

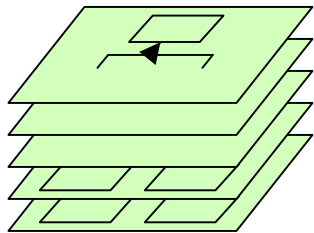
差分ベースモジュールを有する プログラミング言語:

MixJuice

2002年8月30日

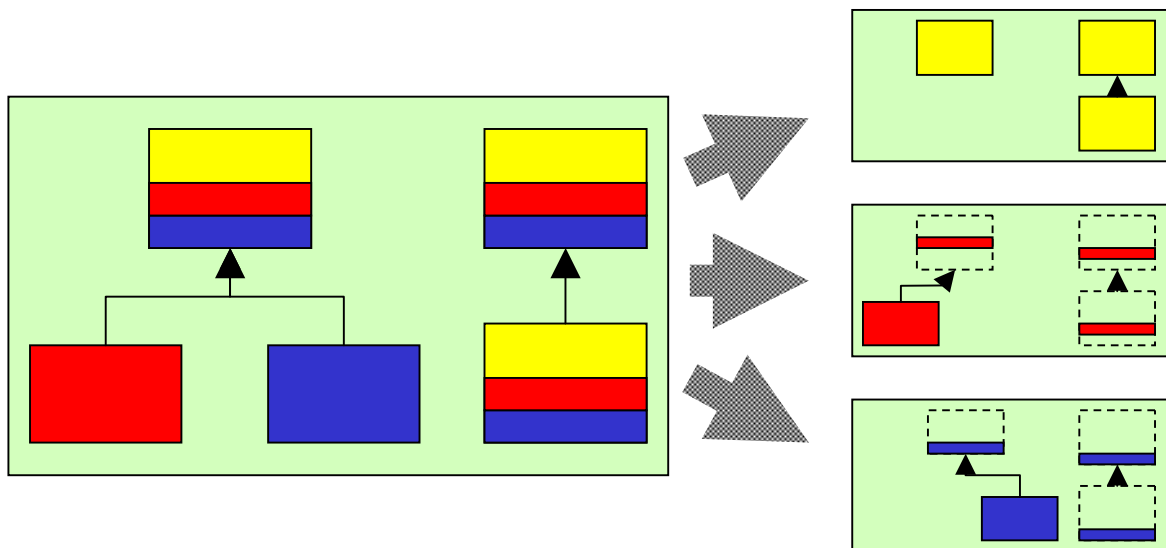
産業技術総合研究所 情報処理研究部門

一杉裕志



差分ベースモジュール

- Java のモジュール機構よりシンプル
 - “protected”、 “nested classes” はもはや不要
- **Java より高い拡張性、再利用性**
- Separation of cross-cutting concerns



発表の概要

- 現在のオブジェクト指向言語の問題点
- 差分ベースモジュール
- 拡張モジュールの衝突の問題
- サンプルプログラム: ドローツール
- MixJuice による HTTP server のモジュール化
- レイヤードクラス図
- MixJuice によるデザインパターンの改善
- モジュールの安全な結合

現在のオブジェクト指向言語の 問題点

クラスはモジュールではない

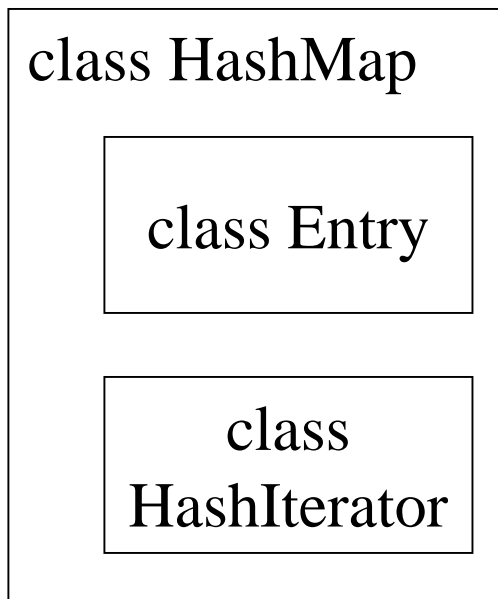
- クラスはオブジェクトの雛型
- モジュールは、再利用・情報隠蔽の単位
- クラスとモジュールを同一視すると問題
 - クラスは再利用の単位としては不適切
 - クラスは情報隠蔽の単位としては不適切

クラスは情報隠蔽の単位として は不適切

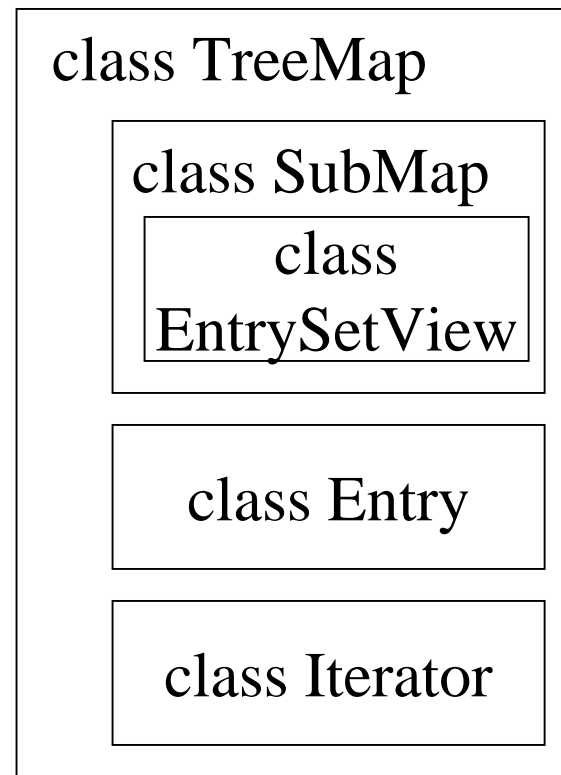
- class = module は**近似的**に成り立っているに過ぎない。 [Szyperski ECOOP92]
 - 近似精度を上げる「補正」の繰り返し：
protected, package, inner class,...
- **問題**: クラスが多機能になり巨大化すると、スコープも巨大化し、保守しにくくなる。

例: java.util.HashMap, TreeMap

- inner class を活用して内部を隠蔽



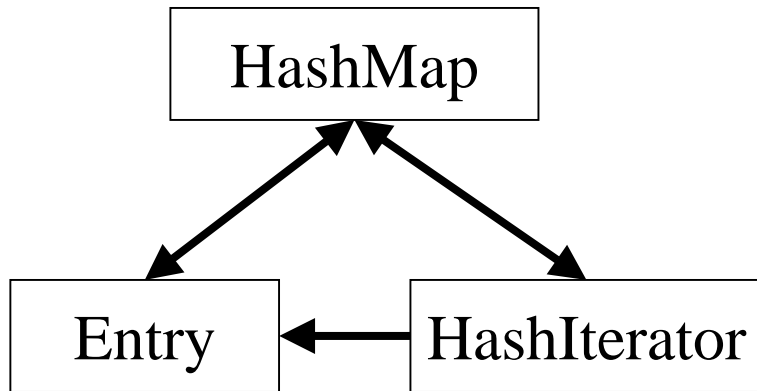
HashMap.java(約500行)



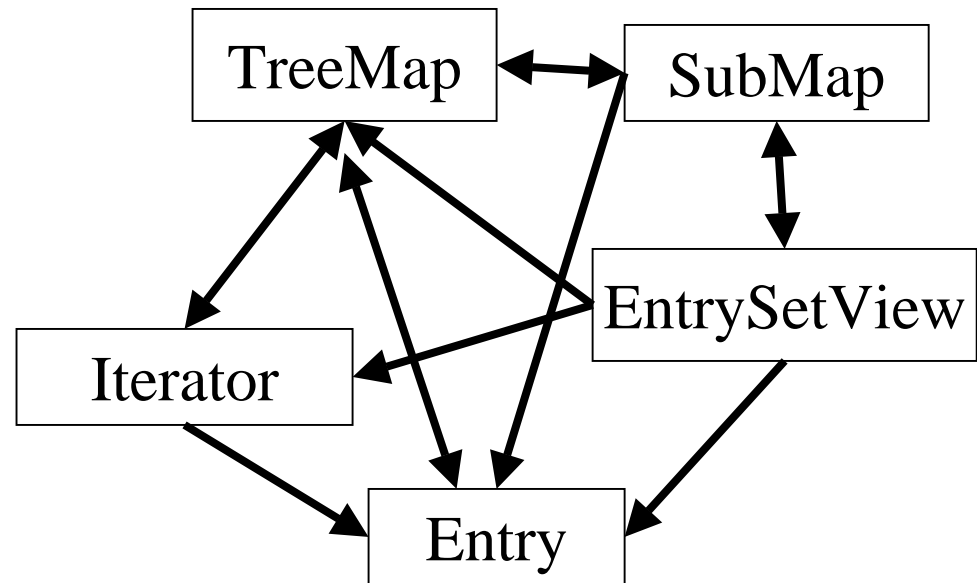
TreeMap.java(約1000行)

HashMap, TreeMapの内部

- クラスが相互依存していてモジュライティが悪い！
- これ以上整理不可能 現在のOOPLの限界



HashMap.java(約500行)



TreeMap.java(約1000行)

class は再利用の単位としては 不適切

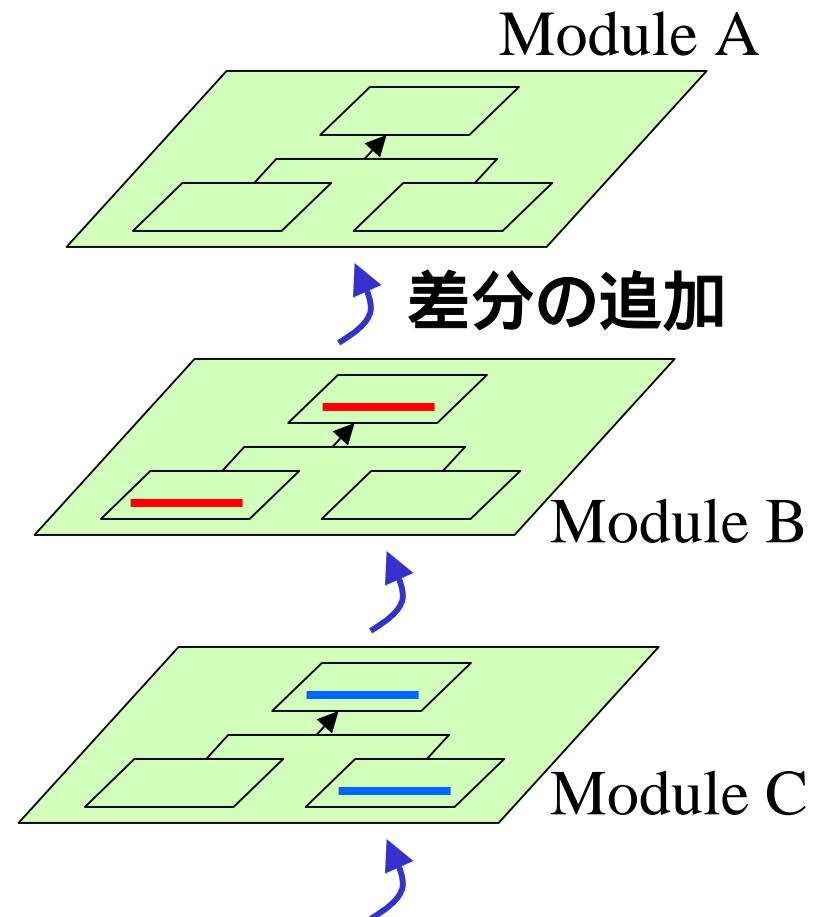
- Separation of crosscutting concerns (クラスをまたがるコードの分離) ができない [Kiczales 97]
- **さまざまな提案**
 - AspectJ [Kiczales 99]
 - Hyper/J [Ossher ICSE 99]
 - Mixin layers [Smaragdakis ECOOP98]
 - BCA [Keller ECOOP98]
 - adaptive p-n-p [Mezini OOSPLA98]
 - collabolation-based design [VanHilst OOPSLA96]
 - ...
 - Subject-oriented programming [Ossher OOPSLA92] [Ossher OOPSLA93]
- **それぞれ、なんらかの欠点を持つ**
 - 情報隠蔽がない、分割コンパイルできない、型チェックできない、言語仕様が複雑、...

差分ベースモジュール

クラス モジュール

差分 = モジュール

- MixJuice のモジュールは、
 - オリジナルのプログラムに対する差分
 - 情報隠蔽の単位
 - 再利用の単位
 - 分割コンパイルの単位
- 型チェック、分割コンパイルできる “patch ファイル” のようなもの。

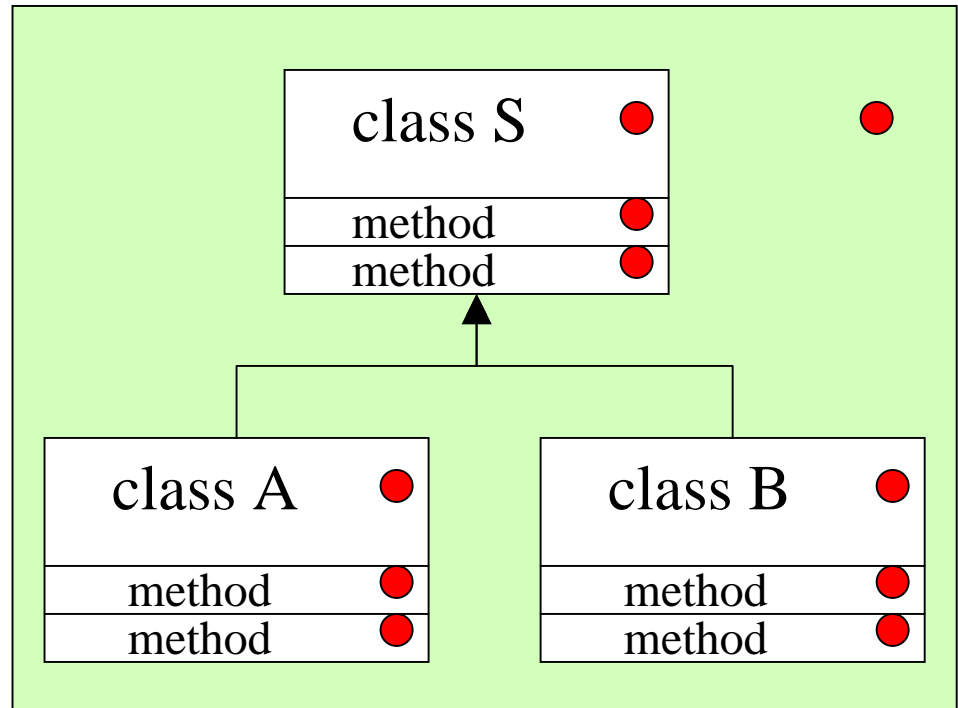


差分の追加方法

- 以下のことが可能
 - 新たなクラスを追加
 - オリジナルのクラスにフィールド・メソッドを追加
 - オリジナルのメソッドをoverrideして拡張
- すべてのメソッドがhook

