MixJuice 言語による デザインパターンの改善

田中 哲 akr@m17n.org
一杉 裕志 y-ichisugi@aist.go.jp

産業技術総合研究所 情報処理研究部門

デザインパターンの改善

デザインパターンにはモジュラリティ・拡張性・型安全性などの問題がある
GoF 本にも具体的に載っている

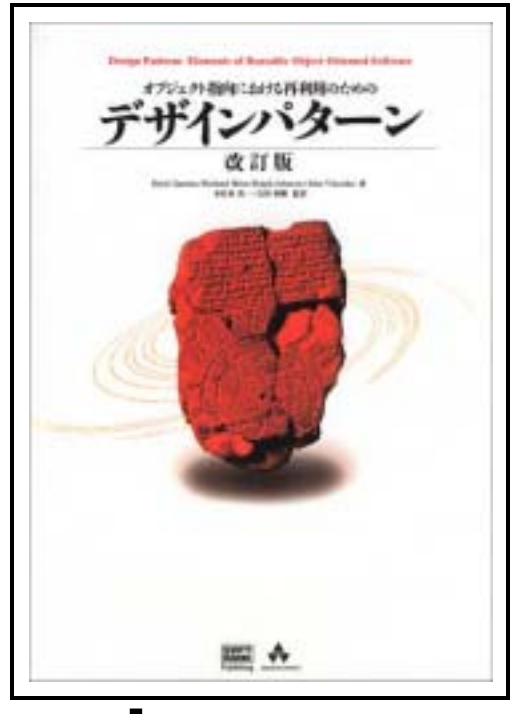
- GoF 本のデザインパターンを網羅的に MixJuice で書き直した
- 載っている問題を 24 個解決
- 載っていない問題を 18 個発見・解決
- カタログを公開
- 問題は従来の言語の制約

発表の流れ

1. デザインパターン
2. MixJuice
3. デザインパターンの改善
4. 改善に役立った機能
5. 他の言語への適用
6. 関連研究
7. まとめ

デザインパターン

OOP における拡張性の高い設計法を集めたカタログ



23 パターン
C++/Smalltalk

「デザインパターン」(GoF 本)

MixJuice

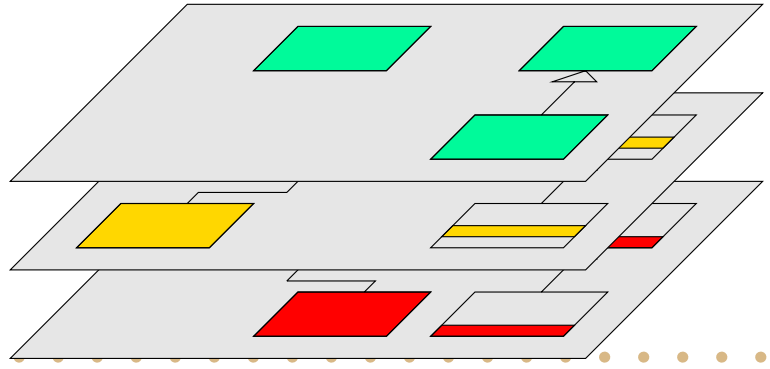
Java のモジュール機構を変更

- AOP 言語の一種
- Crosscutting Concern を分離できる
- 拡張性が高いプログラムが書きやすい

MixJuice のモジュール機構

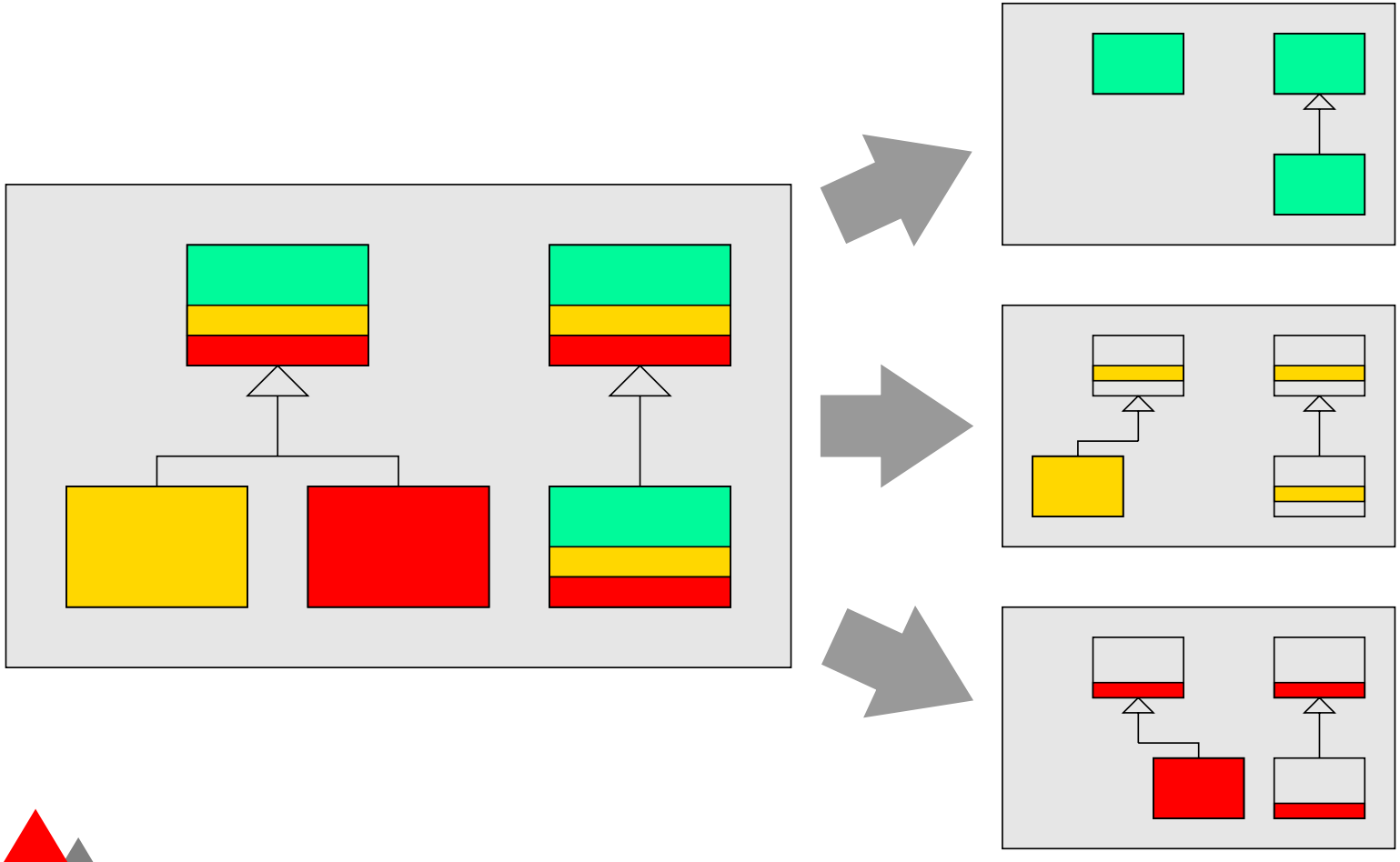
既存のモジュールが定義したクラスを変更できる

- 既存のクラスへのスーパーインターフェース・フィールド・メソッドの追加
- 既存のメソッドのオーバーライド
- 新しいクラスの導入
- etc.

