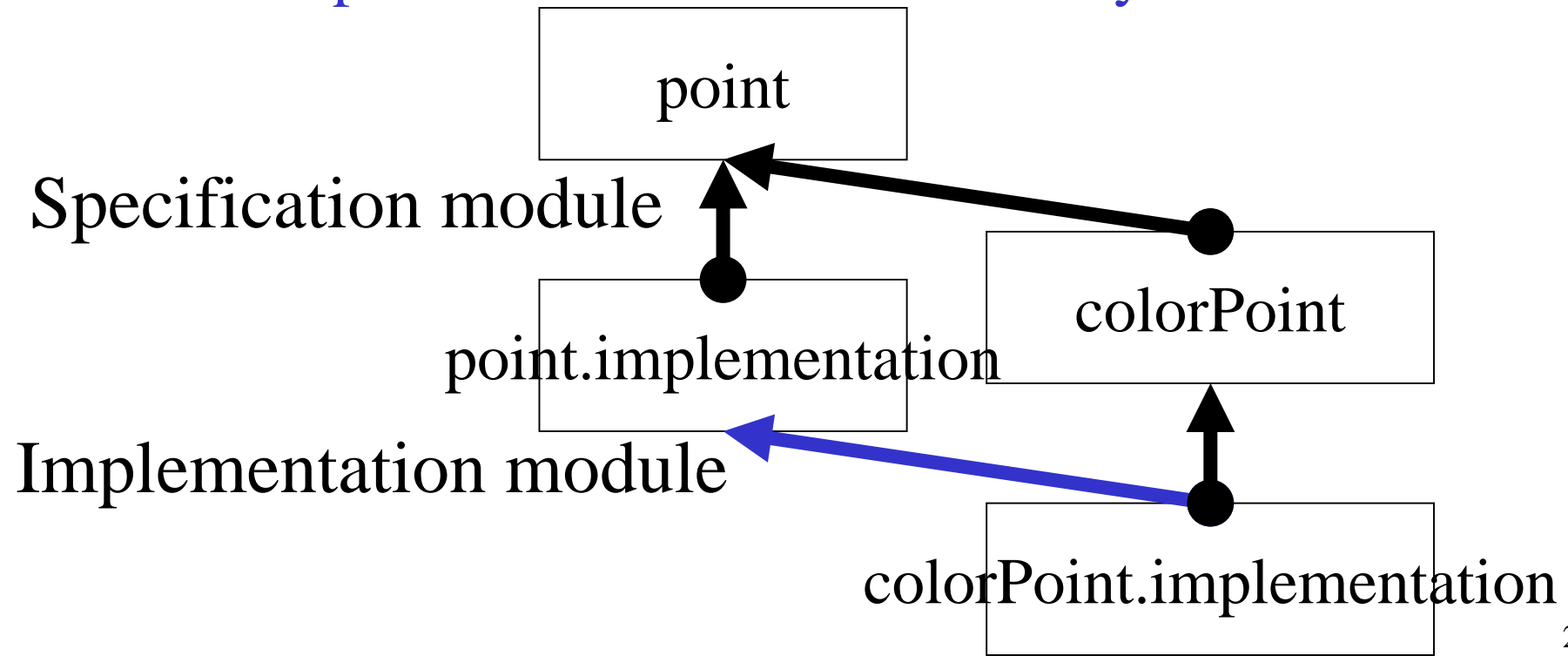
nested name spaces

overlapping name spaces

"protected" is no longer needed

- Both black-box reuse and white-box reuse can be expressed by module inheritance.
- More flexible than "protected" because subclass may inherit super class in black-box reuse style.