高い拡張性

オリジナルのシステム



- 拡張可能な場所 (= hook)

モジュール定義

```
module m2
  extends m1 // このモジュールの super module
  {
    // モジュール本体 (追加する差分)
    define class A {...} // 新たなクラスの追加
    class B {...} // 既存のクラスへの差分の追加
  }
```

field, methodの書き方はJavaと同じ

define のあるなしで、新規定義と差分とを区別

プログラム例

```
module m1 {  
  define class S {  
    define int foo(){ return 1; }  
  } // 1  
  define class A extends S {  
    int foo(){ return original() + 10; }  
  } // 11  
}
```

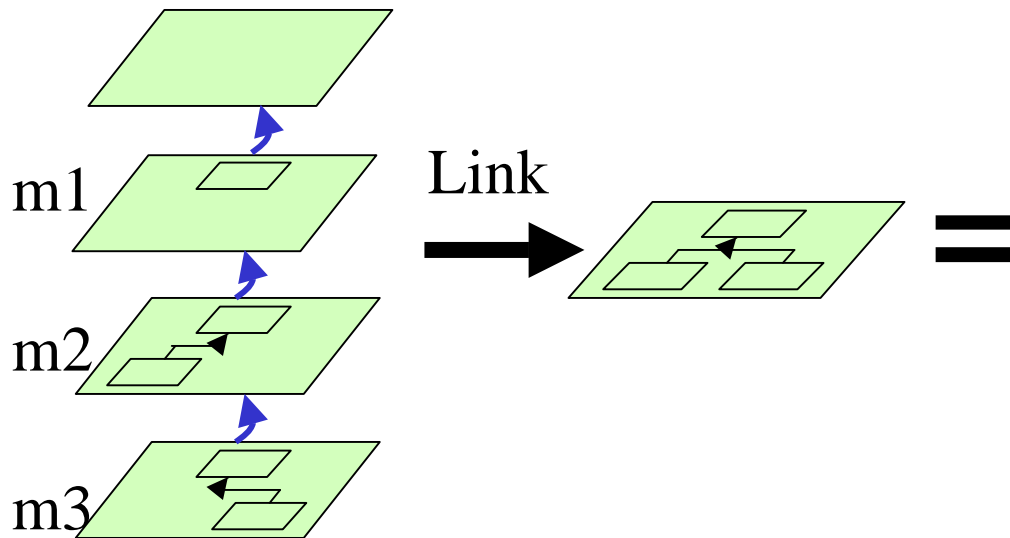
m2 と m3 は
別々に
コンパイル可能

```
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  
}
```

モジュールのリンク

エンドユーザが、
使いたいモジュールを選択



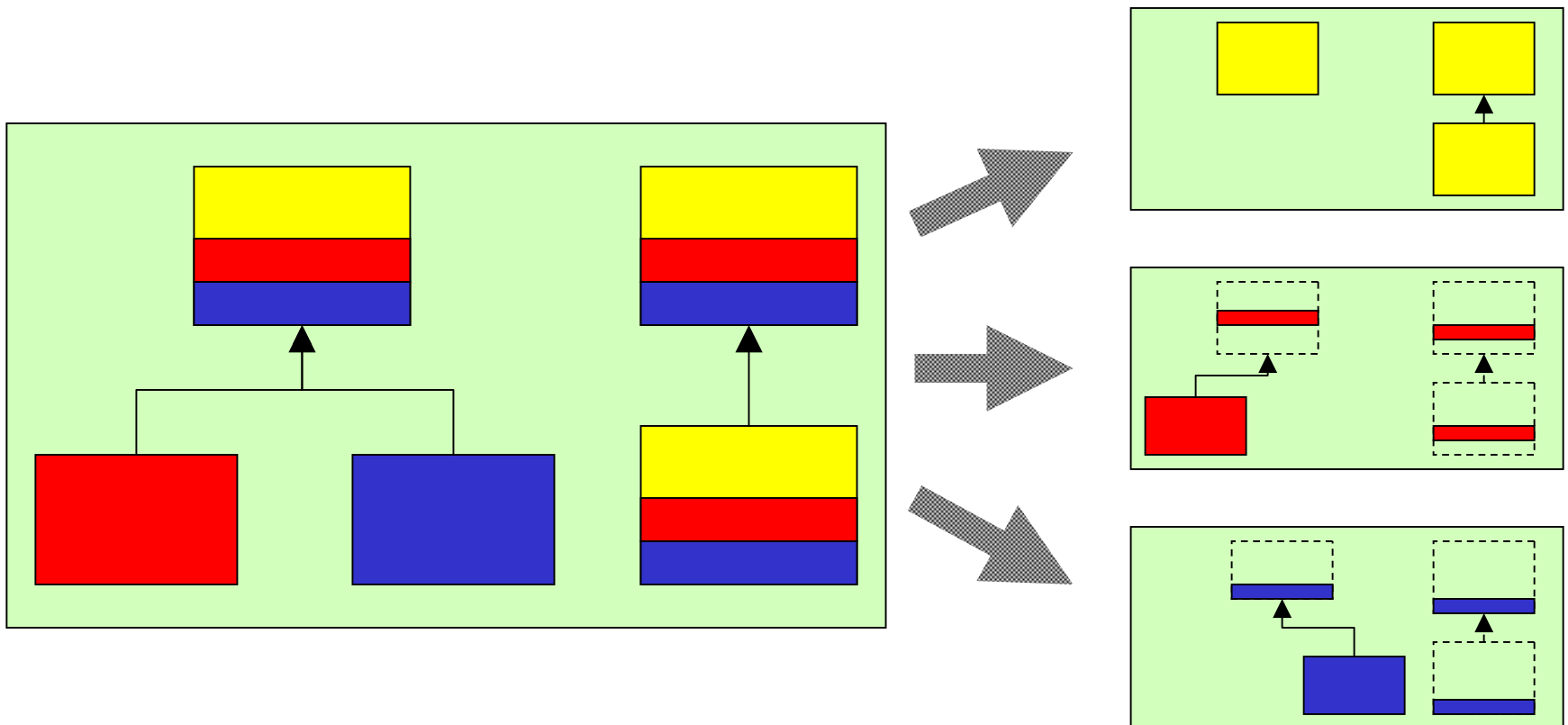
リンクは、モジュールを
1列に並べて、
上から順に差分を
追加していく

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

リンク結果は、従来のオブジェクト
指向言語(Java)と同じ

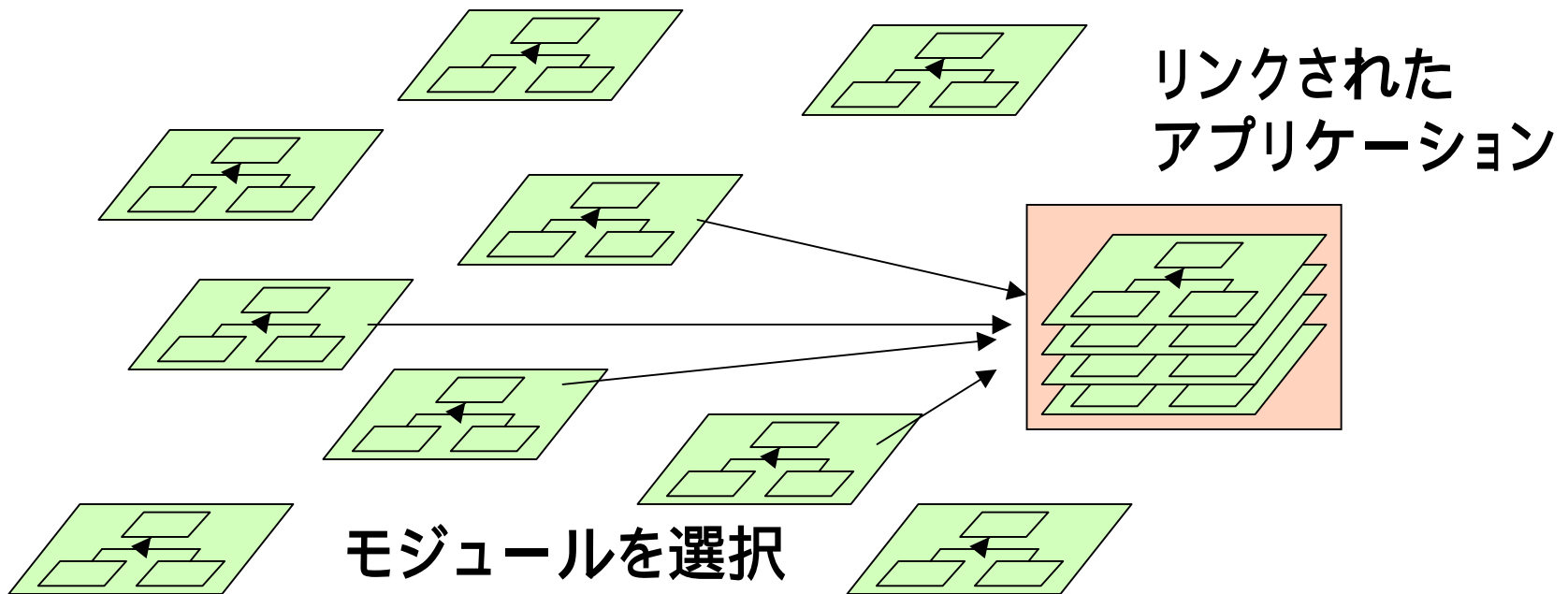
Separation of crosscutting concerns

- 複数のクラスにまたがったコードを分離して記述できる。
- Cf. Aspect oriented programming [Kiczales 97]



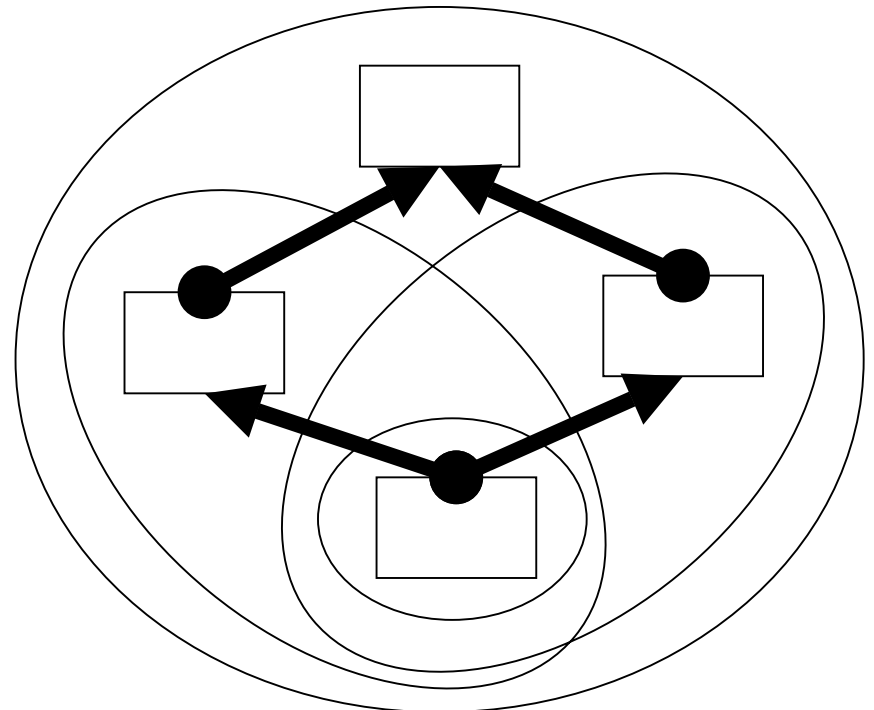
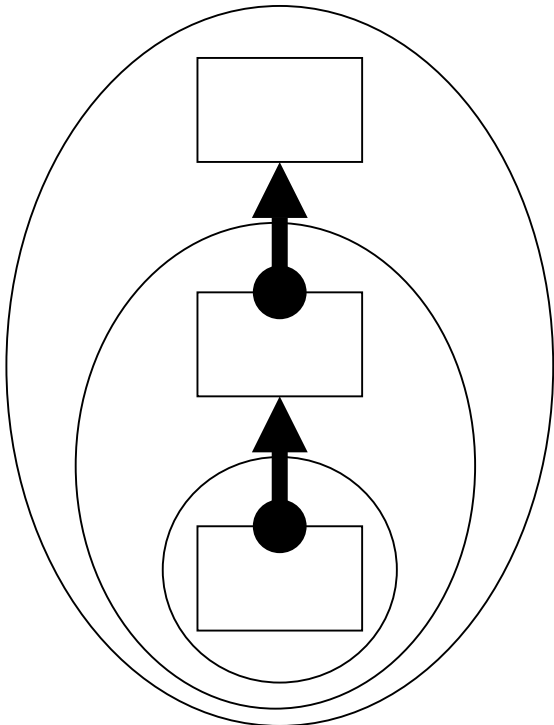
“glue code” が不要