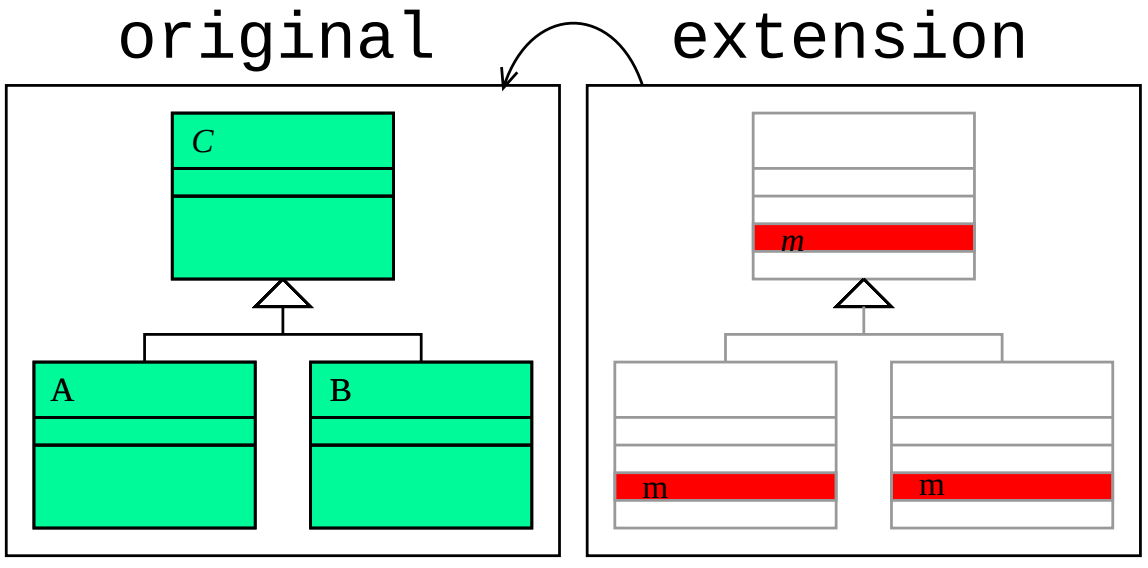


Crosscutting Concerns

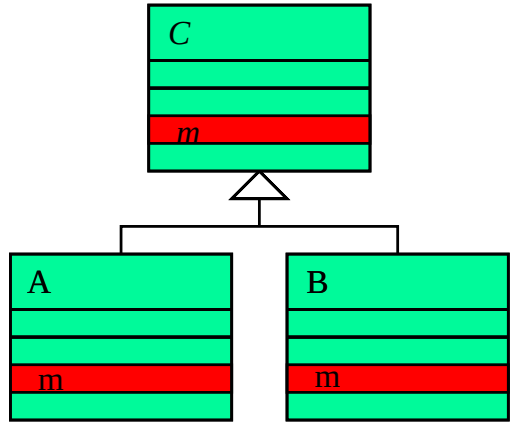
クラスをまたがる観点をモジュール化



メソッド追加



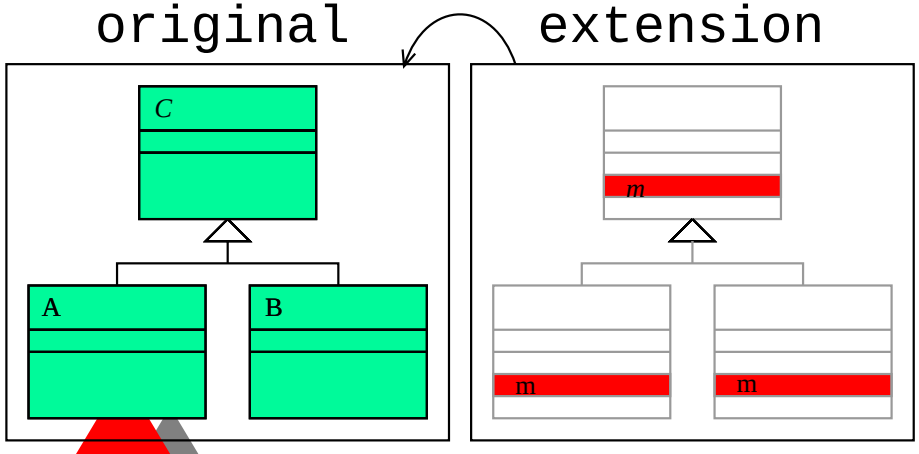
before link



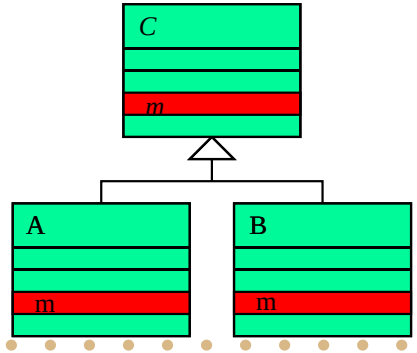
after link

メソッド追加: プログラム

```
module original {
  define abstract class C { ... }
  define class A extends C { ... }
  define class B extends C { ... }
}
module extension extends original {
  class C { define abstract void m(); }
  class A { void m() { ... } }
  class B { void m() { ... } }
}
```




```
class C {
  ...
  abstract void m();
}
class A extends C {
  ...
  void m() { ... }
}
class B extends C {
  ...
  void m() { ... }
}
```



MixJuice と GoF パターン

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	使用するプログラミング技法
AbstractFactory	改善		p.98				メソッド追加
	別解		p.98		p.100	○	実装モジュール選択
Builder	改善		p109			○	メソッド追加・メソッド拡張
	別解					○	メソッド拡張・実装モジュール選択
FactoryMethod	改善			p.118			名前空間分離
	別解				○	○	実装モジュール選択
Prototype	改善	p.131					メソッド追加
Singleton	別解					p.138	クラスメソッド拡張
Adapter	別解		p.153			p.152	スーパーインターフェース追加
Bridge	改善		○				メソッド追加
	別解				○	○	実装モジュール選択
Composite	改善	○					スーパーインターフェース追加
Decorator	改善		p.191				メソッド追加
	別解					p.190	クラス拡張
Facade	改善			p.201			名前空間分離
	別解					○	クラスメソッド追加
Flyweight	なし						
Proxy	なし						
ChainOfResponsibility	改善	○	p.241				スーパーインターフェース追加・メソッド追加
Command	なし						
Interpreter	改善	○	p.265				メソッド追加
Iterator	改善			p.280			名前空間分離
Mediator	なし						
Memento	改善			p.307			名前空間分離
Observer	改善	○	p.318				クラス拡張
State	改善		○				メソッド追加
Strategy	改善		p.339				メソッド追加
	別解					p.340	実装モジュール選択
TemplateMethod	改善		p.351			○	メソッド実装
Visitor	改善 1			p.359			フィールド追加・仕様/実装モジュールの分離
	改善 2	○	p.358				メソッド追加
	別解	○	p.358	p.359		○	メソッド追加