point

Specification module

point.implementation

colorPoint

Implementation module

colorPoint.implementation

The functions of classes and modules

Java, C++   Functions   MixJuice

class ── Templates of objects

class ── Subtyping ── class

Reuse

package, nested class ── Information hiding

patch ── Separate compilation ── module

#ifdef ── Differential programming

Conditional compilation
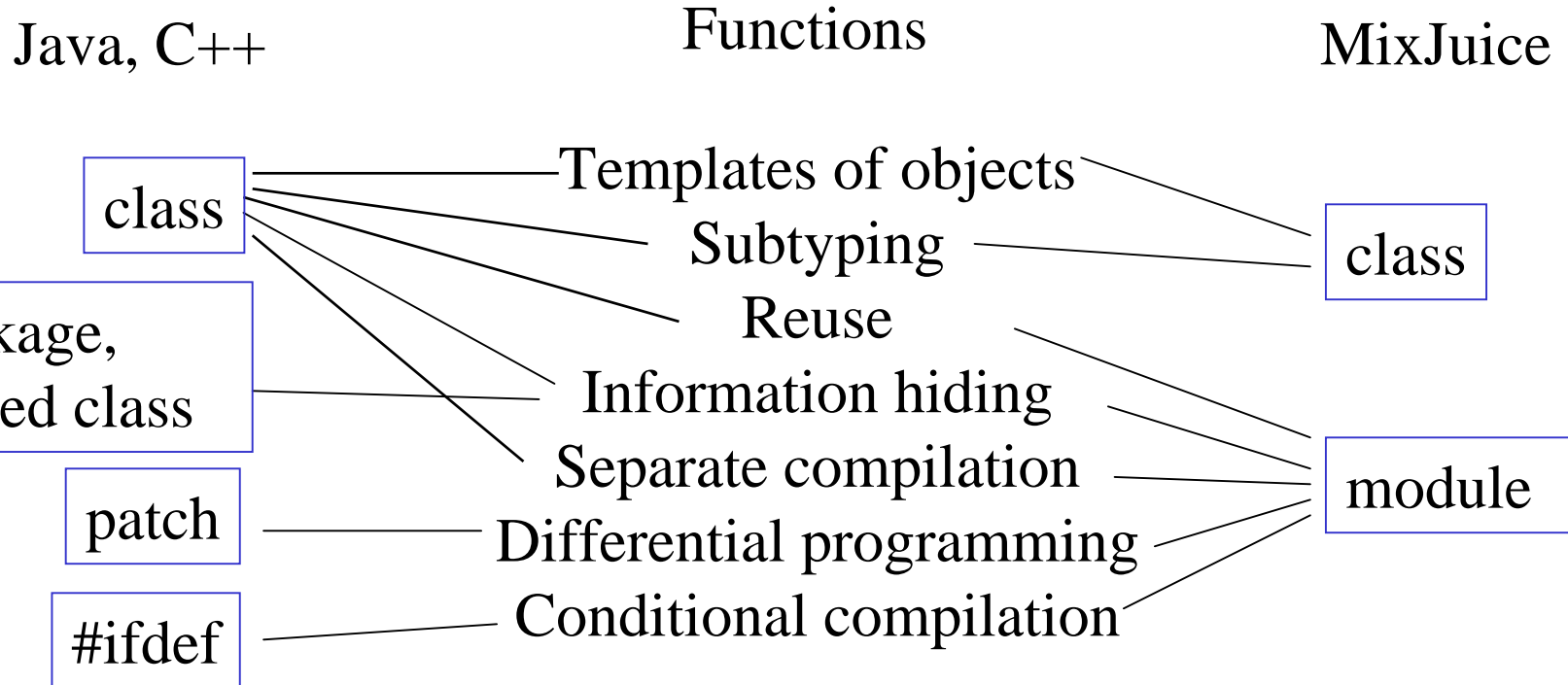
Collision problems

Collision problems

- General problems in extensible systems

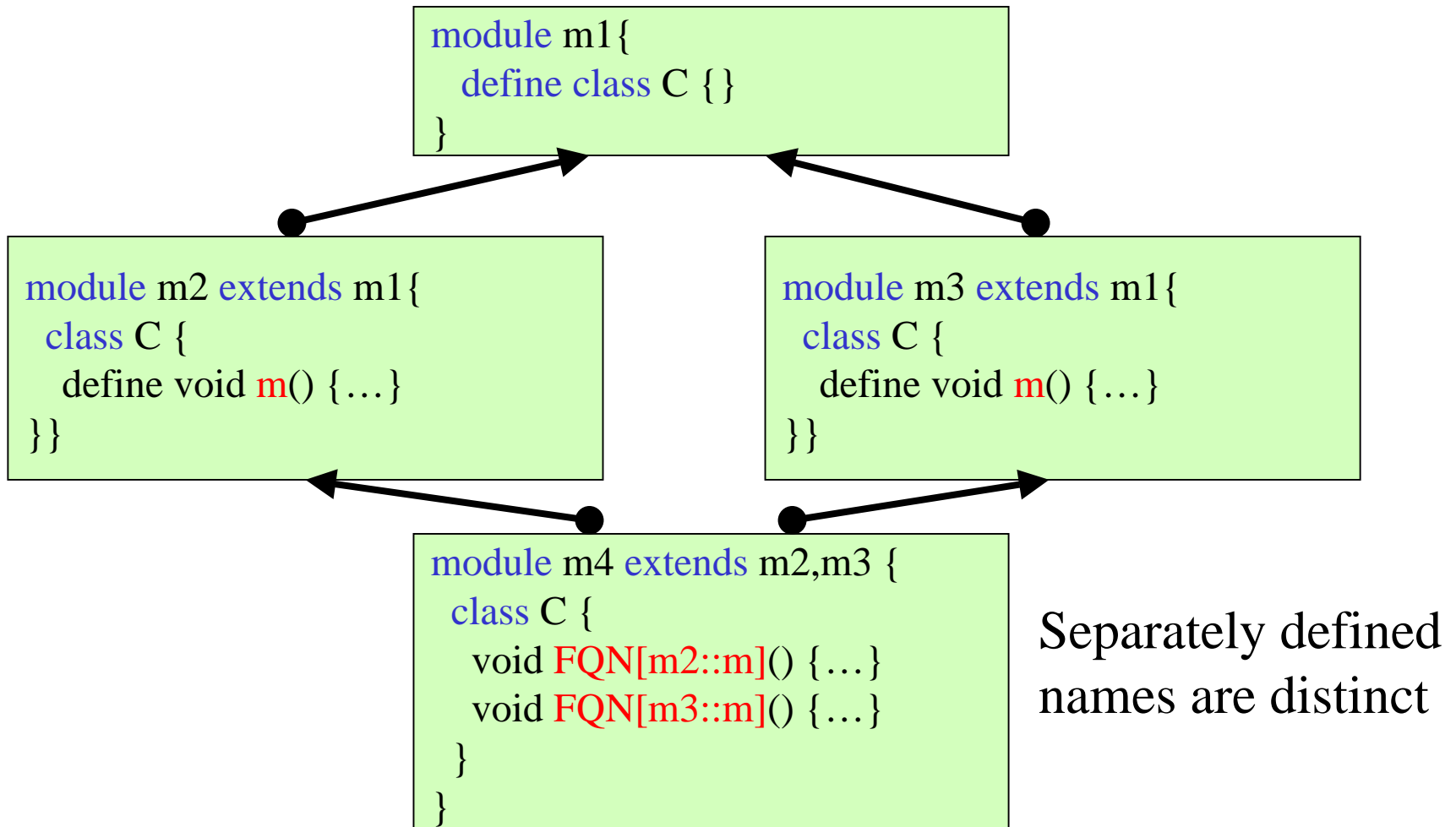Correct program   Correct program   Sometimes not works!
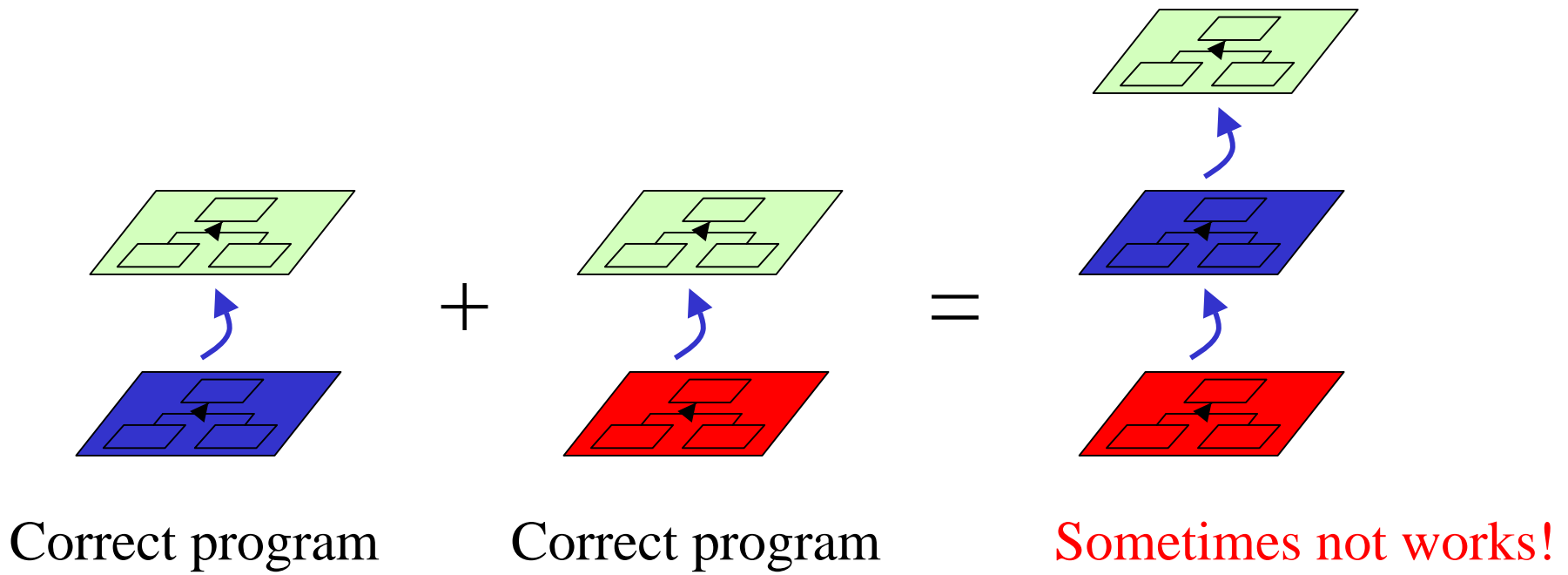
The cause of problems