- エンドユーザは、実装の詳細をしらなくても、モジュールを組み合わせて独自のアプリケーションを構築できる。



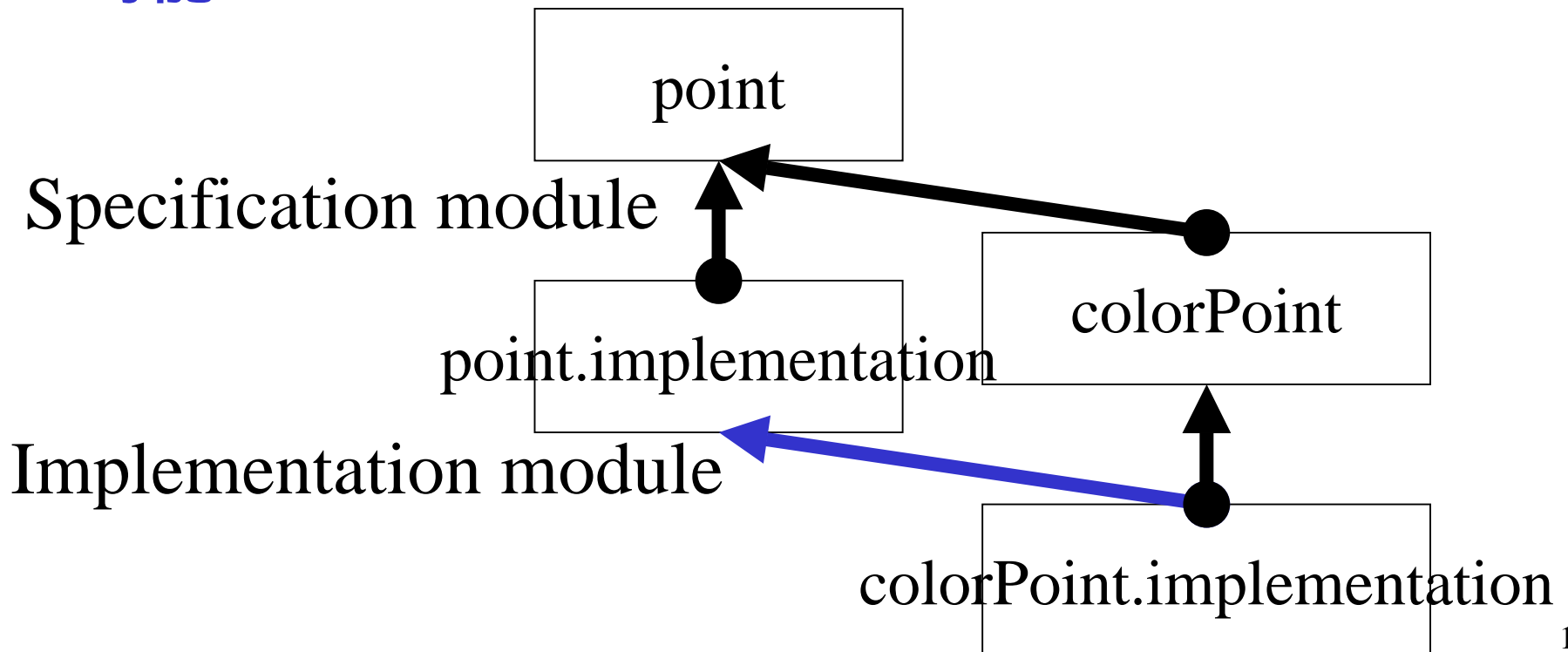
“nested classes” が不要

- すべての名前がサブモジュールから見える
- nested classes よりも柔軟な名前空間制御が可能



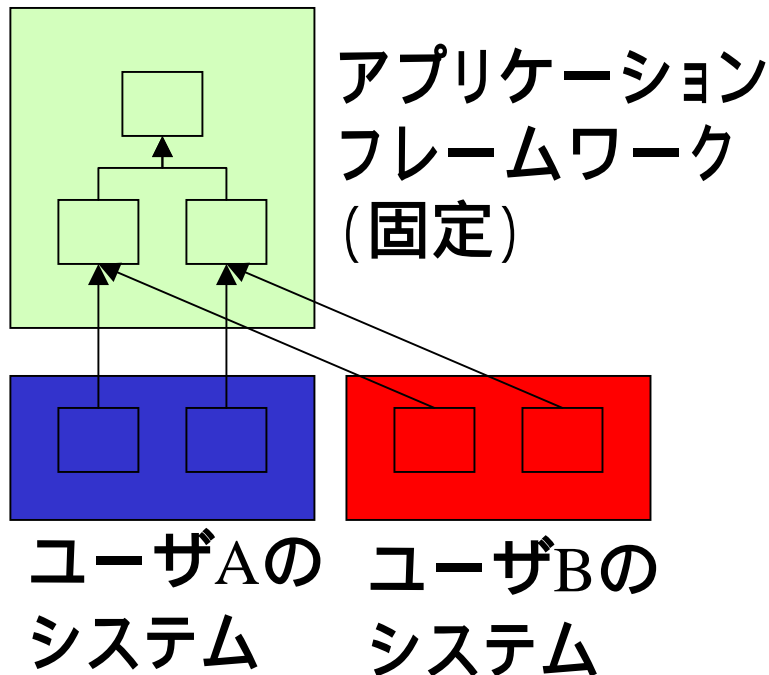
“protected” が不要

- black-box reuse と white-box reuse の両方が、モジュールの継承で表現可能
- subclass が super class を black-box reuse することも可能



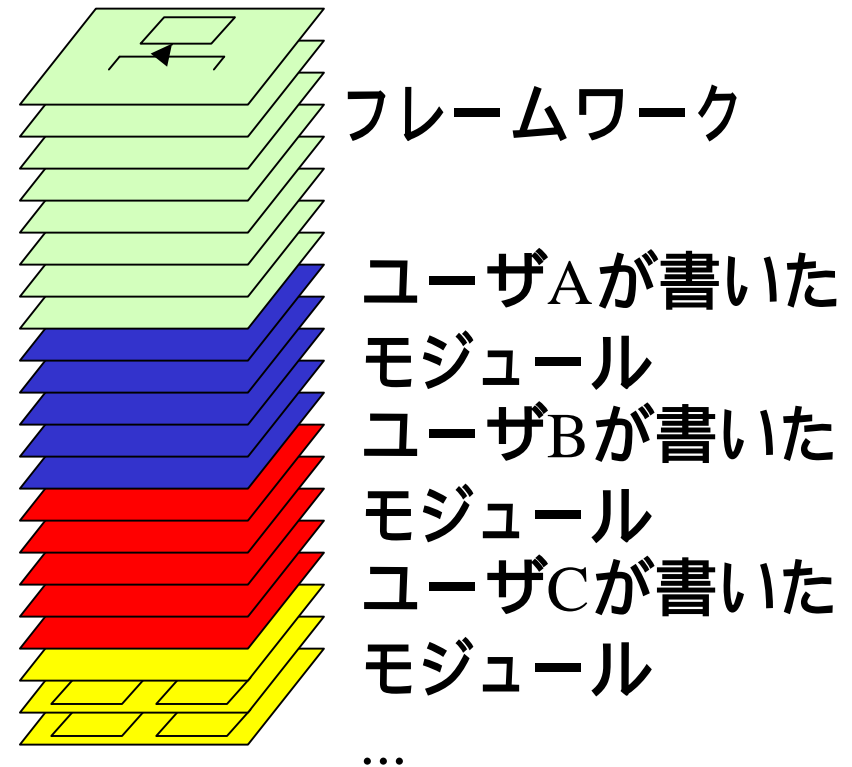
再利用性の向上

従来:
固定した「フレームワーク」へのサブクラスの追加しかできない。



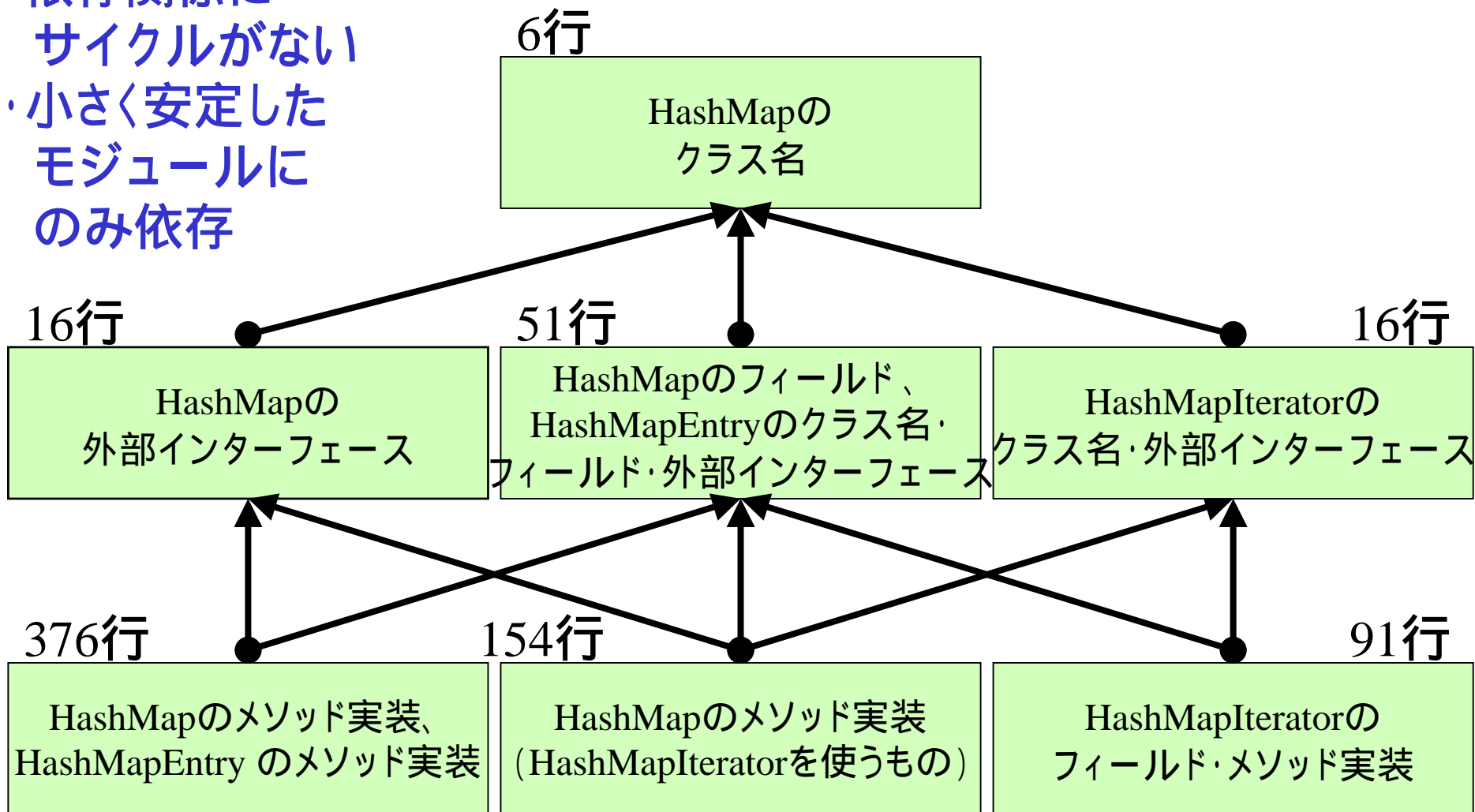
MixJuice:

Mixinのように、第三者の書いたコードを組み合わせさせて使える。



HashMapをMixJuiceで記述

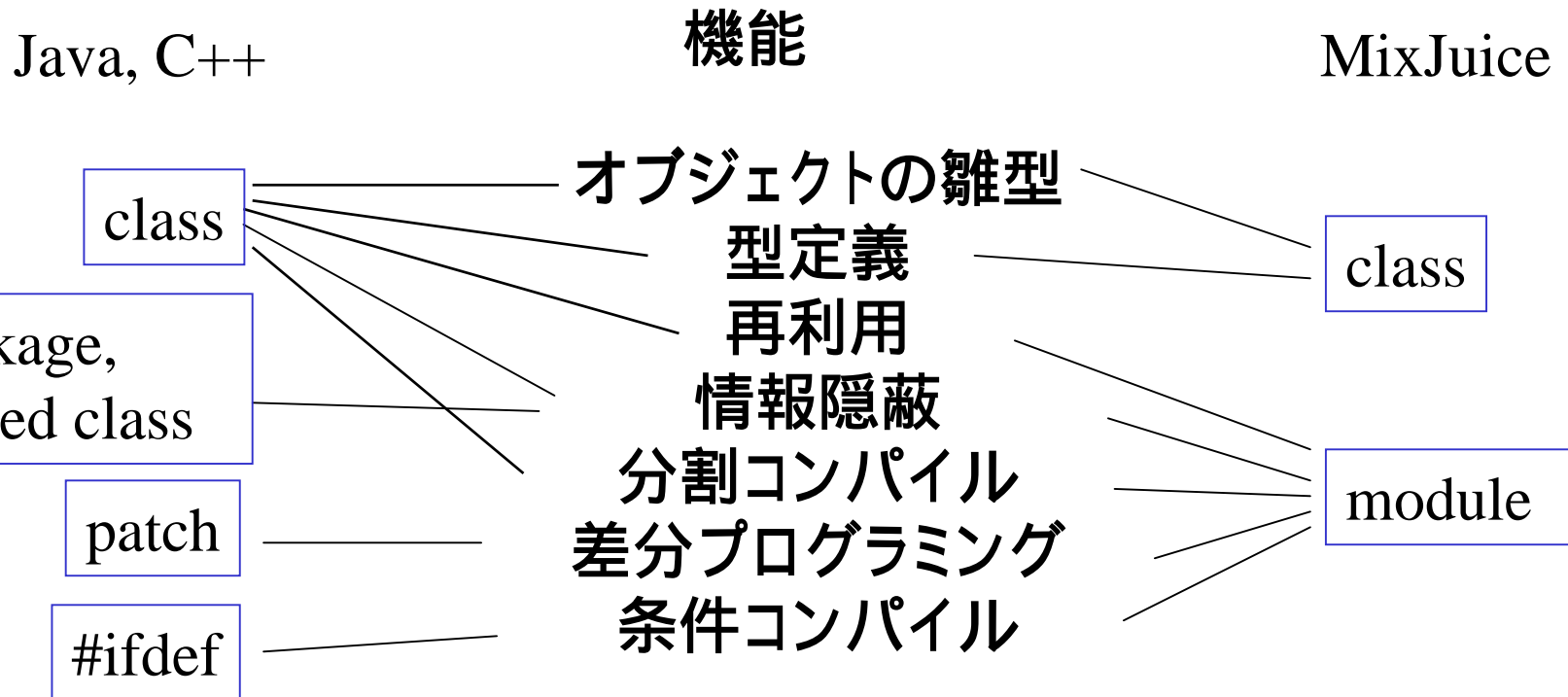
- ・依存関係に
サイクルがない
- ・小さく安定した
モジュールに
のみ依存



利点のまとめ

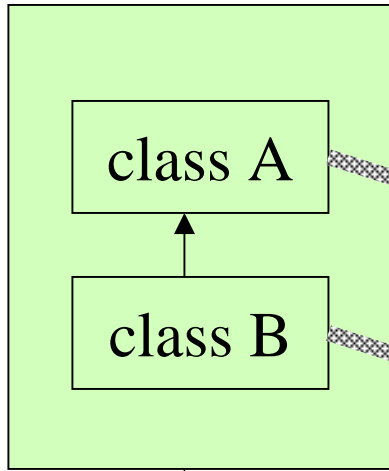
- **高い拡張性**
 - すべてのクラス、メソッドが、拡張可能な hook
- **高い再利用性**
 - 複数のモジュールを組み合わせられる
- **柔軟なモジュール分割が可能**
 - クラスとは完全に直交したモジュール機構
 - separation of crosscutting concerns **が可能**
- **柔軟な名前空間制御が可能**
 - 名前空間の多重継承
 - 重なりのある名前空間も表現可能

クラスとモジュールの役割分担

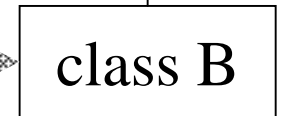
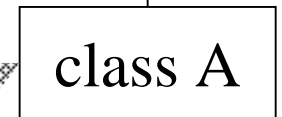
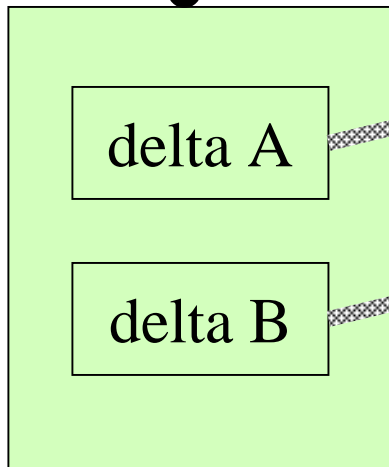