MixJuice と GoF パターン

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	使用するプロ
AbstractFactory	改善		p.98				メソッド追加
	別解		p.98		p.100	○	実装モジュー
Builder	改善		p109			○	メソッド追加
	別解					○	メソッド拡張
FactoryMethod	改善			p.118			名前空間分離
	別解				○	○	実装モジュー
Prototype	改善	p.131					メソッド追加
Singleton	別解					p.138	クラスメソッ
Adapter	別解		p.153			p.152	スーパーイン
Bridge	改善		○				メソッド追加
	別解				○	○	実装モジュー
Composite	改善	○					スーパーイン
Decorator	改善		p.191				メソッド追加
	別解					p.190	クラス拡張
Facade	改善			p.201			名前空間分離
	別解					○	クラスメソッ
Flyweight	なし						
Proxy	なし						
ChainOfResponsibility	改善	○	p.241				スーパーイン
Command	なし						
Interpreter	改善	○	p.265				メソッド追加
Iterator	改善			p.280			名前空間分離

Visitor パターン

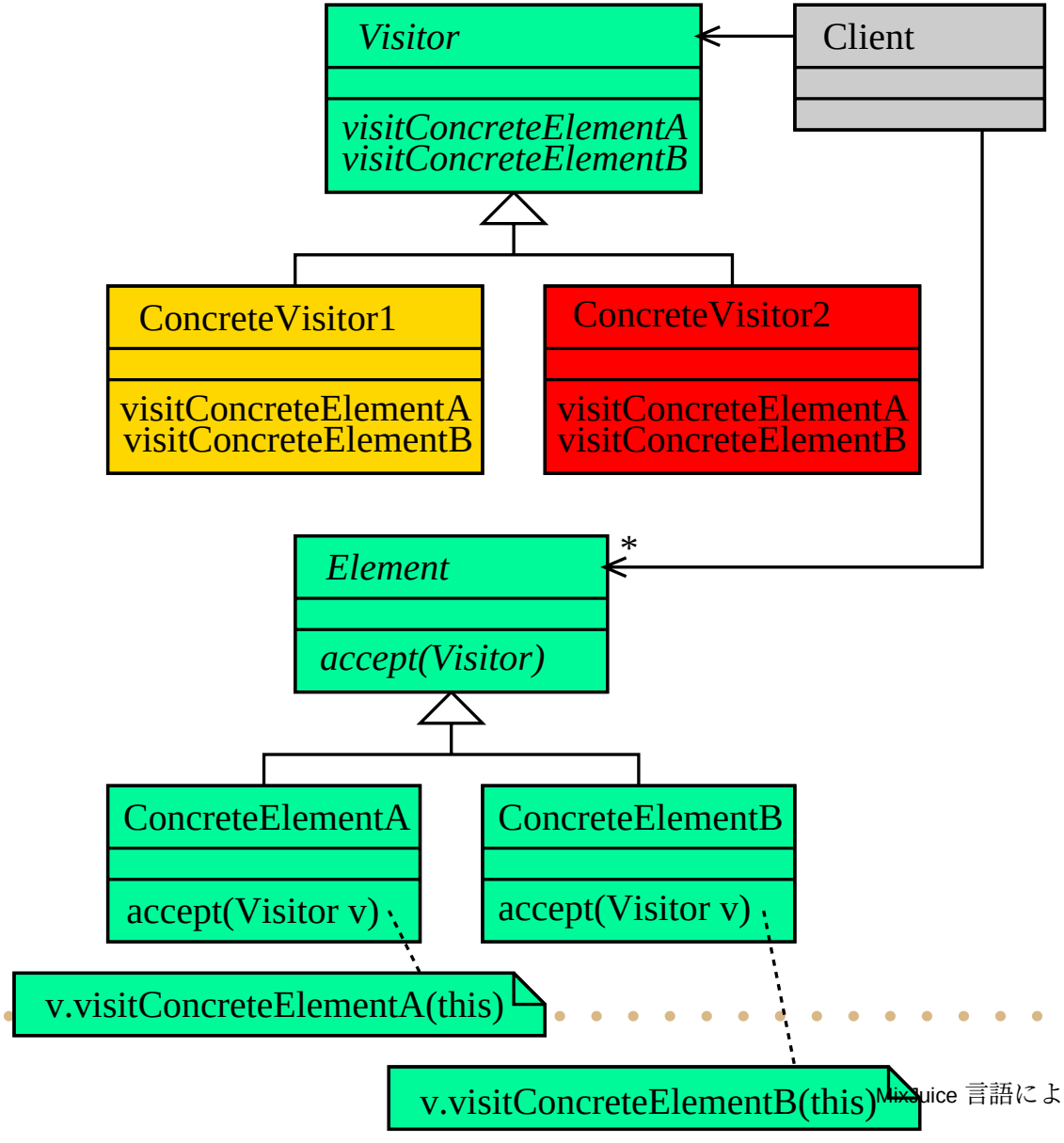
目的: オブジェクト構造に適用するオペレーションを

- ソースコードを修正せずに
- 追加可能

にする