- We have categorized problems into three types.
  - 1. Name collision
  - 2. Implementation defect
  - 3. Semantic collision
- **We give solutions to all these problems.**

1. Name collision problem

- Solved by "fully qualified name of methods"


```
module m1{
   define class C {}
}
```

```
module m2 extends m1{
  class C {
   define void m() {…}
}}
```

```
module m3 extends m1{
  class C {
   define void m() {…}
}}
```

```
module m4 extends m2,m3 {
  class C {
   void FQN[m2::m]() {…}
   void FQN[m3::m]() {…}
  }
}
```

Separately defined names are distinct

2. Implementation defect

```
module m1{
    define abstract class S {}
    define class A extends S {}
}
```

Add new subclass

```
module m2 extends m1{
    define class B extends S {}
}
```

Add new
abstract method

```
module m3 extends m1{
    class S { define abstract void m();}
    class A { void m() {…}}
}
```
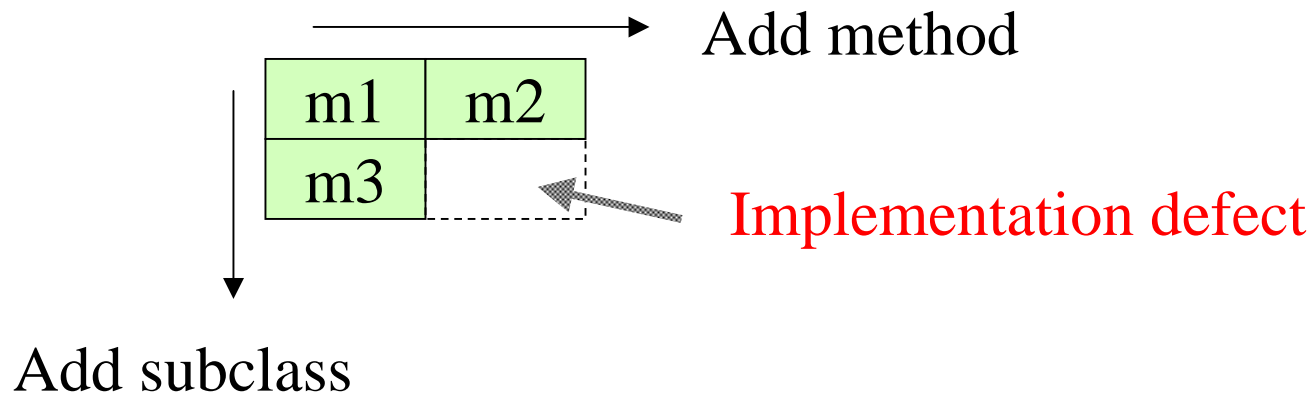
If both m2 and m3 are composed link time error occur
because **method m of class B** is not implemented.