「差分の追加」の実現方法

module m1



module m2

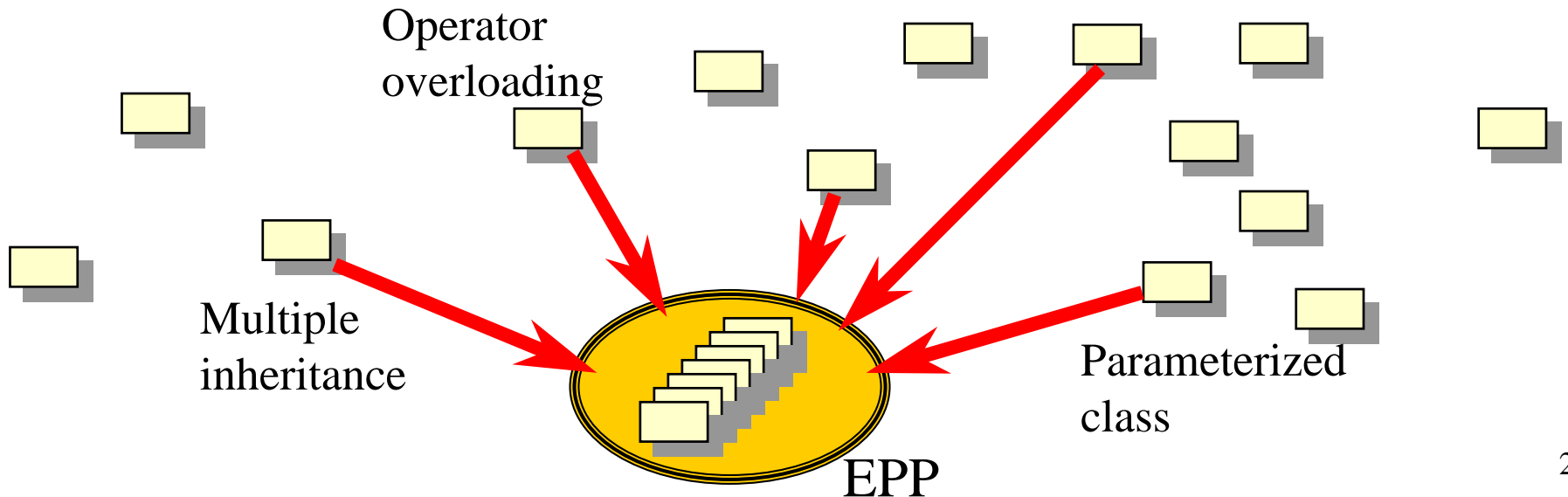


ソースコード変換
(コンパイル時)

バイトコード変換
(リンク時)

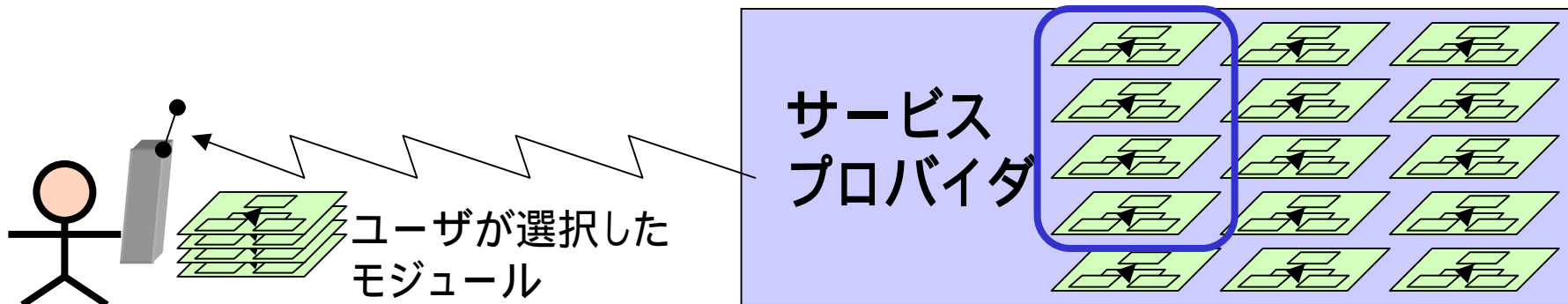
適した用途1: 拡張性の高いソフトウェア

- 例: 拡張可能プリプロセッサ EPP 1.1
 - 「差分 = モジュール」のスタイルで記述された実用的ソフトウェア
 - EPP 2.0 は MixJuice で実装中



適した用途(?) 2: PDA上のアプリケーション

- 多機能化するPDA
 - 携帯電話、PIM、メール、リモコン、ゲーム、電子財布、...
- 矛盾する要求
 - 各個人が必要とする機能は千差万別
 - メモリが少なく、多機能アプリケーションは乗らない
- MixJuiceを使えば、
 - 無限のバリエーションを提供でき、
 - ダウンロードサイズは必要最小限にできる

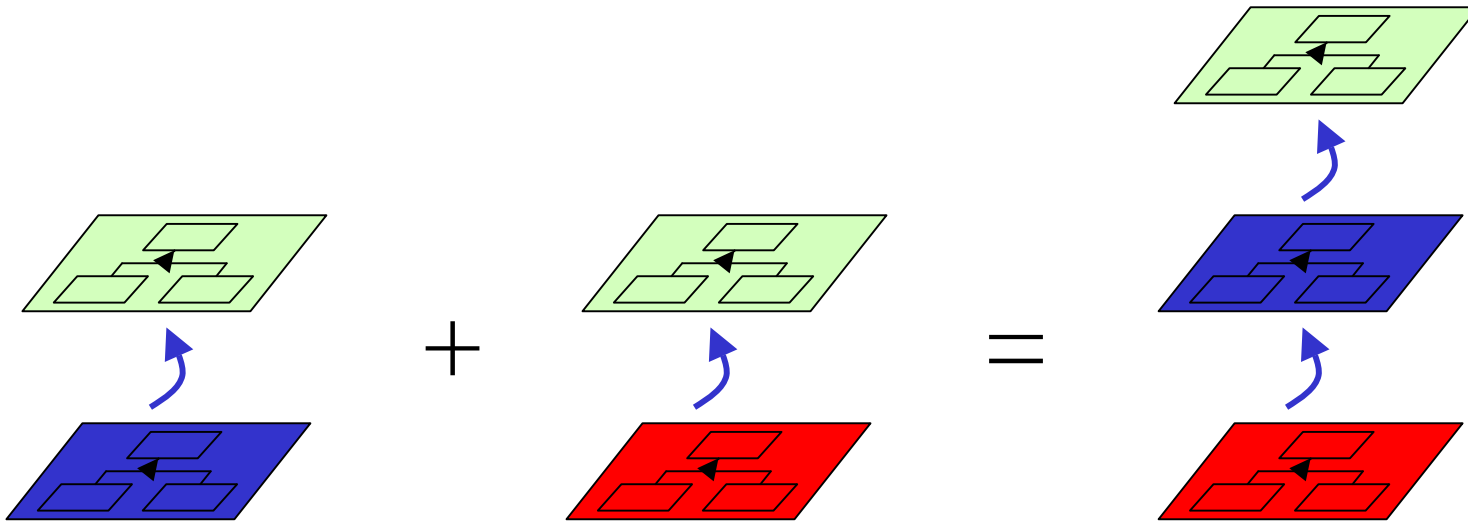


適した用途(?) 3 : 分散環境に おけるソフトウェアの開発

- オープンソースによるソフトウェアの共同開発
 - linux, GNU
- 複数のバージョンの統合の問題
 - 低レベルのツール(diff/patch, cvs)
 - カリスマ的主導者の必要性
- MixJuiceは、言語レベルで、より安全に差分の統合を支援

拡張モジュールどうしの
衝突の問題と、
その解決案

衝突の問題



正しいプログラム

正しいプログラム

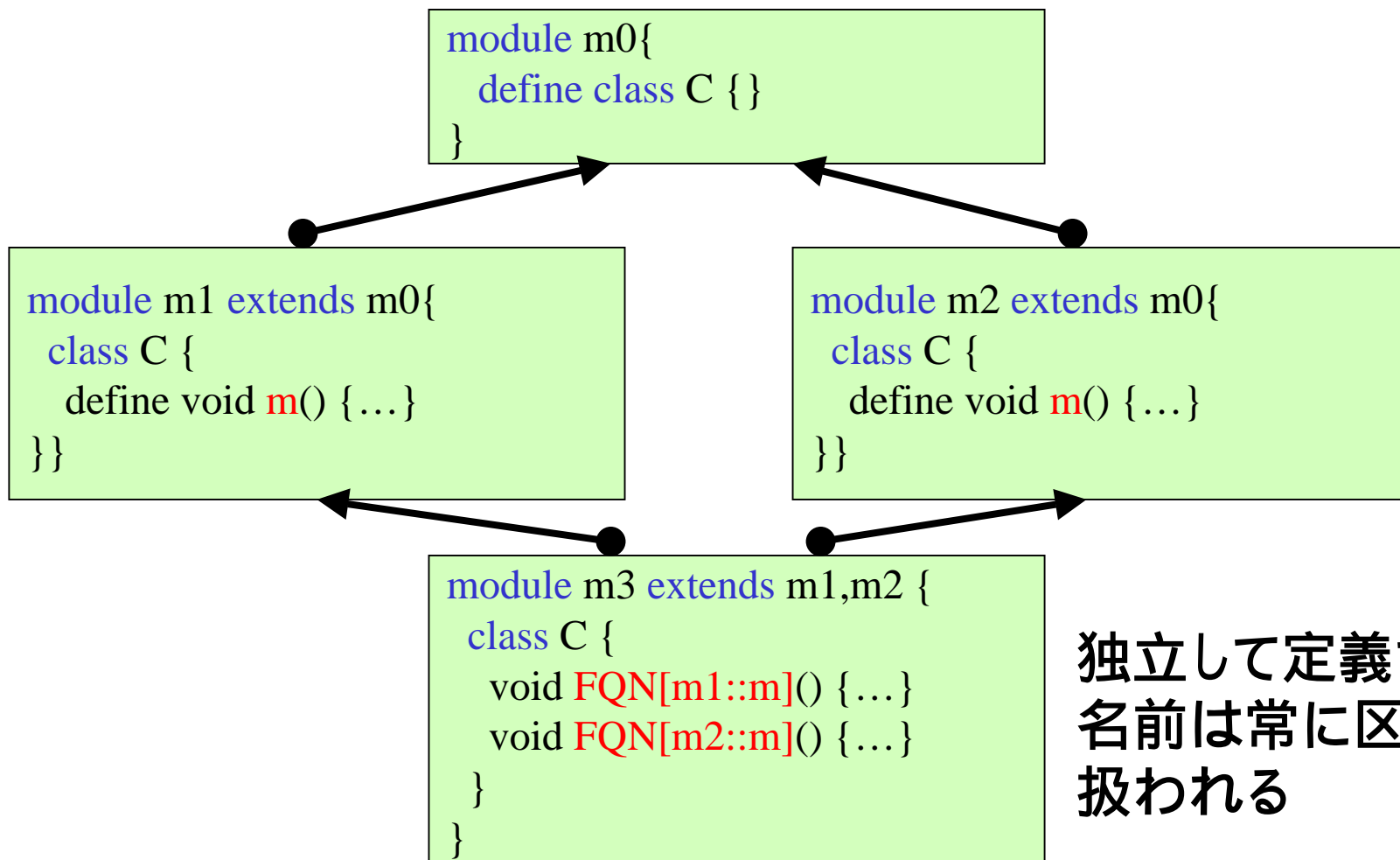
正しく動くとは限らない！

衝突の原因

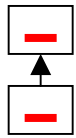
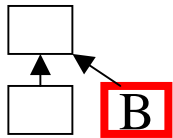
- 3種類に分類できる:
 - 名前の衝突(Name collision)
 - 実装欠損(Implementation defect)
 - 意味的な衝突(Semantic collision)
- それぞれに対して解決案を考えた。

名前の衝突の問題

- すべての名前に「完全限定名」を導入して解決。



実装欠損の問題



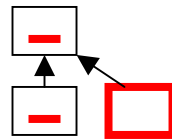
```
module m0{  
  define class S {  
  define class A extends S {  
}
```

サブクラスの追加

abstract method の追加

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

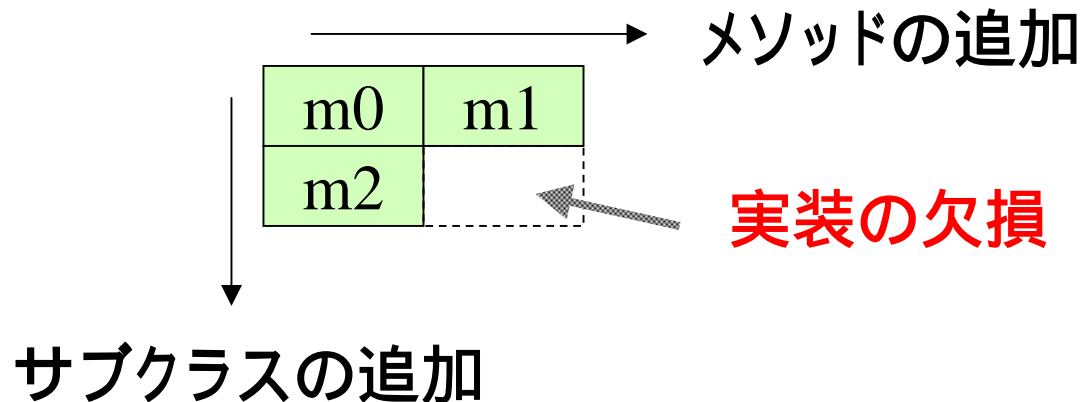
```
module m2 extends m0{  
  class S { define abstract void m();  
  class A { void m() {...}}  
}
```



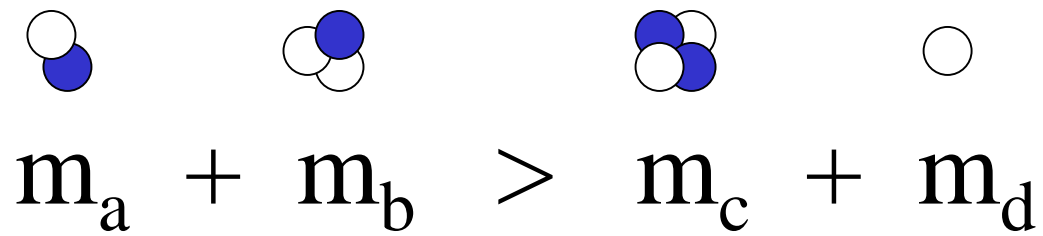
両方のモジュールを同時に使おうとすると、リンク時エラーになる！
(だれもクラスBのメソッドmを実装していないから。)

実装の欠損

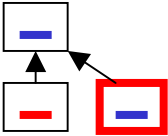
- 異なる2つ以上の「拡張の方向」があるときに起きる。
- m1とm2の両方の知識をもつだけ**が欠損を**補完**しないと、システムは動かない。