例: 言語処理系の構文木に対する「型検査」「最適化」「コード生成」「プリティプリント」などの追加

Visitor パターンの構造



Visitor パターンの問題点

問題 (1) オブジェクト構造が拡張不能

問題 (2) オブジェクト構造が情報隠蔽不能

問題 (3) 記述が繁雑

各問題を MixJuice **で解決**

問題(1): 拡張性

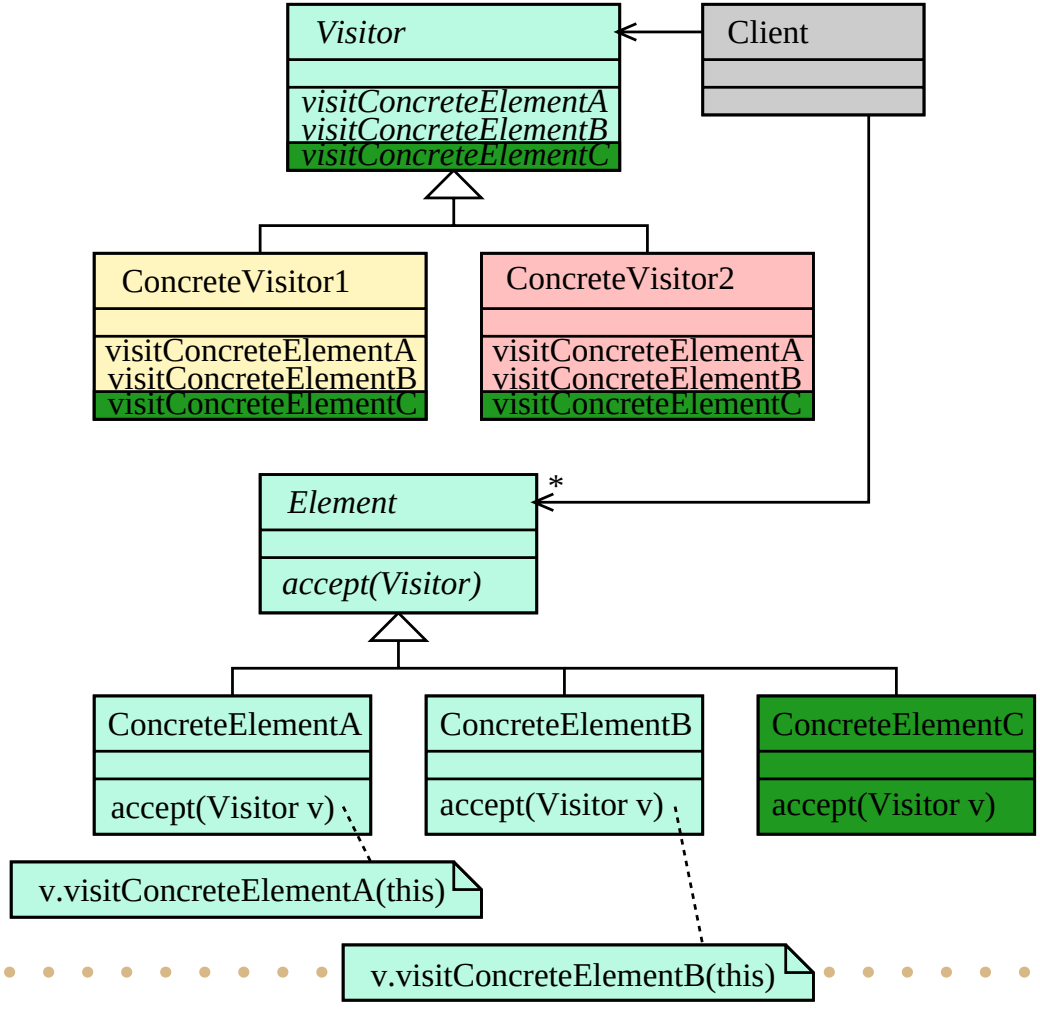
オブジェクト構造に新しい要素を導入できない

GoF 本,p.358,l.7–12:

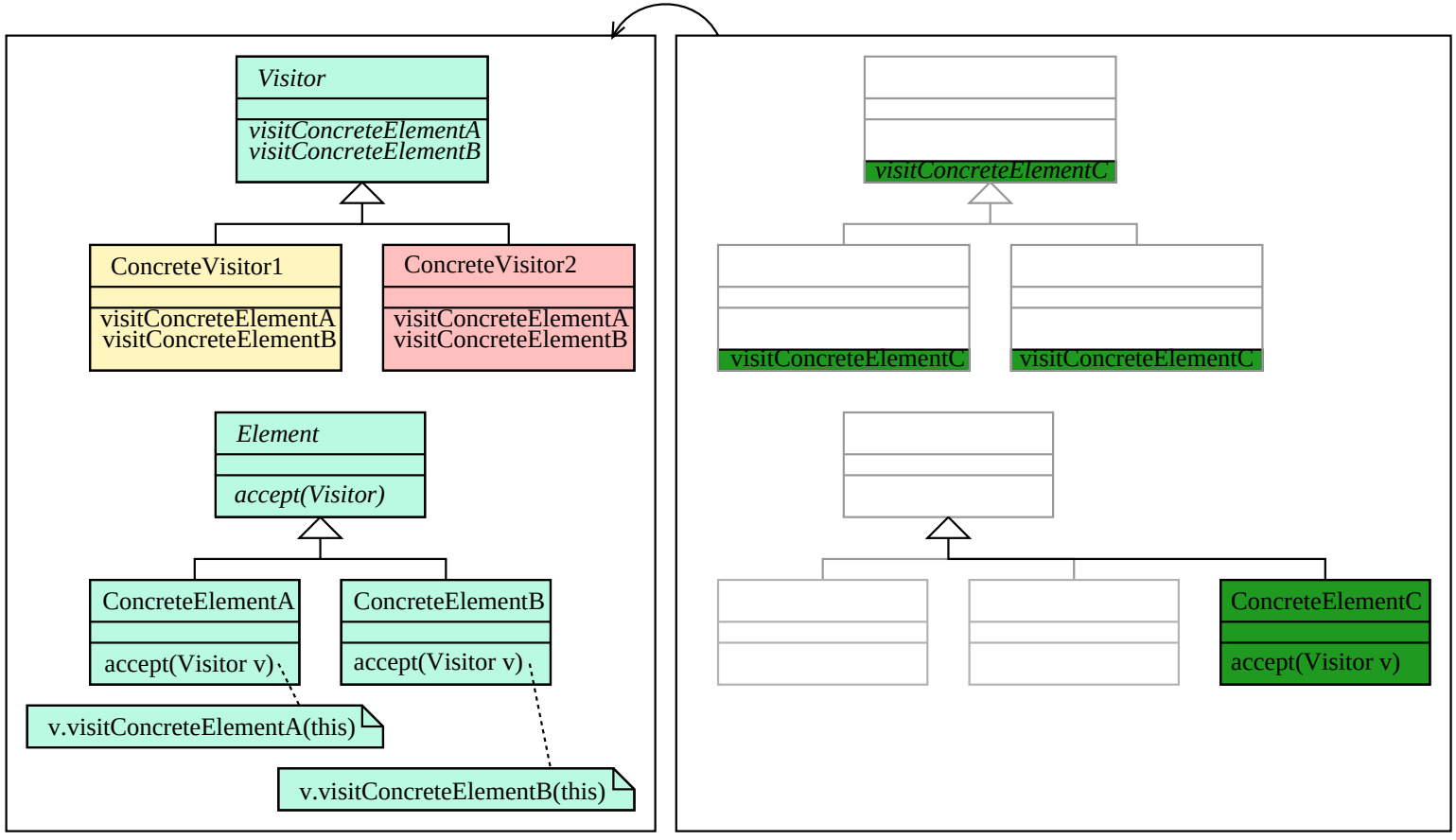
3. 新しい ConcreteElement クラスを加えることは難しい。Visitor パターンでは、Element の新しいサブクラスを加えることを難しくする。新しい ConcreteElement クラスを導入することにより、Visitor クラスでは新しい抽象化されたオペレーションを宣言し、各 ConcreteVisitor クラスではそれに対応する実装を行なわなければならなくなる。デフォルトの実装を Visitor クラスに与えて、ほとんどの ConcreteVisitor クラスにこれを継承させることもときにはできるだろう。しかし、これは例外的な場合である。

オブジェクト構造が拡張不能

ソースコードの修正が必要



改善(1): Visitor クラスにメソッド追加



改善(1) の利点・欠点

- ○ オブジェクト構造を拡張可能
- ○ (既存のデザインパターンの構造のままで適用可能)
- × (オブジェクト構造の拡張とオペレーションの追加を同時に行なった場合には補完モジュールが必要)