Implementation defect (contd.)

- Two different directions of extensions will produce implementation defects.

- Someone who knows the specification of both m2 and m3 should complement the defect.

Add method

m1	m2
m3	

Implementation defect

Add subclass

An example of implementation defect in the real world

- Someone should complement the defect
  - = Someone should supply device drivers

	PC	Windows	Linux	Solaris	…
	Mouse				
	Video card		device drivers		
	LAN card				
	…				

OS →

Devices ↓

28
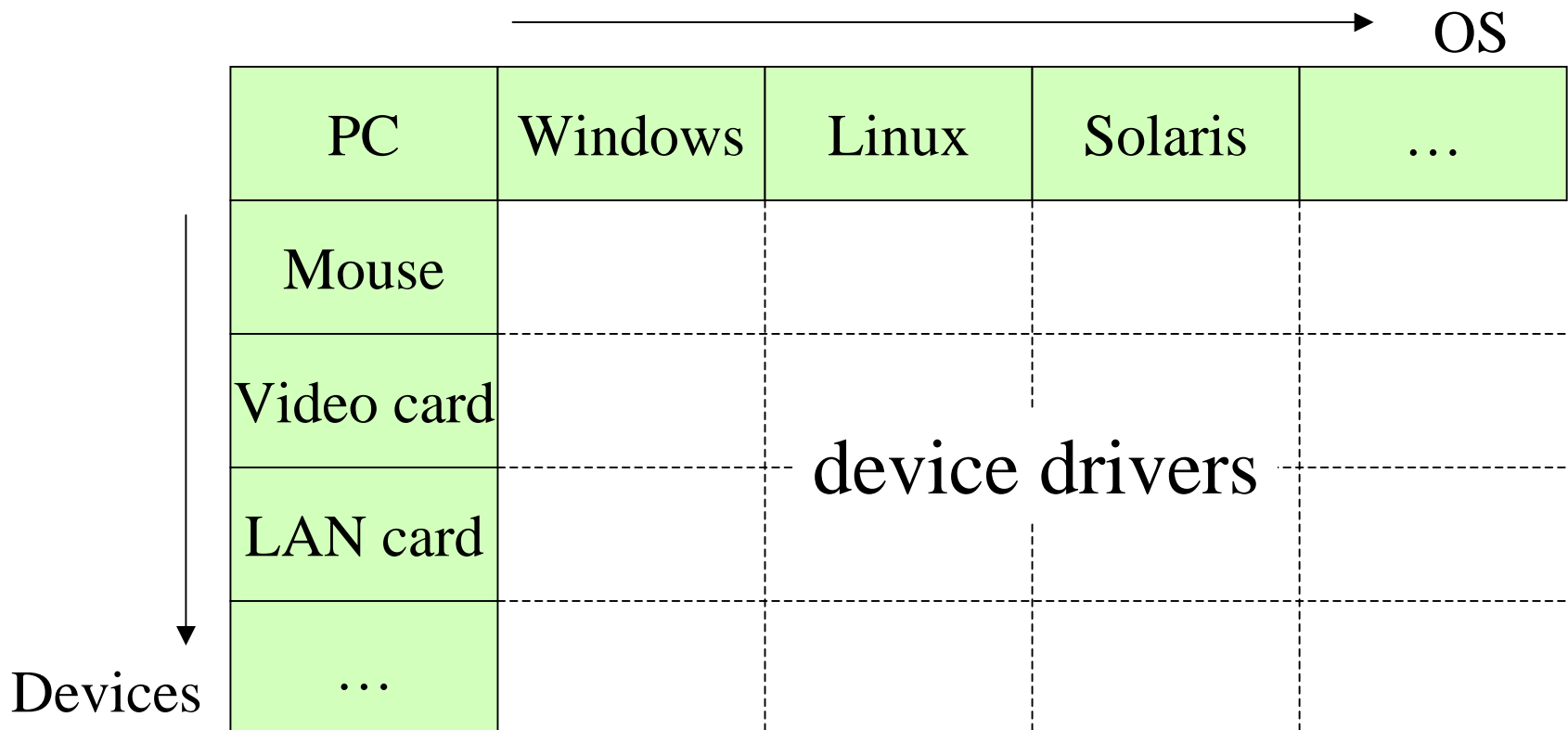
Language support: Complementary module

- m4(implemented by someone) will be found from CLASSPATH and automatically linked if both m2 and m3 are specified for the linker.

```
module m1{
    define class S {}
    define class A extends S {}
}
```

```
module m2 extends m1{
  define class B extends S {}
}
```

```
module m3 extends m1{
    class S { define abstract void m() {…}}
    class A { void m() {…}}
}
```

```
module m4 complements m2,m3 {
  class B {
    void m() {…}
  }
}
```