質量欠損



実装欠損に対する従来の対策

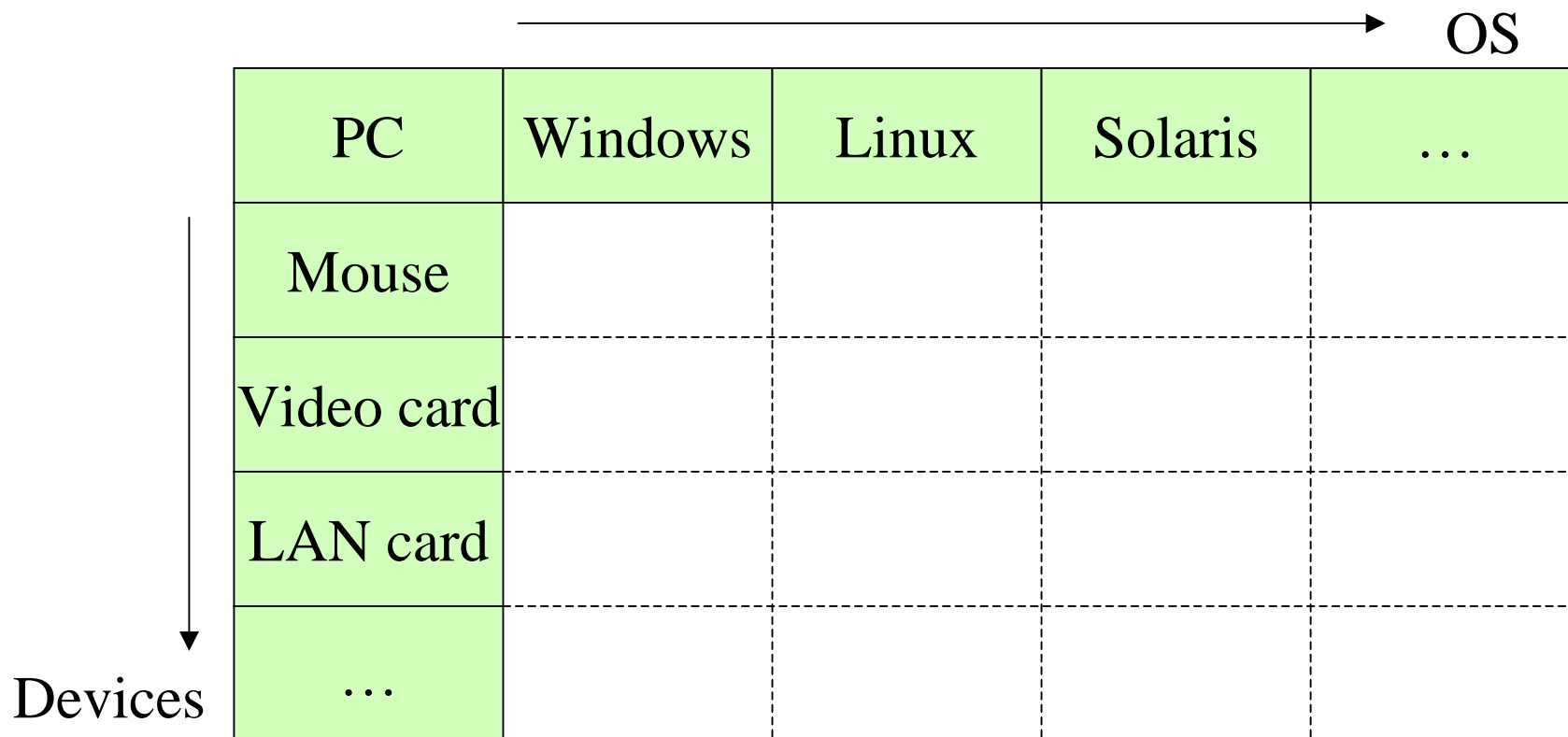
- 拡張方法に制限を加え、1方向にしか拡張できないように強制する。
 - 実装欠損は起きない。
 - しかし、拡張性に限界。
- 「デフォルトの挙動」の定義を強制する。

[Millstein, Chambers ECOOP99]

 - Super class への abstract method の追加を禁止。
 - リンク時型チェックは必ず通るようになる。
 - しかし、一般には「正しいデフォルトの挙動」は定義不可能。

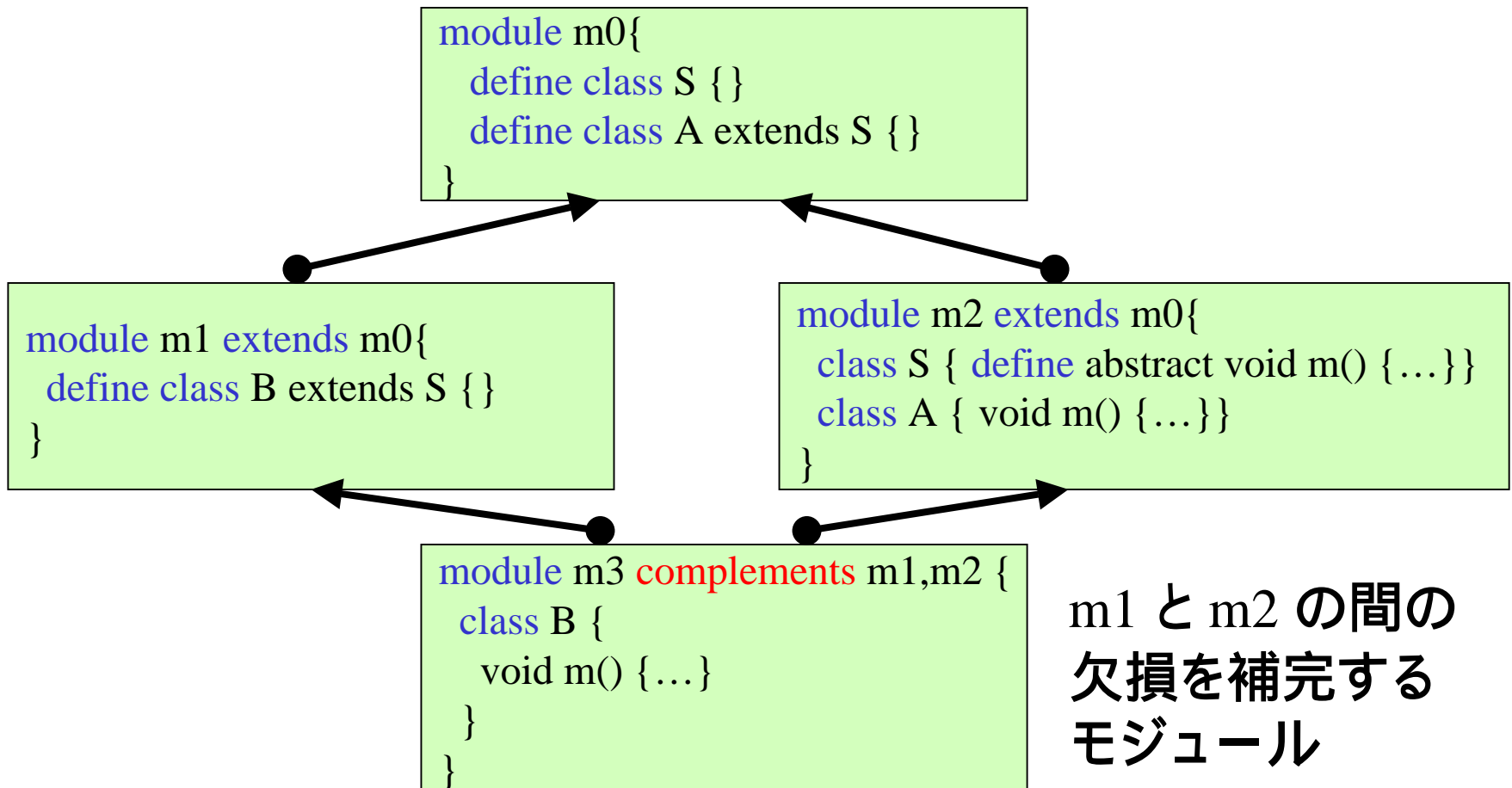
実世界における、実装欠損の例

- だれかがOSとデバイスとの間の補完を行わなければならない。(= デバイスドライバの実装)



補完モジュールの言語サポート

- リンカーが、m3 (誰かが実装) を CLASSPATH から検索して、自動的にリンクする。



意味的な衝突の問題

- 仮定: 契約による設計(design by contract)
[Meyers '92]
 - 部品実装者は仕様を満たすように部品を実装
 - 部品利用者は仕様のみを仮定して部品を利用
- 安全に結合可能なモジュールを定義するための拡張ルール(後述)
- 差分ベースモジュール用に拡張した表明検査機構(未実装)

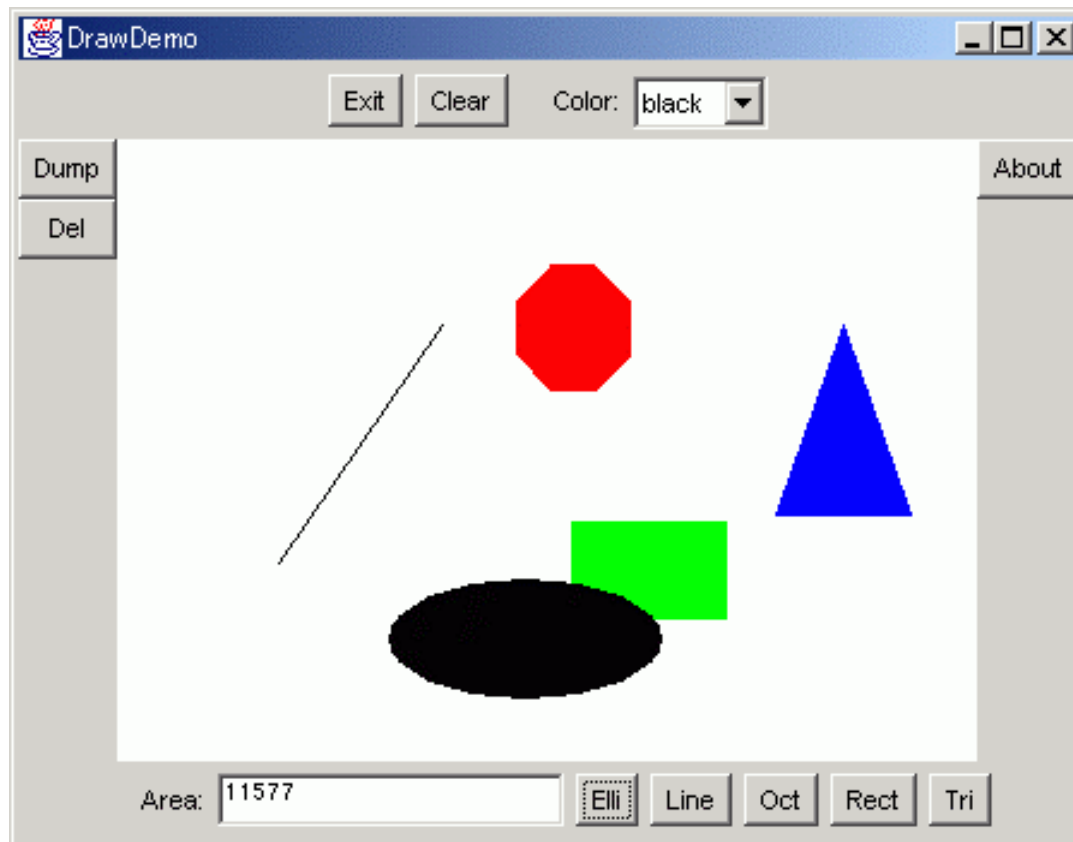
衝突の問題の解決案のまとめ

- **名前の衝突**
 - 完全限定名の導入により解決
- **実装の欠損**
 - リンク時に検出
 - 補完モジュールの言語サポート
- **意味的な衝突**
 - 拡張ルール(後述)
 - 表明検査により実行時に検出(未実装)