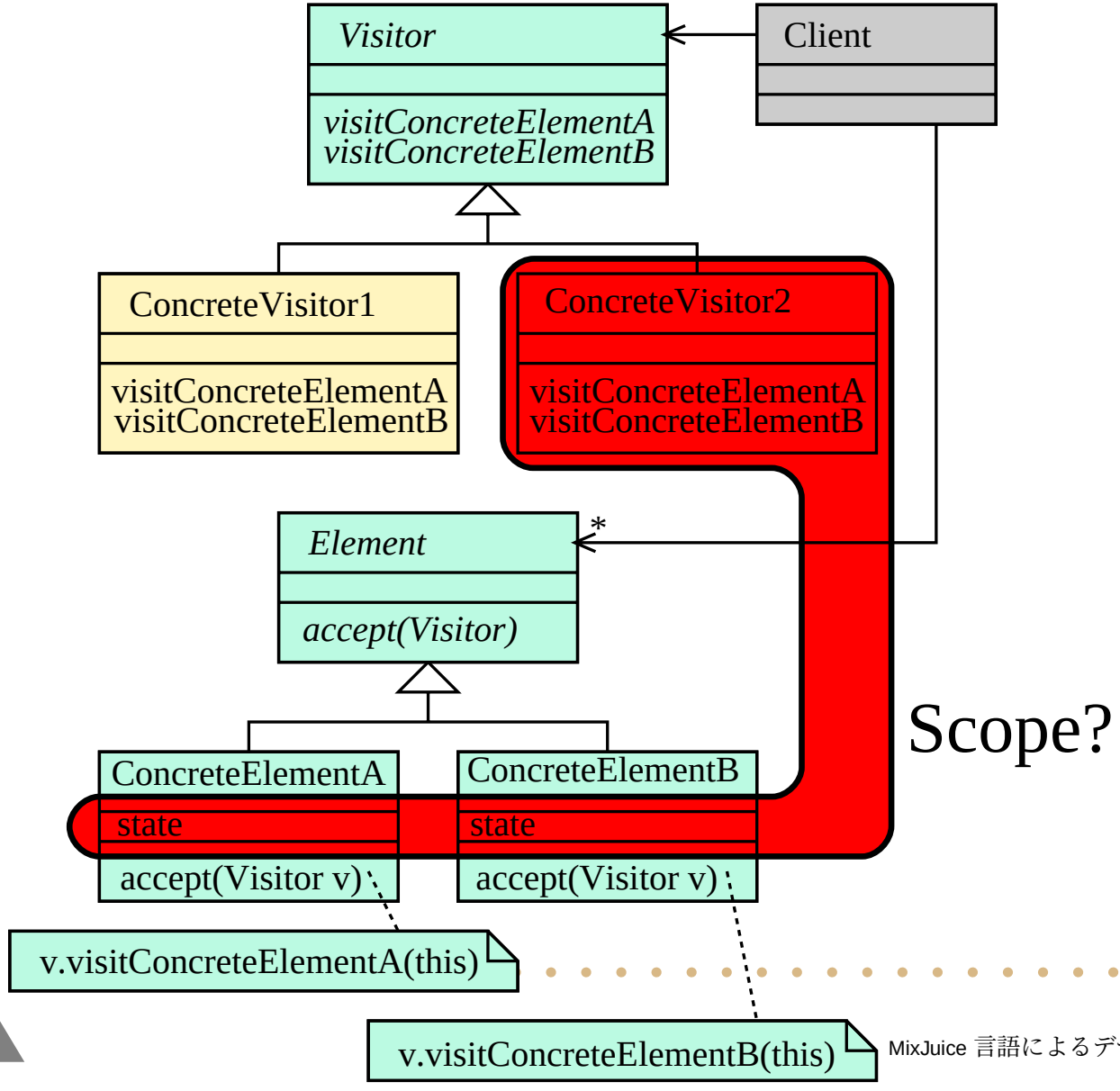
問題(2): 情報隠蔽

オブジェクト構造が情報隠蔽不能

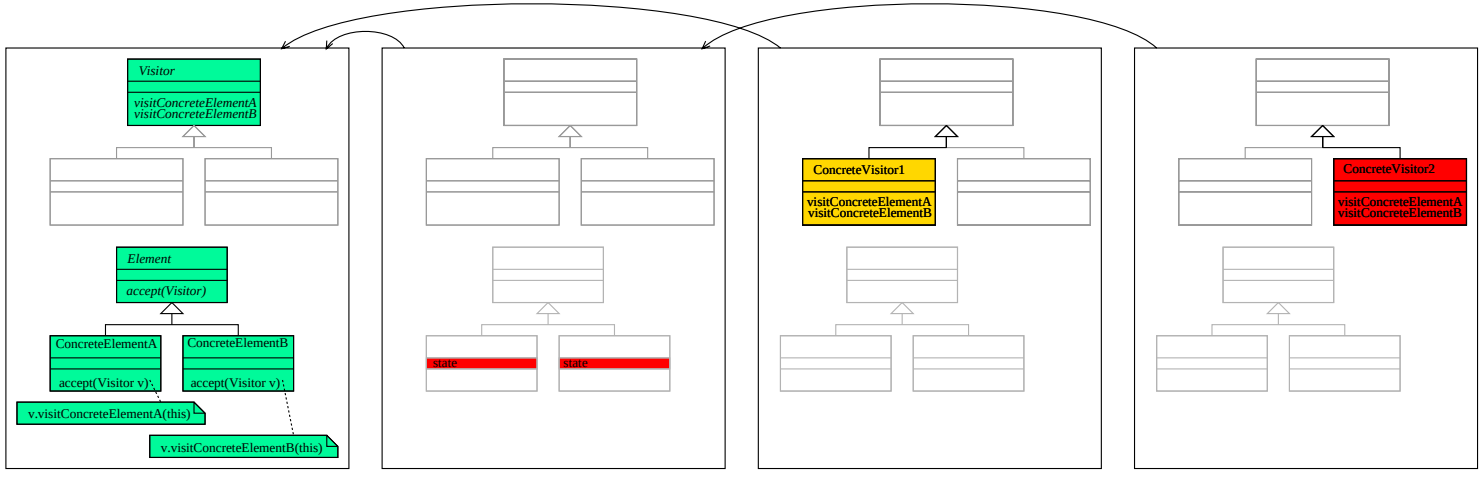
GoF 本,p.359,l.5–8:

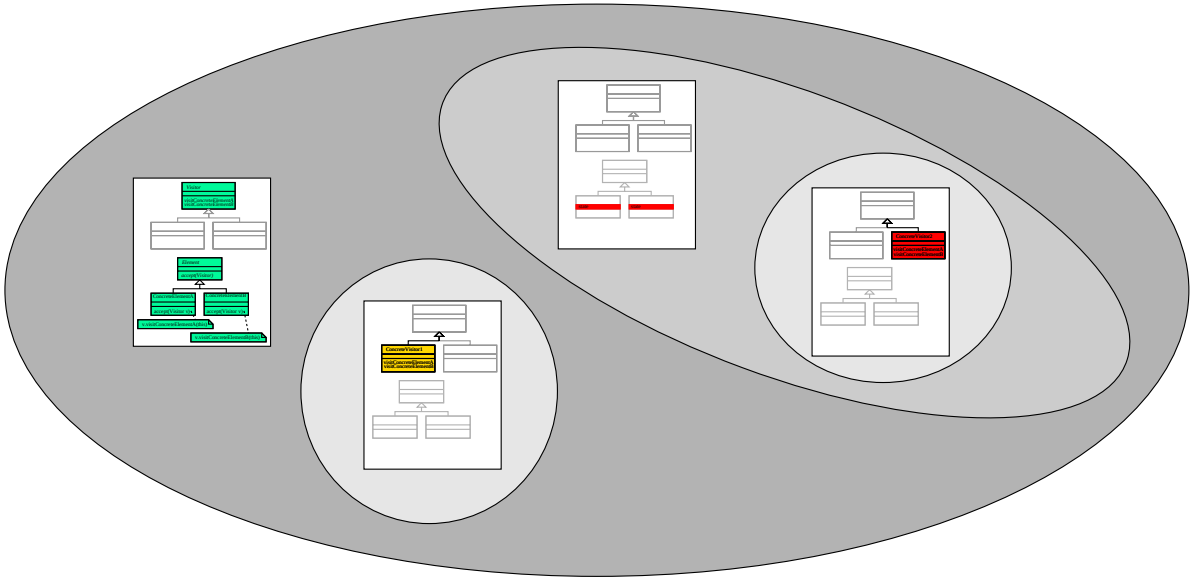
> 6. **カプセル化を破る。**visitor **によるアプローチでは、**ConcreteElement **クラスのインターフェースが、**visitor **が仕事を行うのに十分、強力であることを仮定している。その結果、このパターンでは要素の内部状態にアクセスする公開オペレーションを提供するように強いられることがしばしばある。したがって、カプセル化に対して妥協を与えることになるかもしれない。**