Complement "defect" between m2 and m3.
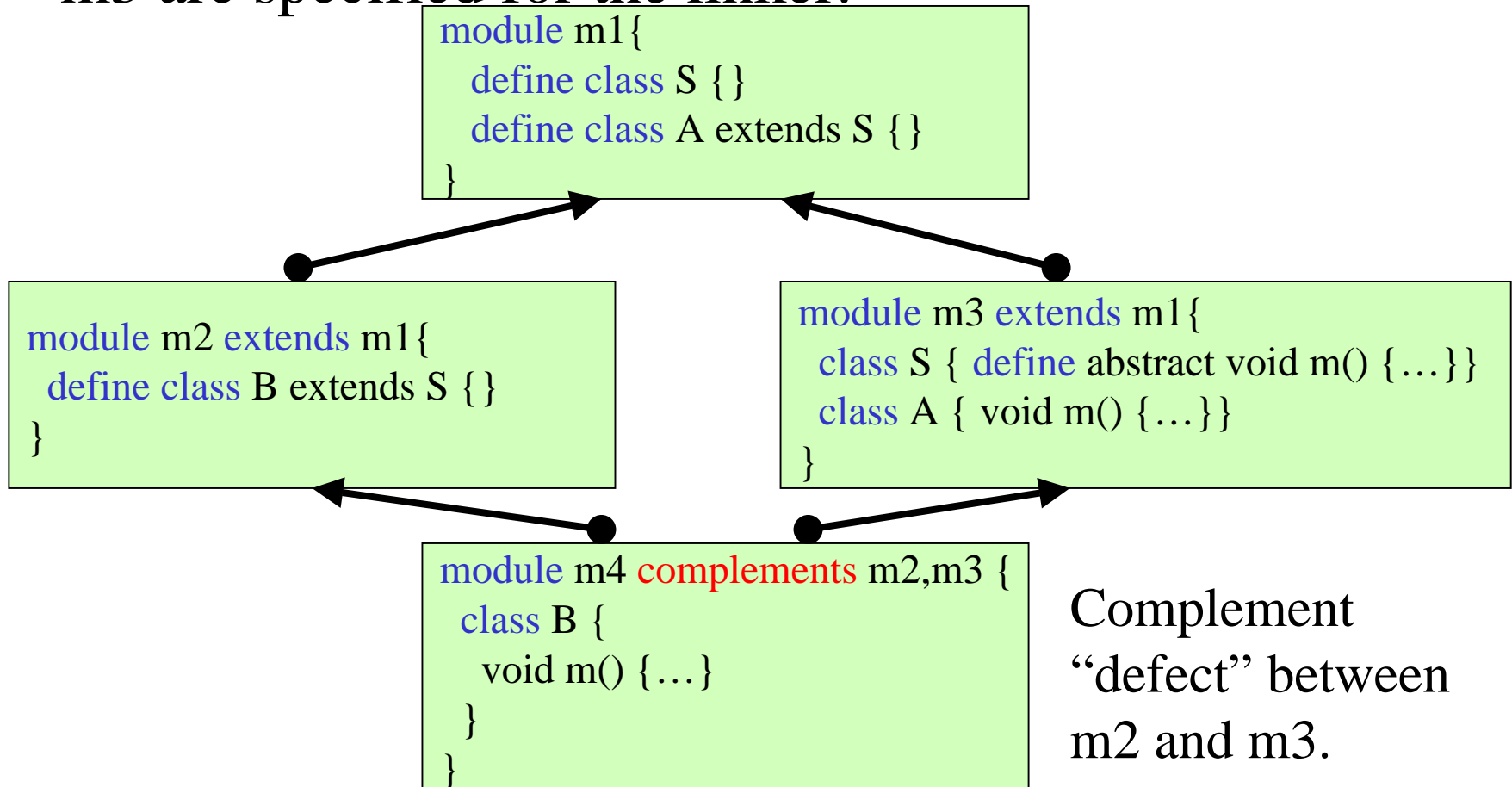
3. Semantic collision

- On-going research.

- Applying the ideas of **Design by Contract** [Meyers '88] and **behavioral subtyping** [Meyers '88][Liskov '94] to mixins and difference-based modules.

- If specification of each modules conforms to "**a rule**", they can be safely combined.

- The correctness of the rule will be formally proved.

Related Work

- These systems have achieved similar things:
  - Fragment system / virtual classes of BETA
  - AOP languages
    - AspectJ, Hyper/J, Demeter/Java, Composition Filters, …
  - Collaboration-based languages
    - Mixin Layers, AP&PC, PCA, delegation layers,…
  - Open-class languages
    - CLOS, Smalltalk, Objective-C, Cecil, Dubious, MultiJava, Ruby…
- None of these systems have intended to *simplify* existing information-hiding mechanism.