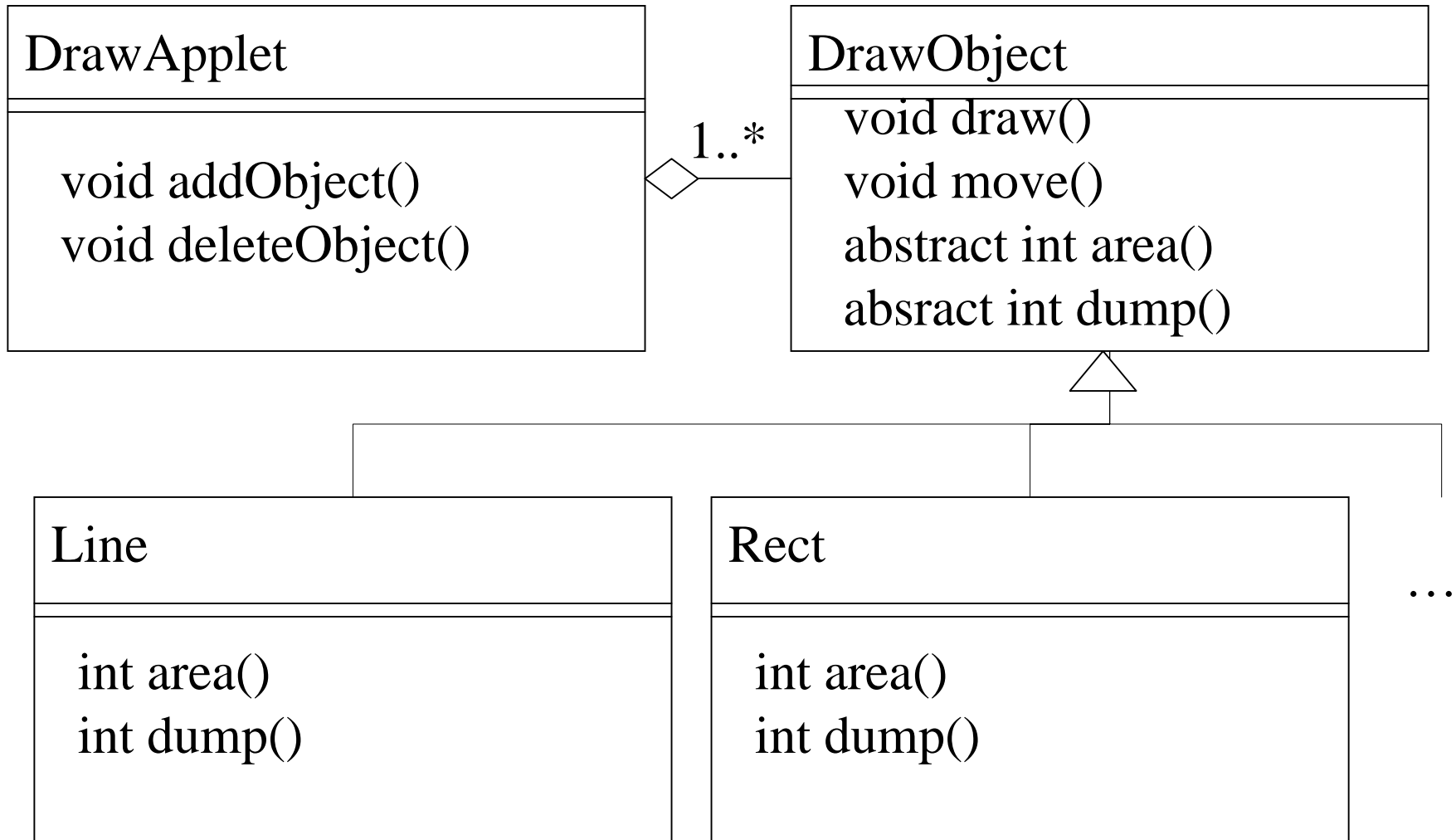
サンプルプログラム:
ドローツール

サンプルプログラム： ドローツール

- 「初心者プログラマー」が1ヶ月弱で作成



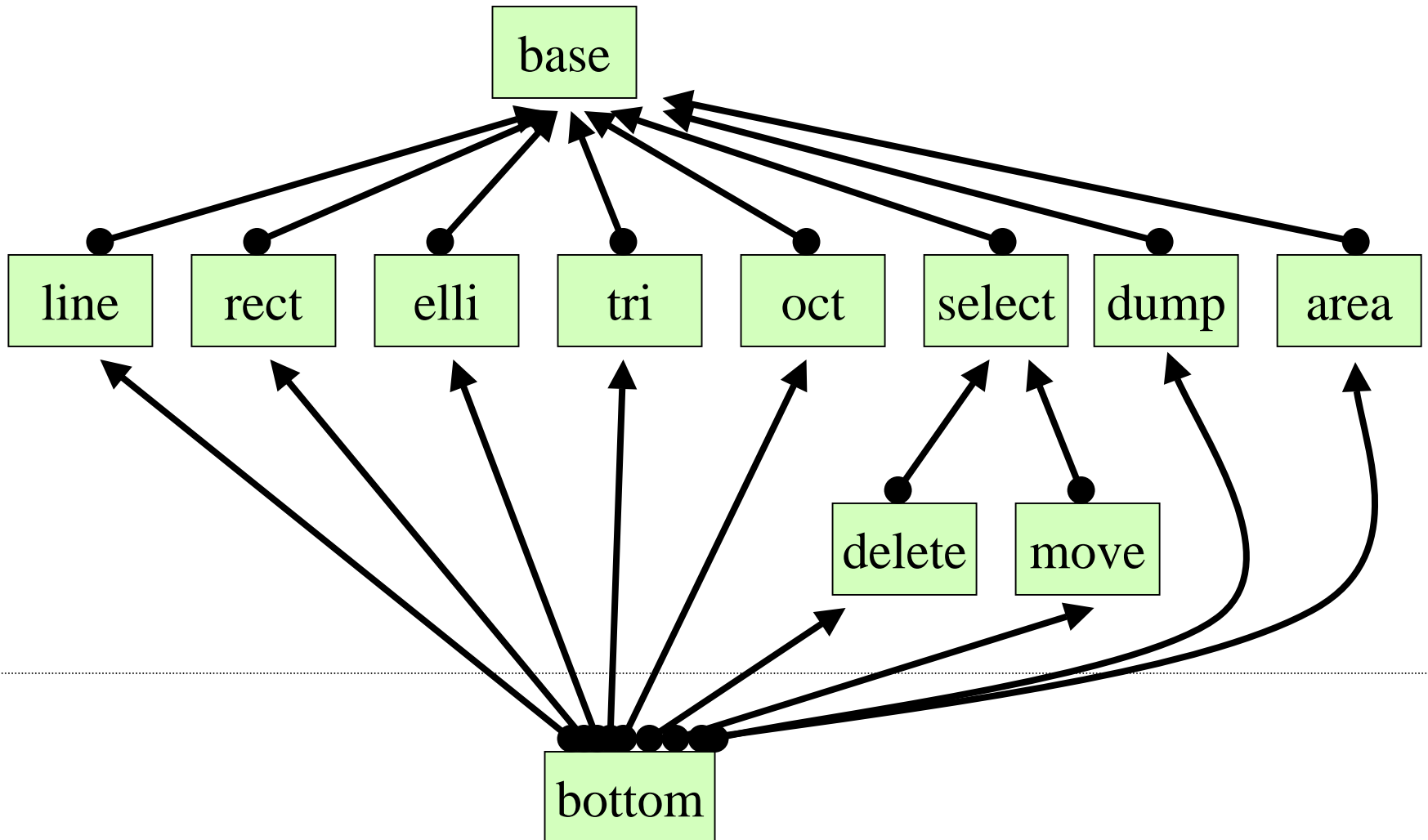
ドロートツールのクラス階層



各モジュールの機能

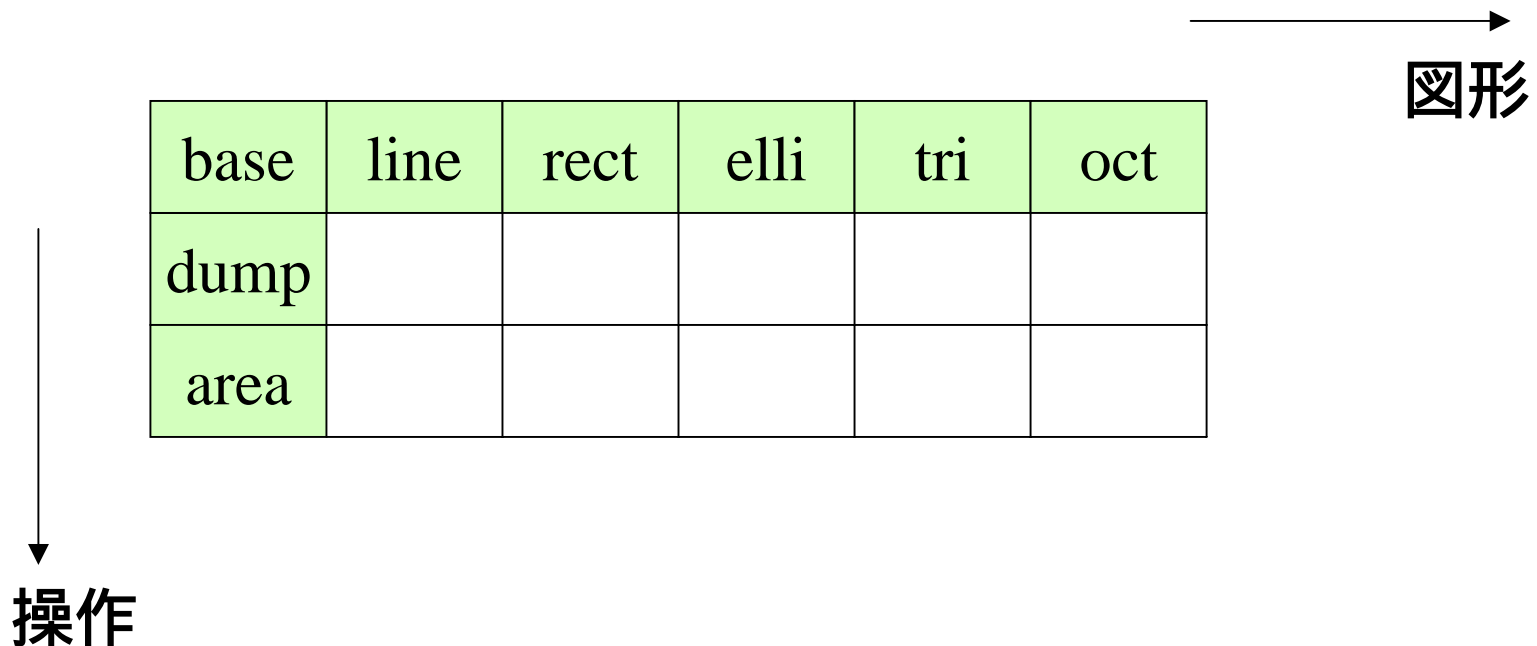
- base : フレームワーク
- 機能
 - select : 図形の選択
 - delete : 選択した図形の削除
 - move : 選択した図形の移動
 - dump : 画面情報をテキスト形式で出力
 - area : 全図形の面積(単位はピクセル)の合計を表示
- 図形
 - line : 直線
 - rect : 長方形
 - elli : 楕円
 - tri : 三角形
 - oct : 八角形

モジュール間の依存関係



ドローツールの補完モジュール

- { dump, area } × { line, rect, elli, tri, oct } の
10個が必要



MixJuiceによる
HTTP serverのモジュール化

Jasper HTTP サーバ

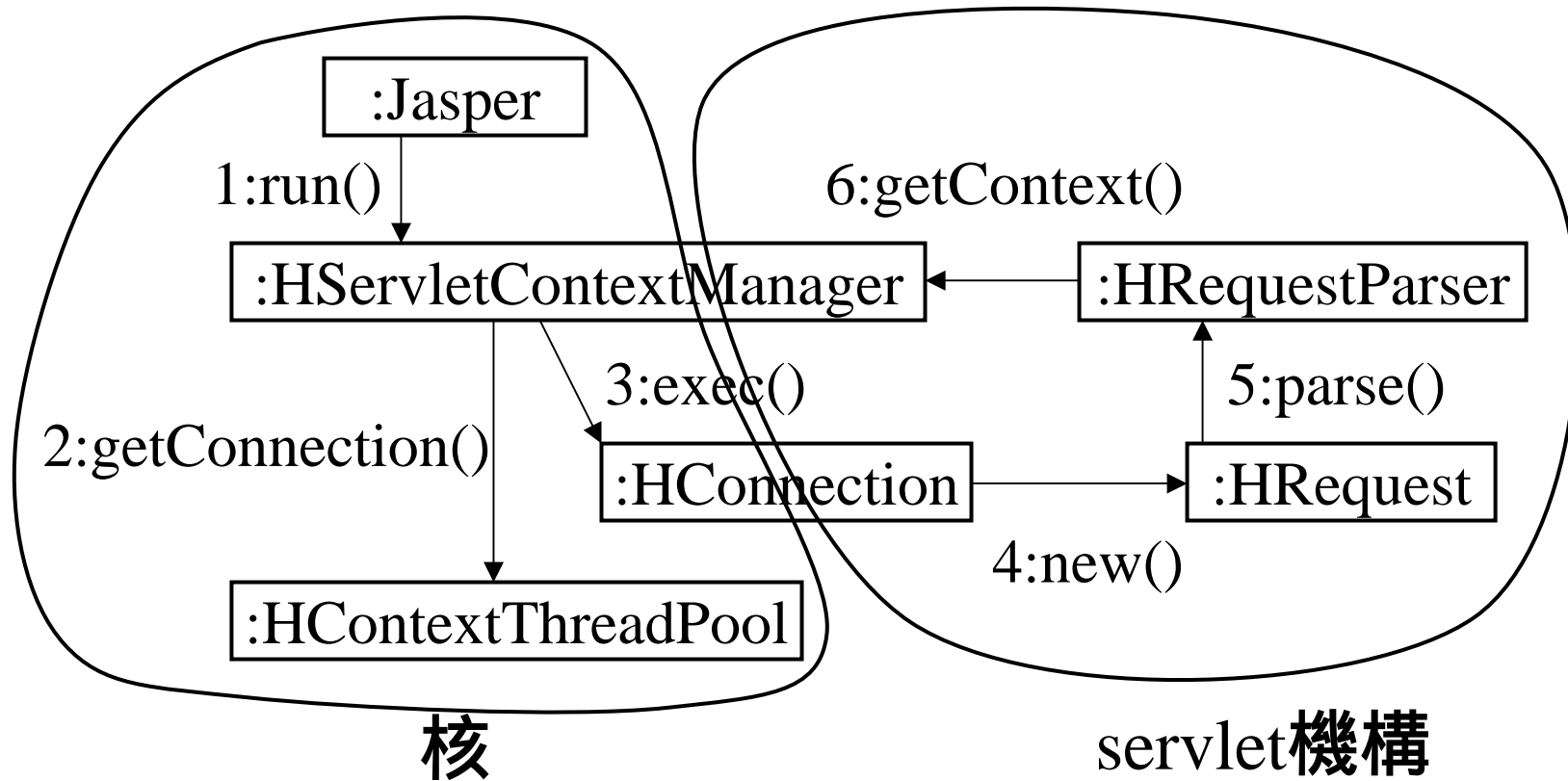
- OpenJEが提供しているフリーソフトウェアのHTTPサーバ
- 100% pure Java
- 拡張性が高い (servletベース、dynamic loadingによる拡張機構)
- コンパクト (約10,000行・約230KB・約50KBのservlet libraryが必要)

Jasper のサイズ内訳

核、util.	servlet機構	XML config.	servlet llib.	280KB
---------	-----------	-------------	---------------	-------