クラス単位では情報隠蔽不能



改善 (2): オブジェクト構造を仕様・実装モジュールに分離



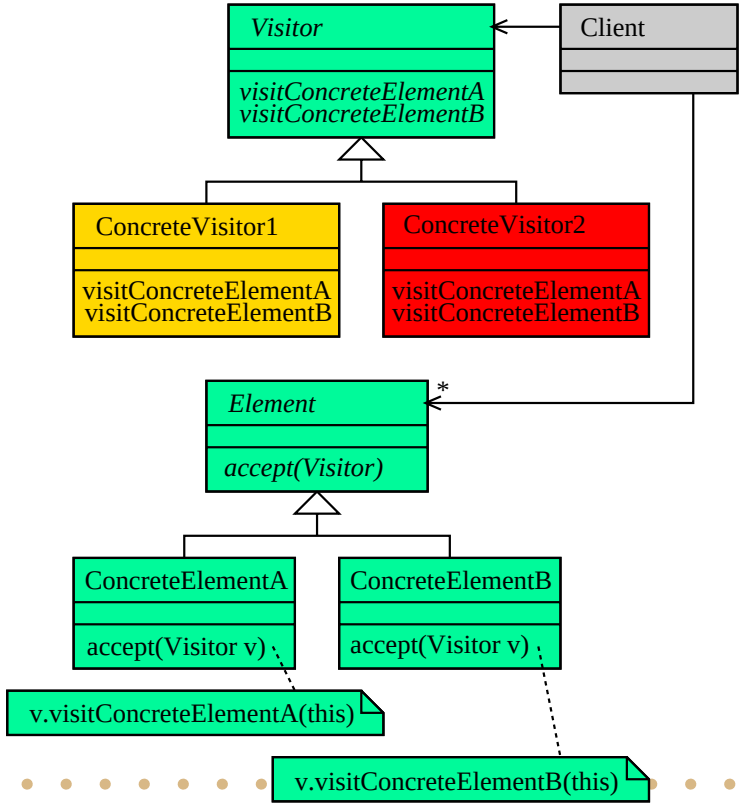


改善 (2) の利点

- ○ オブジェクト構造の公開オペレーションと内部状態のスコープを分離可能

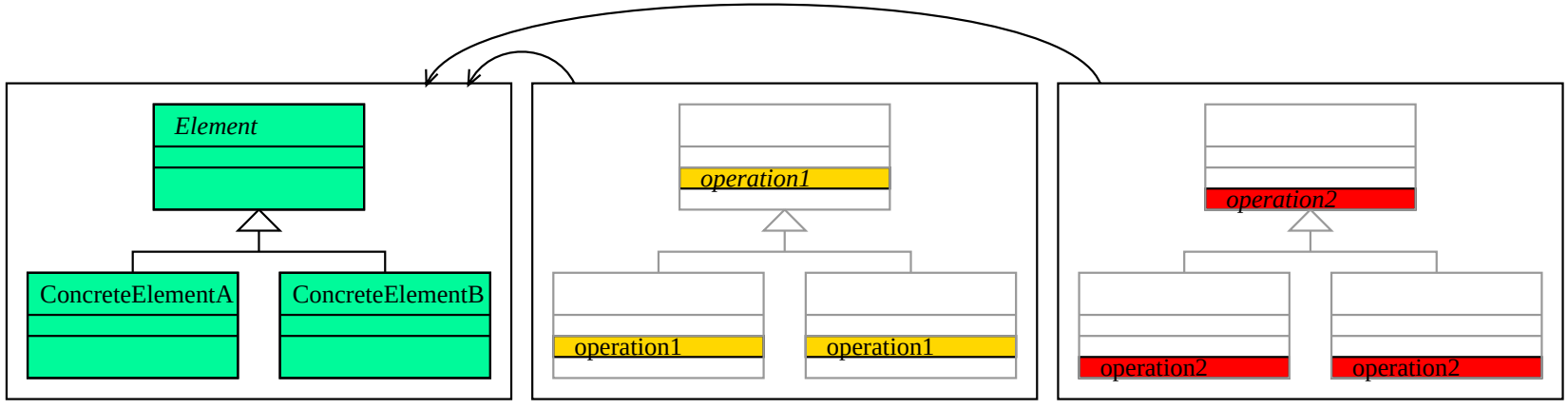
問題(3): 繁雑さ

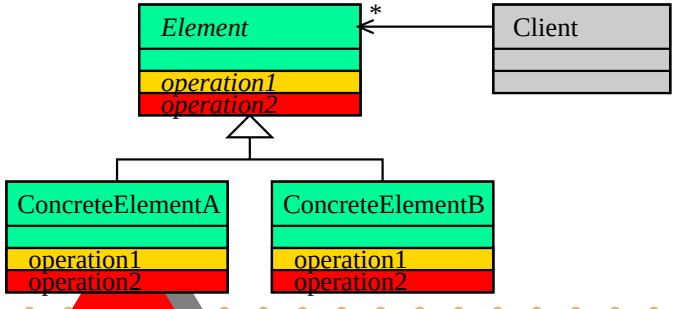
オブジェクト構造のメソッドによる直接的な再帰と比べて、Visitor パターンは繁雑で書きにくい