Application
Extensible Java pre-processor EPP

- Framework for composable language extension
- ECOOP2002 poster session

Operator
overloading

Assertion

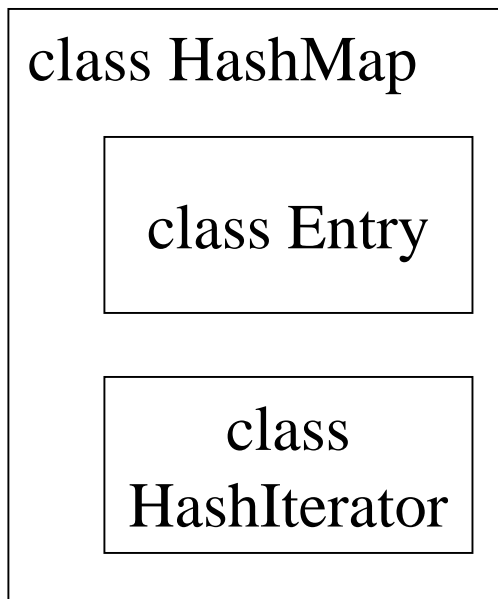Module
mechanism

Parameterized
types

EPP
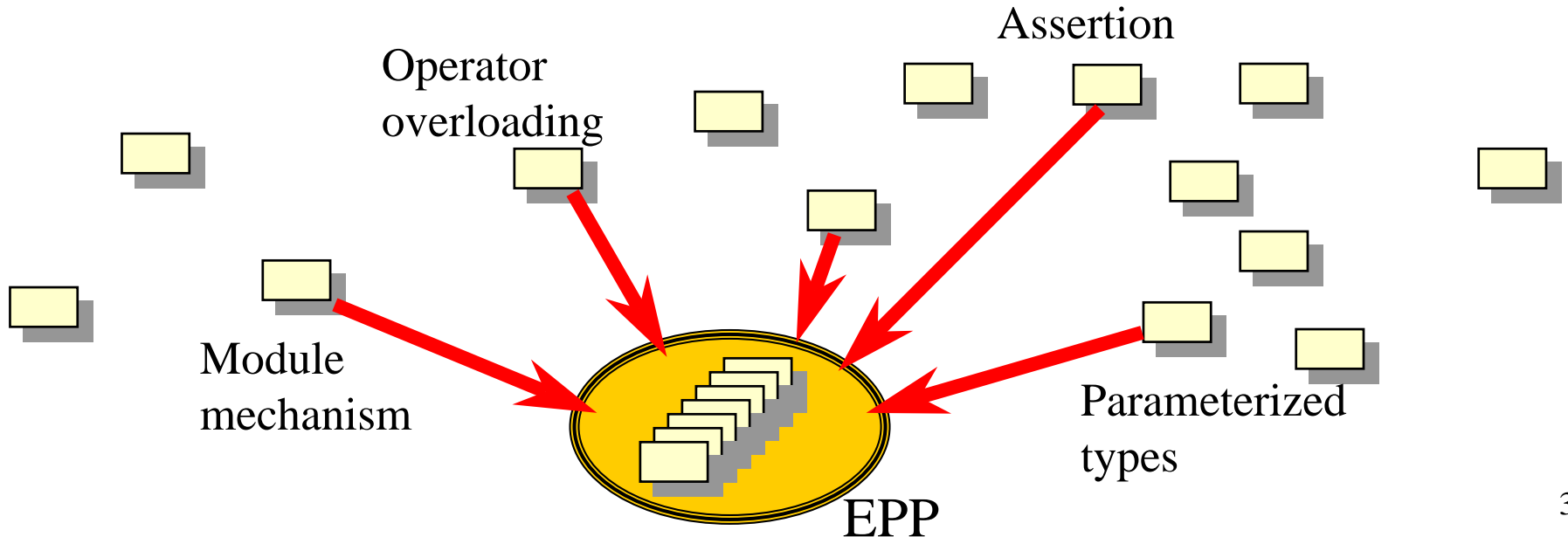
MixJuice Programming is Happy !!
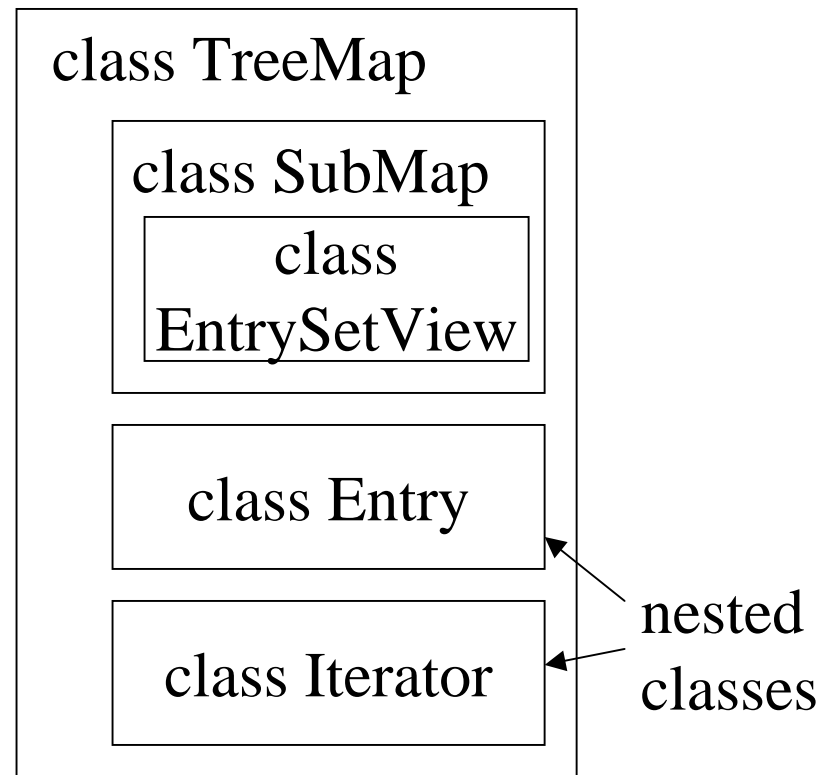
Because of
freedom of modularization and
freedom of class modeling.