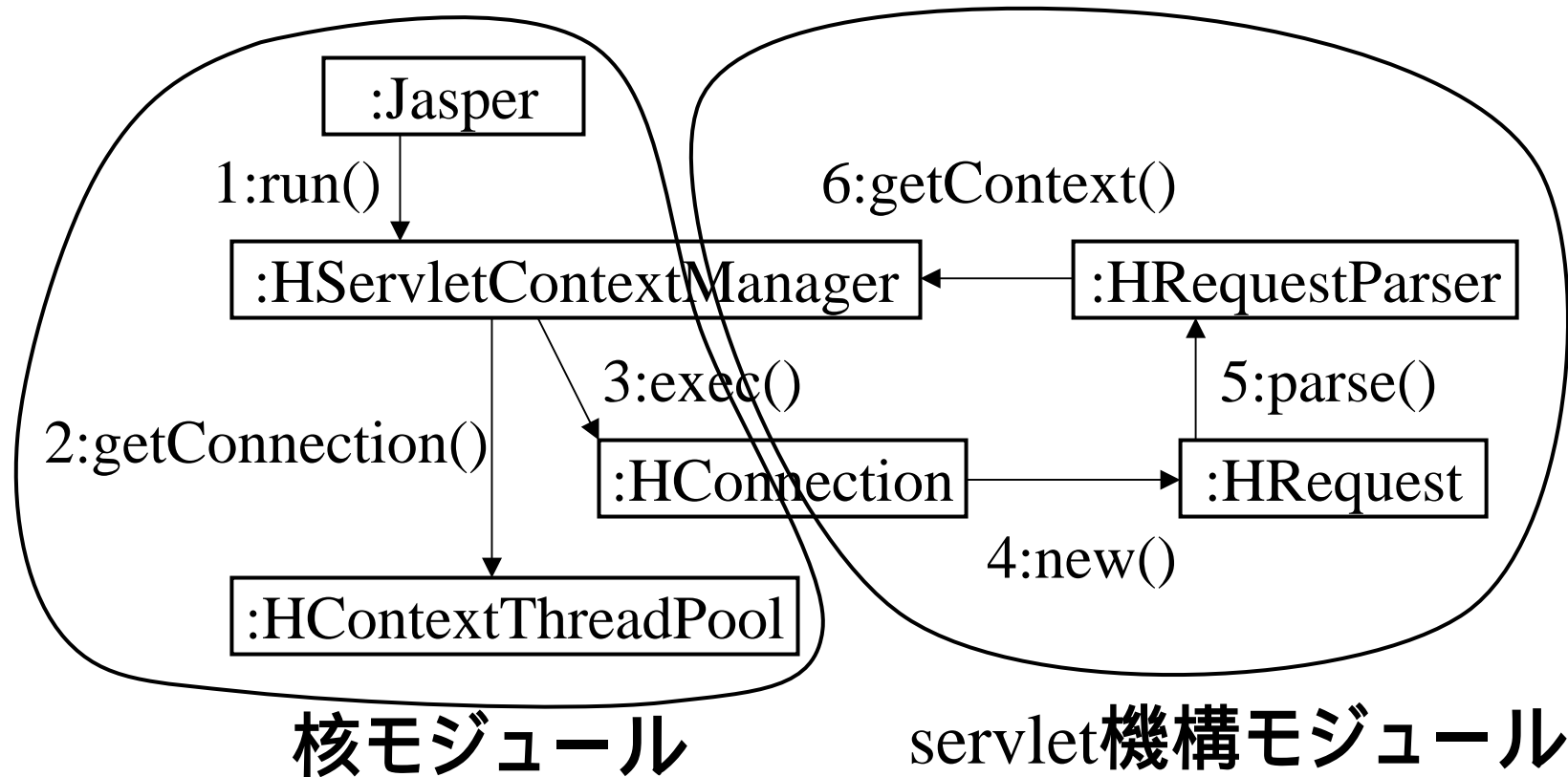
- servlet機構が大きい

HTTP リクエストの処理



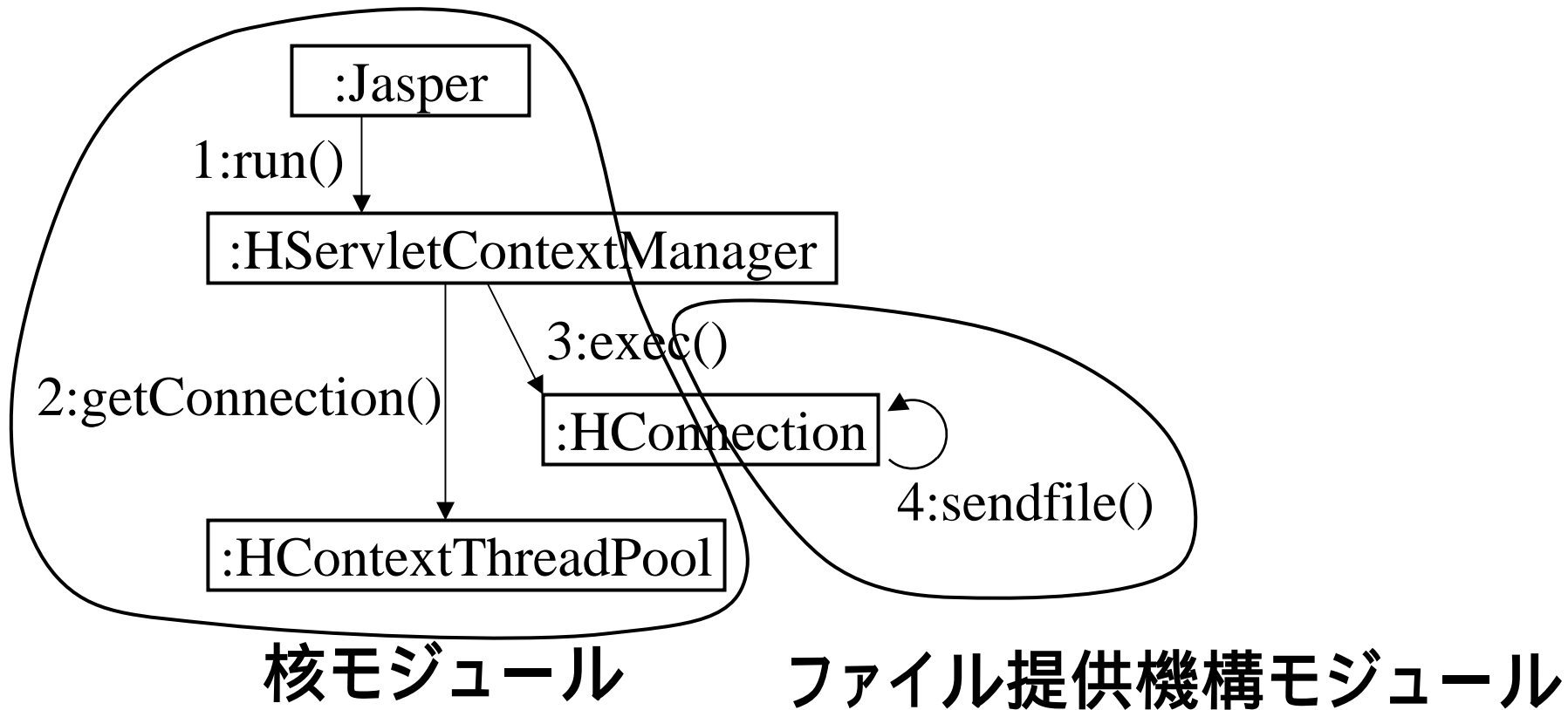
servlet機構を分離することを考えては
設計されていない

MixJuice によるモジュール化



一つのクラスを複数のモジュールに分離可能

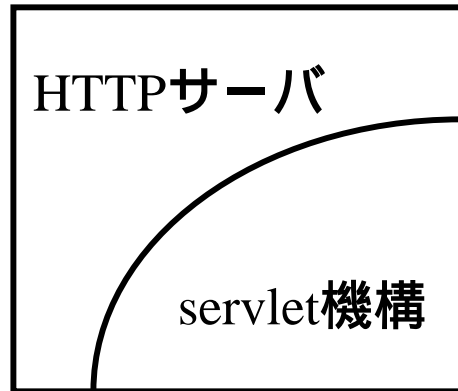
モジュールの差し替え



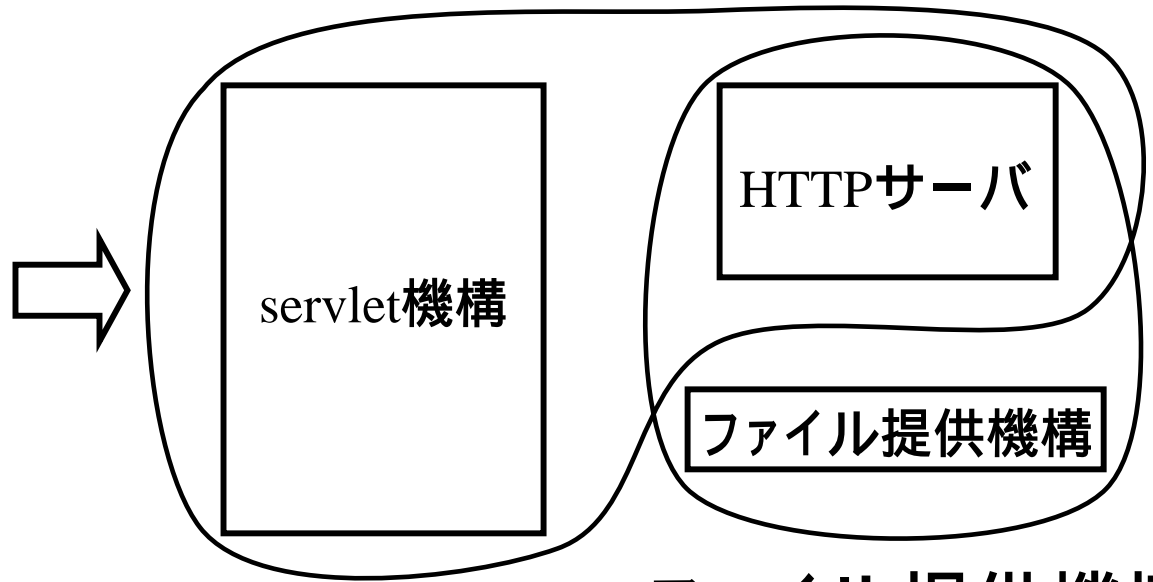
ファイル提供機構:

静的なコンテンツを提供する簡単なモジュール

サイズの変化



A:オリジナル



B:servlet機構 C:ファイル提供機構

Java

MixJuice

A	核、util.	servlet機構	XML config.	servlet lib.	280KB
B	核、util.	servlet機構	XML config.	servlet lib.	360KB
C	核、util.	ファイル提供機構	その他	XML config.	

他の機能選択機構の実装方法

- **機能選択機構：**
機能を選択してプログラムの挙動をカスタマイズするための機構
- **従来の機能選択機構の実装方法：**
 - C の #ifdef
 - Strategy パターン [GoF]
 - AOP, AspectJ [Kiczales, etc.]

機能選択機構の実装手法の比較

機能選択機構	cpp	Strategy	AspectJ	MixJuice
機能選択のタイミング	コン パイ ル時	実行時	コンパイ ル時・ 実行時	リンク時
静的な型・整合性検査	○	○	○	○
機能毎の分割コンパイル	×	○	×	○
不要なコードのリンク	○	×	○	○
バイナリへの選択肢追加	×	○	△	◎
実行効率	◎	△	△	○

まとめ

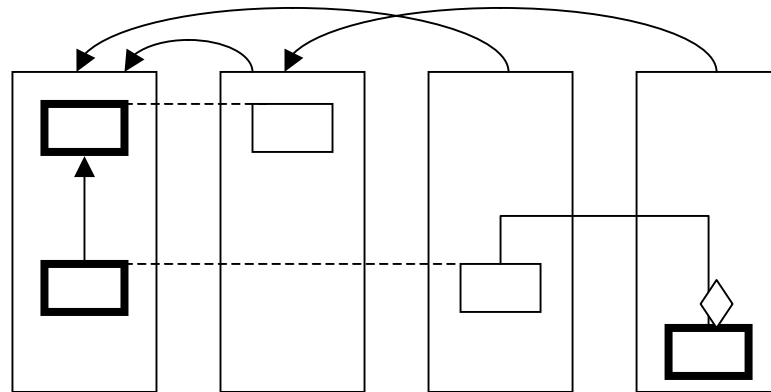
Jasper を case study として次のようなことを確認した

- MixJuice で機能選択機構を容易に実現できる
- 不要な部分を容易に外すことで、アプリケーションの最小構成サイズを小さくできる

レイヤードクラス図

レイヤードクラス図とは？


- プログラムがモジュールによっていかに拡張されるかを説明するための記法
- 普通のクラス図の拡張になっている
 - クラス図で書けることはすべて書ける
- ソースコードから機械生成可能

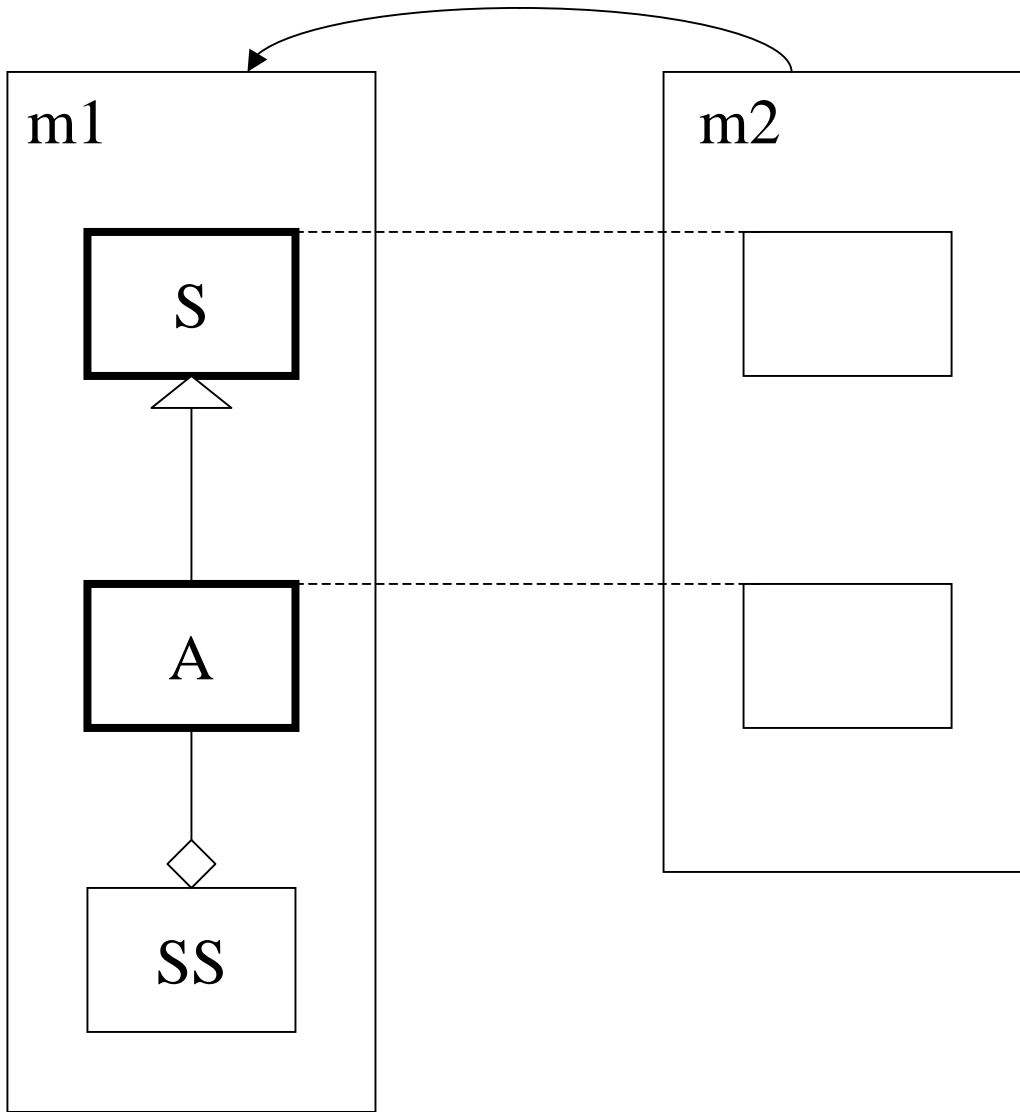


Sample program

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

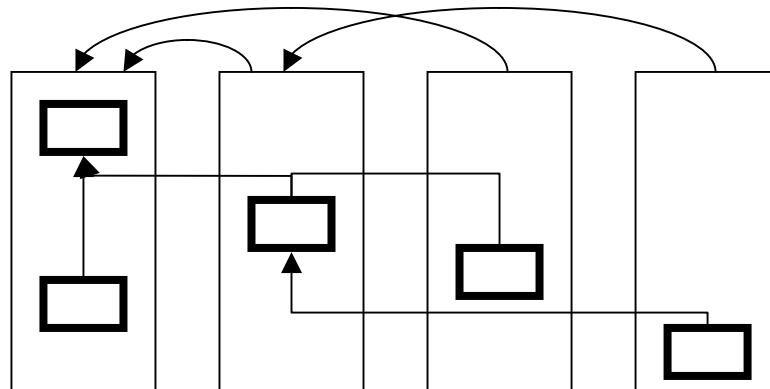
```
module m2 extends m1 {  
  class S { int foo(){ return original() + 2; } }  
  class A { int foo(){ return original() + 20; } }  
}
```