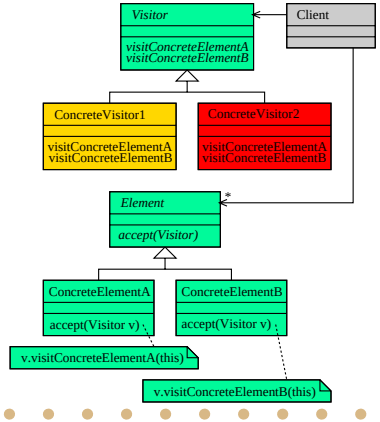


改善(3) オブジェクト構造にメソッド追加

メソッド追加によってオペレーションをモジュール化する

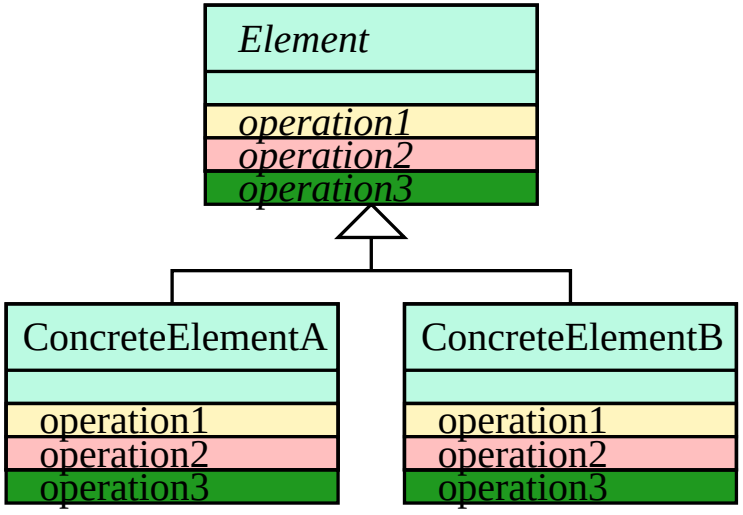


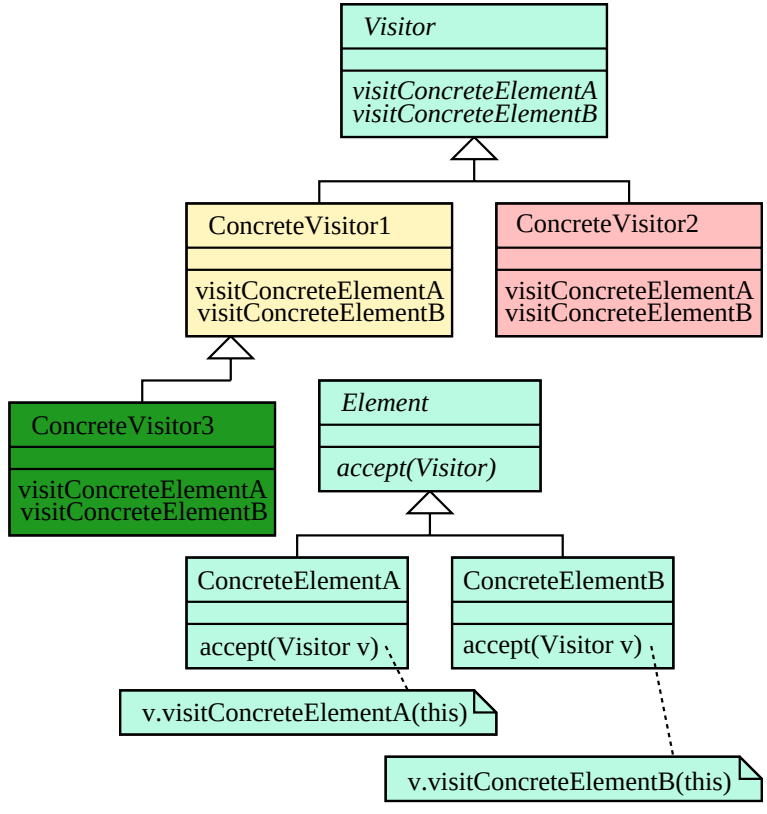




改善(3)の欠点

複数のオペレーションで共通の処理を再利用することは困難





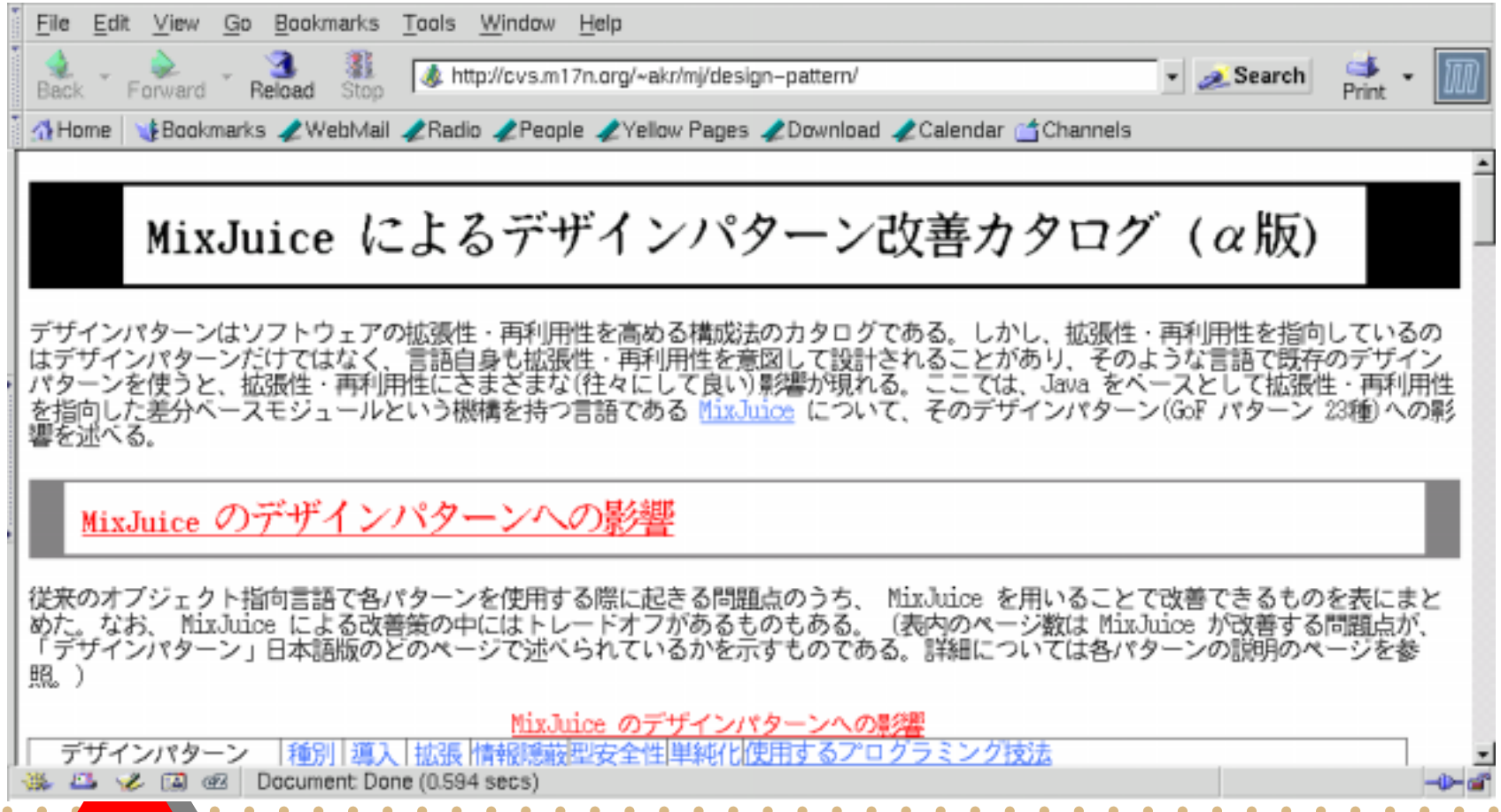
改善(3)の利点・欠点

- ○ 記述が単純化
- × 複数のオペレーションで共通の処理を再利用することは困難
- ○ (オブジェクト構造を拡張可能)
- × (オブジェクト構造の拡張とオペレーションの追加を同時に行なった場合には補完モジュールが必要)

MixJuiceによるデザインパターン改善カタログ

GoF パターン 23 種

http://cvs.m17n.org/~akr/mj/design-pattern/



カタログの構造

- パターンの目的
- GoF パターンのクラス図
- 改善策 (1)
  - GoF パターンの問題点 (引用)
  - 対策
  - 結果 (利点・欠点)
  - 構造 (図・コード)
- 改善策 (2) ...