Additional slides

java.util.HashMap,TreeMap

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

class TreeMap

class SubMap

class EntrySetView

class HashMap

class Entry

class Entry

class HashIterator

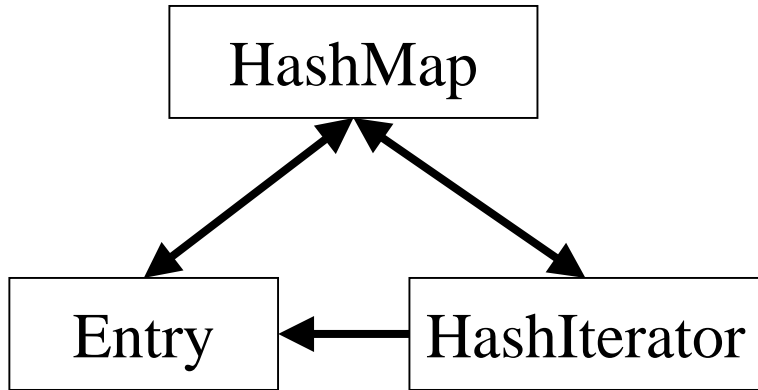class Iterator

nested classes
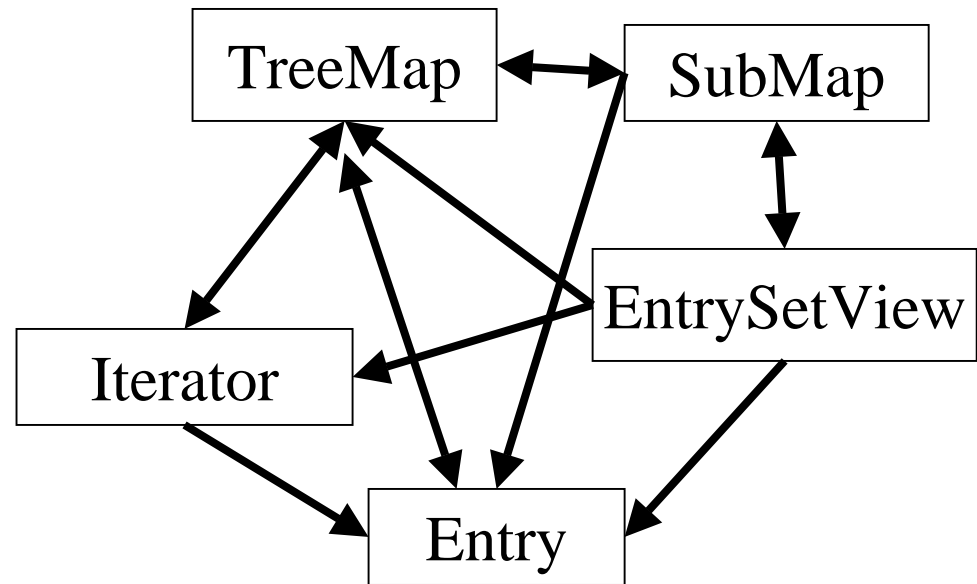
HashMap.java(500 lines)   TreeMap.java(1000 lines)

Internal of HashMap,TreeMap

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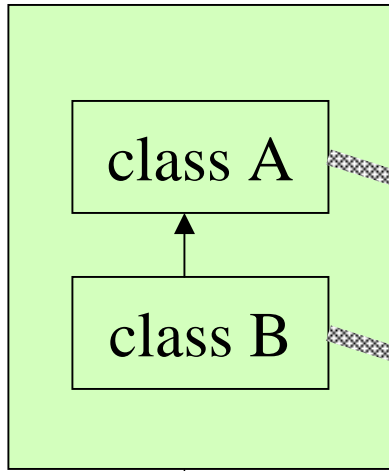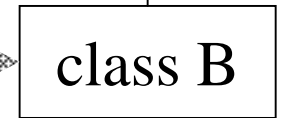- Because of the limitation of the module mechanism.

HashMap.java(500 lines)   TreeMap.java(1000 lines)

Implementation

module m1

class A

class B

module m2

delta A

delta B

class A1

class B1

class A2

class B2

class A1
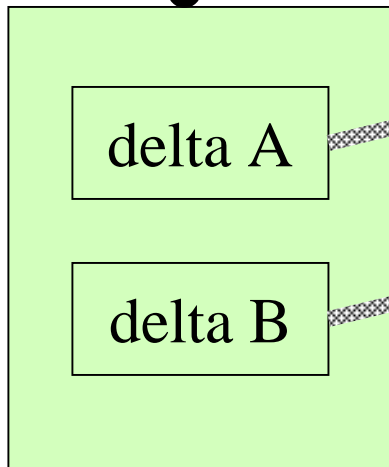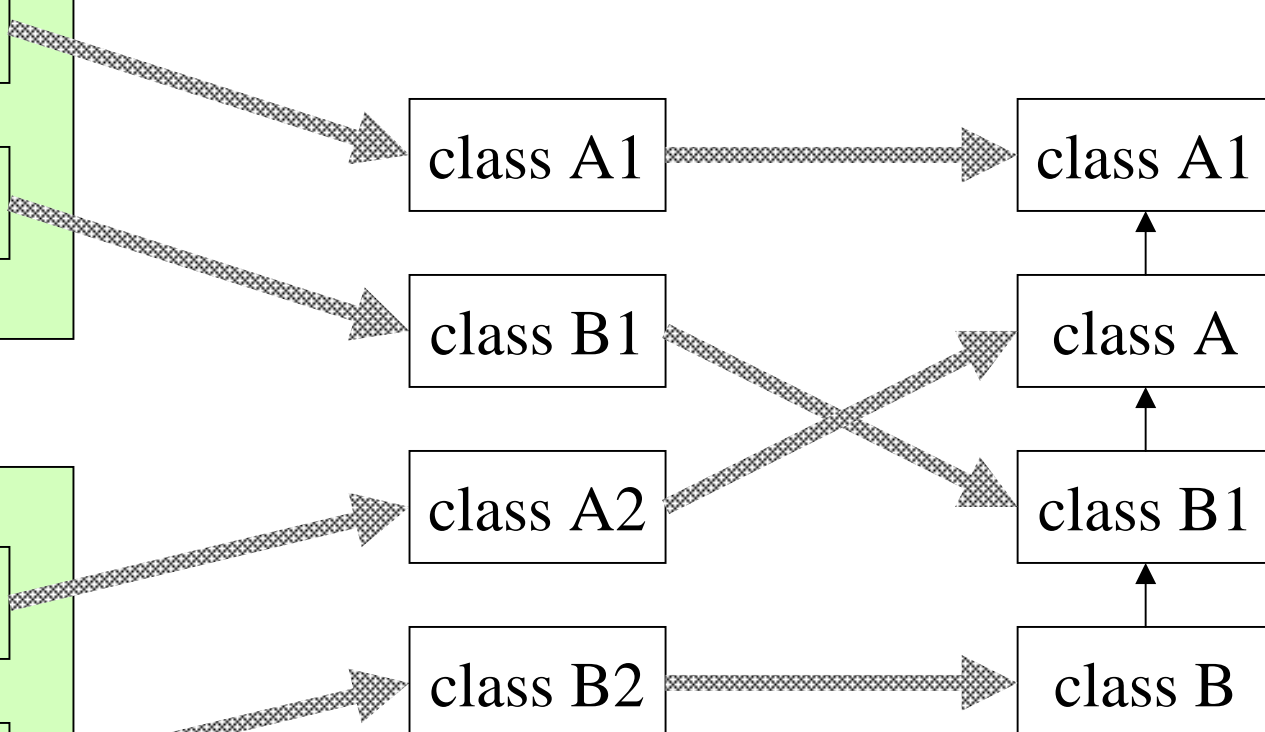
class A