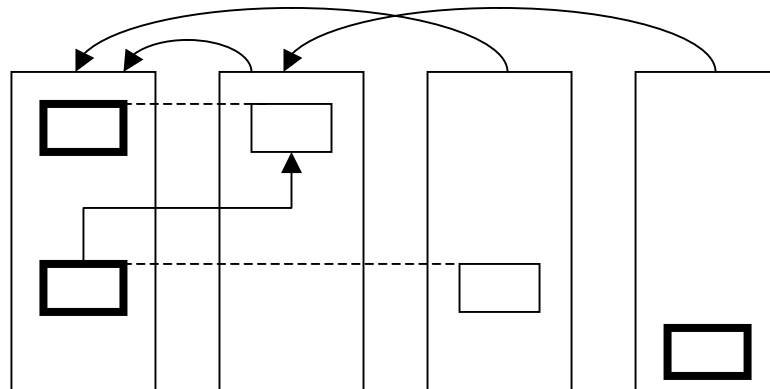
Writing rules (1/3)

- 垂直方向はモジュール、水平方向はクラス。
- モジュールは左から右に継承関係（依存関係）に従って並べる。
- クラスは上から下に継承関係に従って並べる。



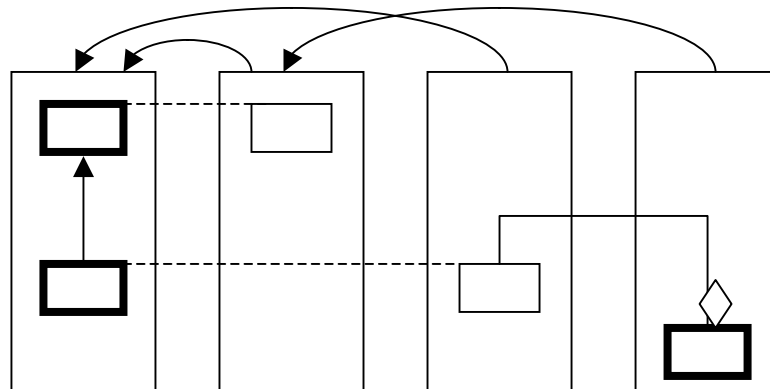
Writing rules (2/3)

- クラス定義は太い四角で書く。
- クラス定義とそれに対する拡張は、同一の垂直位置に書いて、水平の点線で結ぶ。



Writing rules (3/3)

- 継承、関連等の矢印は、それを行っている四角から引く。行き先はどのクラス断片でもよい。



MixJuice によるデザインパター ンの改善

GoF のデザインパターン

- 拡張性の高いプログラムに繰り返し現れる技法のカタログ
- 各パターンの利点と問題点が詳細に述べられている
- MixJuice によって、以下の問題点が改善
 - GoF 本で述べられている問題点約20箇所
 - 述べられていない問題点約10箇所
- 「問題点」は、現在のオブジェクト指向言語の限界を表していたと言える！

例: Visitor pattern

- **導入可能性の問題点**

- 既存のデータ構造のソースを修正して、accept メソッドを実装しないと Visitor パターンを導入することはできない。

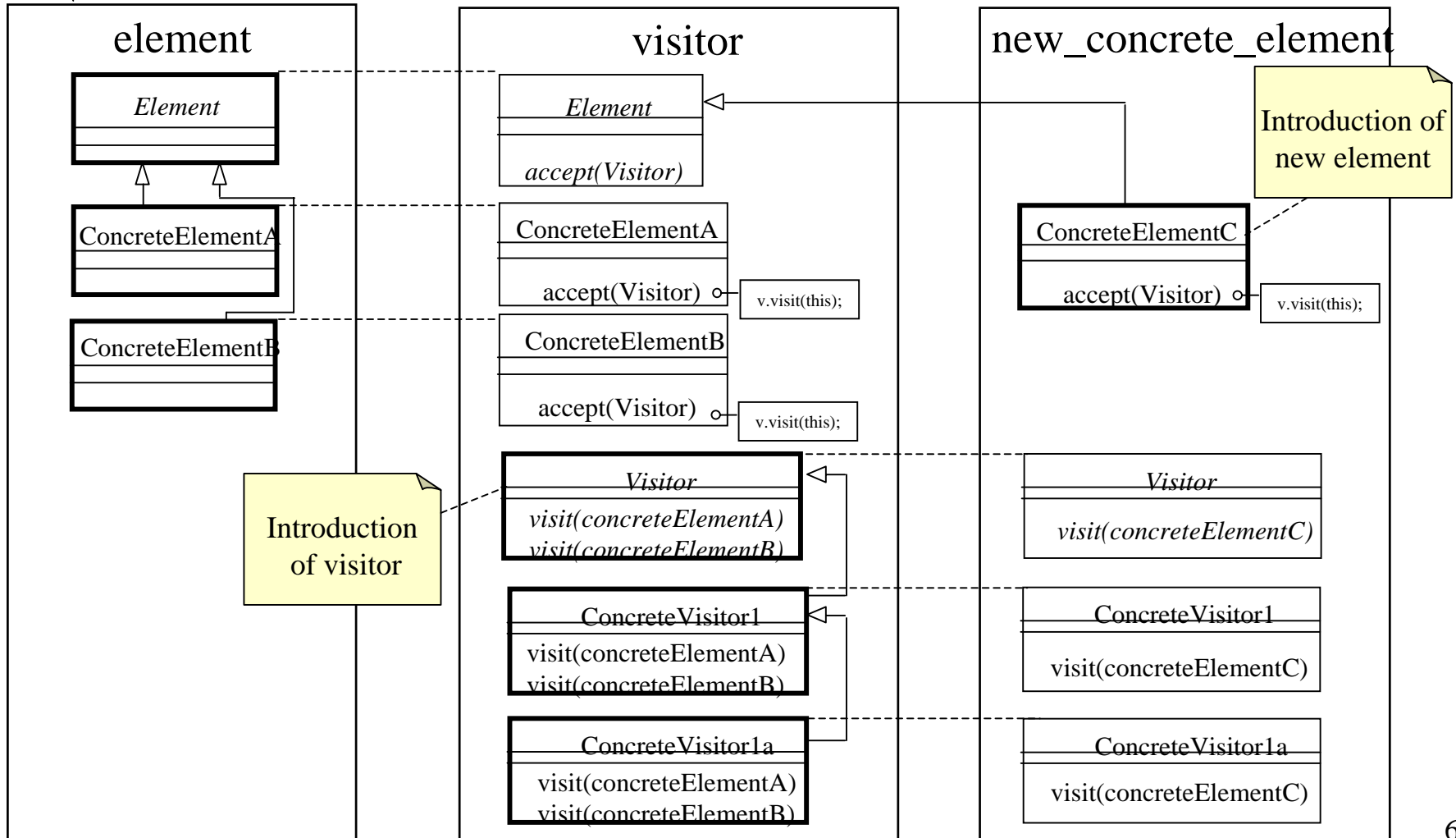
- **拡張性の問題点**

p.358 「3. 新しい ConcreteElement クラスを加えることは難しい」

(Visitor に visit(NewElement) というメソッドを追加する必要がある。)

MixJuice による改善

Tree structure without visitor



MixJuice による改善点の分類

- **導入可能性の改善**
 - 既存のクラスを、新たなパターンの構成要素にすることが可能
- **拡張性の改善**
 - 既存のソースを改変せずに機能を追加可能
- **情報隠蔽の改善**
- **型安全性の改善**
- **クラスモデルの単純化**

デザインパターン	種別	導入	拡張	情報隠蔽	型安全性	単純化
AbstractFactory	改善		p.98			
	別解		p.98		p.100	
Builder	改善		p109			
	別解					
FactoryMethod	改善			p.118		
	別解					
Prototype	改善	p.131				
Singleton	別解					p.138
Adapter	別解		p.153			p.152
Bridge	改善					
	別解					
Composite	なし					
Decorator	改善		p.191			
	別解					p.190
Facade	改善			p.201		
	別解					
Flyweight	なし					
Proxy	なし					
ChainOfResponsibility	改善		p.241			
Command	なし					
Interpreter	改善		p.265			
Iterator	改善			p.280		
Mediator	なし					
Memento	改善			p.307		
Observer	改善		p.318			
State	改善					
Strategy	改善		p.339			
	別解					p.340
TemplateMethod	改善		p.351			
Visitor	改善1			p.359		
	改善2		p.358			
	別解					

MixJuice による
デザインパターン
改善カタログ
(版)

Web にて公開中

モジュールの安全な結合

MixJuice (AOP) に対する もっともな疑問

- 既存のクラスの動作が別のモジュール(アスペクト)によって変更できるのは危険では？
- プログラムの意味を局所的に理解することが不可能では？

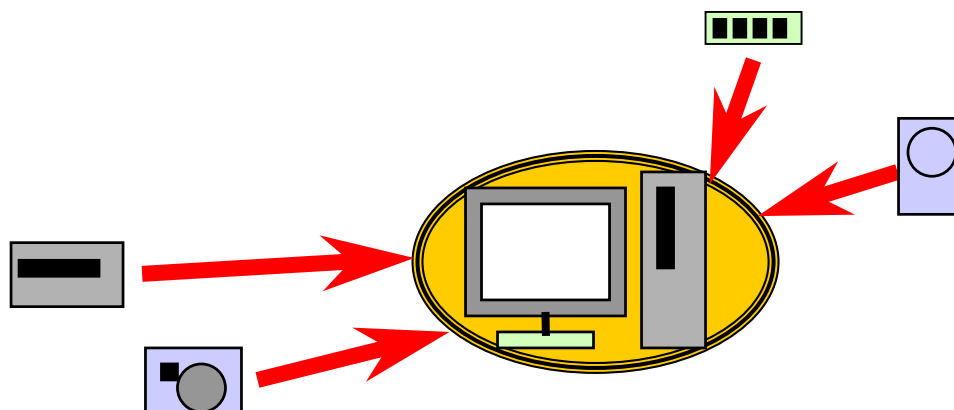
大丈夫！！

ただしプログラマーの協力が必要

アナロジー

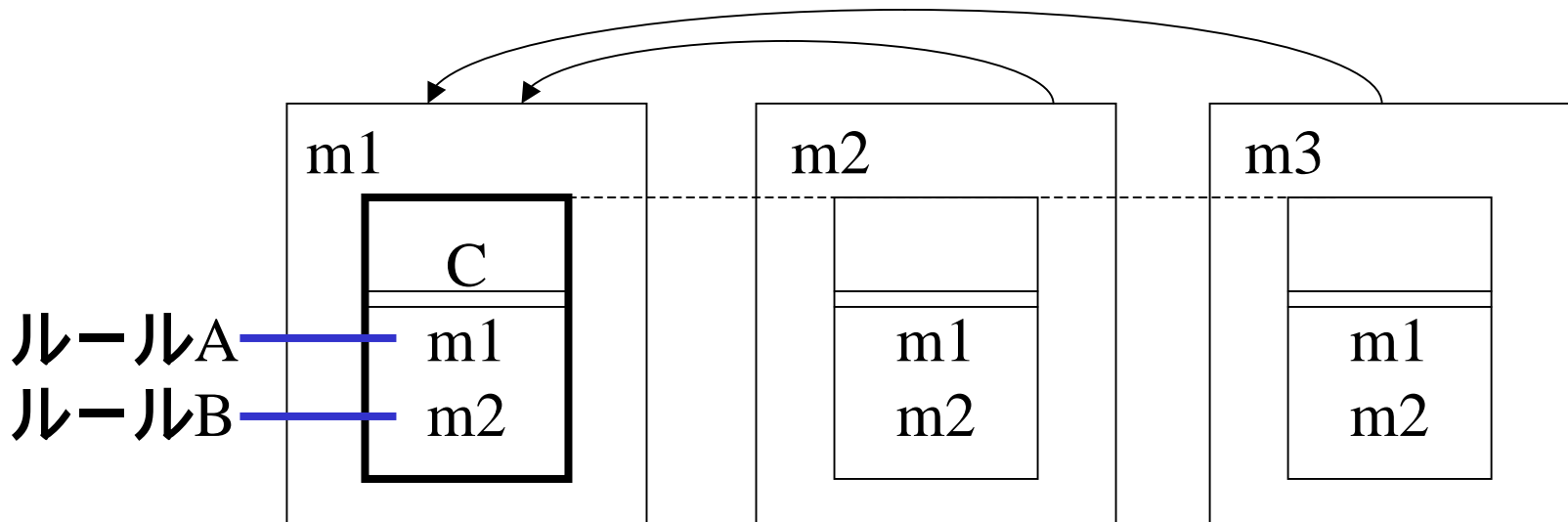
- パソコンと周辺機器

- 安全に接続する規格: USB, IEEE1394, ...
- 本体と周辺機器が、いずれも接続規格を満たしているならば、安全に接続することができる



解決策：「拡張ルール」

- メソッドを定義するときに、それを拡張するための「拡張ルール」も宣言しておく
- すべてのモジュールは、決められたルールに従ってメソッドの振る舞いを拡張する



拡張ルールの例

- 実用上よく使うと思われる4つのタイプのルール
 - after ルール
 - “+” ルール
 - functional protocol ルール
 - disjoint branch ルール

after ルール

- サブクラスによるメソッド拡張は、
 - メソッドの事前条件を弱くできる。
 - 受け取った引数がスーパークラスの事前条件を満たすなら、
 - スーパークラスの事後条件を満たさなければならない。
 - original を最初にちょうど1回呼び出さなければならない。
 - original には自分が受け取った引数をそのまま渡さなければならない。
 - original から受け取った返値と違う返値を返してもよい。
 - スーパークラスから継承した状態を参照・更新してもよい。
 - サブクラス自身が追加した状態を参照・更新してもよい。
 - 受け取った引数がスーパークラスでの事前条件を満たしていないならば、事後条件に制約はない。
- sub-module によるメソッド拡張は、
 - super-module のメソッドの事前条件を変えてはいけない。
 - original を最初にちょうど1回呼び出さなければならない。
 - ...

“+” ルール

- オリジナルの返値に非負の値を足して返してもよい。

```
module m1 {  
  define class C { define int m(){ return v1.size(); } }  
}  
module m2 extends m1 {  
  class C { int m(){ return original() + v2.size(); } }  
}  
module m3 extends m2 {  
  class C { int m(){ return original() + v3.size(); } }  
}
```

MixJuice の拡張 plug-in

Collection plug-in

```
#epp jp.go.etl.epp.Collection
```

```
public class Test {  
    void test(){  
        Vec<String> vec = {"aaa", "bbb", "ccc"};  
        Table<String, int> table = {};  
        foreach (index i, String s in vec){  
            table.put(s, i);  
        }  
        int x = table.get("bbb") ifNull {  
            throw new Error();  
        };  
    }  
}
```

Collection types with type parameters

Type safe foreach statement

Table lookup with enforced null checking

最後に

- 差分ベースモジュールを有する
プログラミング言語 MixJuice
- ソースコードとともに配布中
– <http://staff.aist.go.jp/y-ichisugi/mj/>
- パートナー募集中！
(共同研究、外注作業受注先ソフトハウス、アルバイト、産総研就職希望 etc.)