class B1
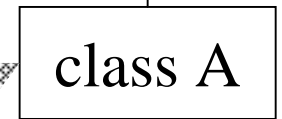
class B

Source-code
Translation
(compile time)

Byte-code
Translation
(link time)

Super classes
are changed
by the linker.[37]

safe composition

- To define composable ADT modules
  - **Design by Contract** [Meyers '88]
- To define composable sub-classes:
  - **Behavioral subtyping** [Meyers '88][Liskov '94]
- To define composable mixins:
  - **Each class mixin should be disjoint**
  - Method combinations (before/after/around, +, append, …)
- To define composable difference-based modules:
  - **Slightly stronger** rule than mixins.
  - Formal definition of the rule is on-going research.

Composability

- In order to make m2 and m3 composable, method extensions in m2 and m3
  - should preserve pre/post condition of the method.
  - should call `original` method.
  - should not access to the state of the original class.
  - may access to the extra state added by the module.

m1

C1

C2

m2

m3

C1

C2

C1

C2

Visitor pattern in MixJuice
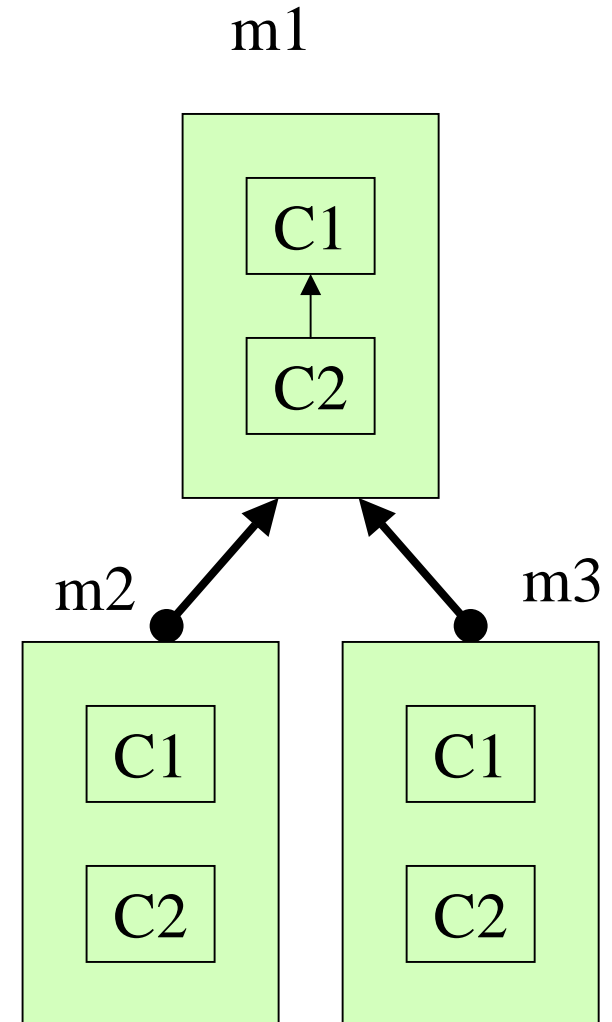
Tree structure without visitor

element

Element

ConcreteElementA

ConcreteElementB

visitor

Element

accept(Visitor)

ConcreteElementA

accept(Visitor) ○— v.visit(this);

ConcreteElementB

accept(Visitor) ○— v.visit(this);

Introduction of visitor

Visitor

visit(concreteElementA)
visit(concreteElementB)

ConcreteVisitor1

visit(concreteElementA)
visit(concreteElementB)

ConcreteVisitor1a

visit(concreteElementA)
visit(concreteElementB)

new_concrete_element

Introduction of new element

ConcreteElementC

accept(Visitor) ○— v.visit(this);

Visitor

visit(concreteElementC)

ConcreteVisitor1

visit(concreteElementC)

ConcreteVisitor1a

visit(concreteElementC)