MixJuice の何が役に立ったのか?

- 拡張手段の増加
  既存のモジュールが定義したクラスを変更できる
- クラス独立な情報隠蔽機構
  クラスをまたがったスコープをつくれる
- (実装モジュール選択による静的な多態性)
  リンク時にモジュールを選択できる

他の言語でのデザインパターンの改善

AspectJ, CLOS, Ruby:

技法	AspectJ	CLOS	Ruby
スーパーインターフェース追加	declare parents		include
フィールド追加	introduction		代入
メソッド追加		defmethod	def
メソッド拡張	around advice		alias/def
名前空間分離	aspect	package	
仕様モジュールと実装モジュールの分離			
実装モジュール選択	aspect 選択		require 切替え

MixJuice 以外でも必要な機能を備えていれば同様な改善が可能

関連研究: AspectJ

Design Pattern Implementation in Java and AspectJ[Hannemann,OOPSLA2002]

AspectJ **で** GoF **パターンを書き直した**

	AspectJ	MixJuice
モジュラリティの改善	○	○
拡張性の改善		○
ライブラリ化	○	
型安全性		○
コード配布	○	
図による説明		○
スタイル	aspect	introduction

改善されたデザインパターンの使用例

拡張可能 Java プリプロセッサ EPP の MixJuice による再実装

- Visitor パターン (改善 (3))
  抽象構文木を処理するパスの追加
- Decorator パターン
  メソッド拡張により再帰降下パーザの導出規則を追加
- Chain of Responsibility パターン
  環境クラスへのメソッド追加

まとめ

- GoF パターンがどのように改善されるかを網羅的に検討
- カタログにまとめて公開
  `http://cvs.m17n.org/~akr/mj/design-pattern/`
- 実際に役に立った MixJuice の機能
  - 拡張手段の増加
  - クラス独立な情報隠蔽機構
  - 実装モジュール選択による静的な多態性

これ以降は質問用のスライド

安全な拡張

- 既存のクラスへのスーパーインターフェース・フィールド・メソッドの追加
- 新しいクラスの追加
- etc.

新しく追加したものは既存のモジュールからは不可視なので影響を与えない
ただし「既存のメソッドのオーバーライド」を除く

安全なオーバーライド(1)

- 安全な拡張 = 正しさが保存される拡張
- 正しさ = 実装が仕様を満たす証明/確信

→ **証明が保存される拡張は安全**

手続きの事前条件を弱く、事後条件を強くしても、Hoare Logic **による証明の正しさは保存される**

Liskov Substitution Principle **を満たすようにオーバーライドを行ない、**Design by Contract **に従ってプログラムすればよい**

安全なオーバーライド(2)

多重継承・Mixin・MixJuiceのモジュール・AOPなどで、ひとつのメソッドに複数の拡張が施される場合、ある拡張を実装するプログラマはプログラム時に拡張前の正確な事前条件・事後条件を知ることができない

複数の拡張が協調するための規則が必要

安全なオーバーライド(3)

複数のモジュールがメソッドをオーバーライドした時に全体として確実に、事前条件が弱く、事後条件が強くなる規則:

after super を最初に呼び出すことにより上から順に実行する。上の事後条件を壊してはいけない

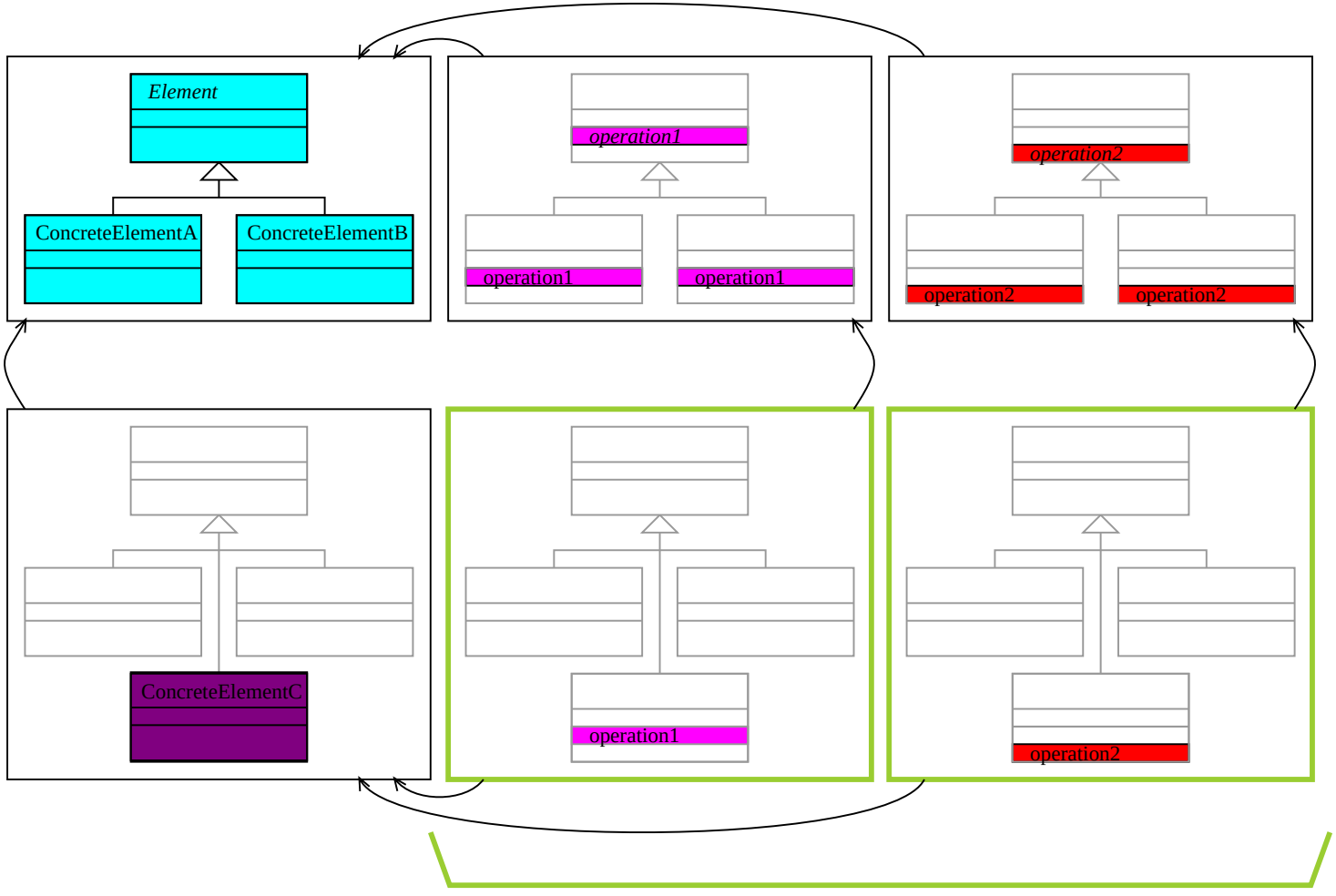
plus 返り値や副作用が単調に増えていく

functional protocol 関数的な挙動(cache)

disjoint branch 定義域を広げていく

etc.

補完モジュール



complementally modules

デザインパターンの拡張性

「デザインパターンの拡張性」
「ソースコードを修正せずに追加可能」

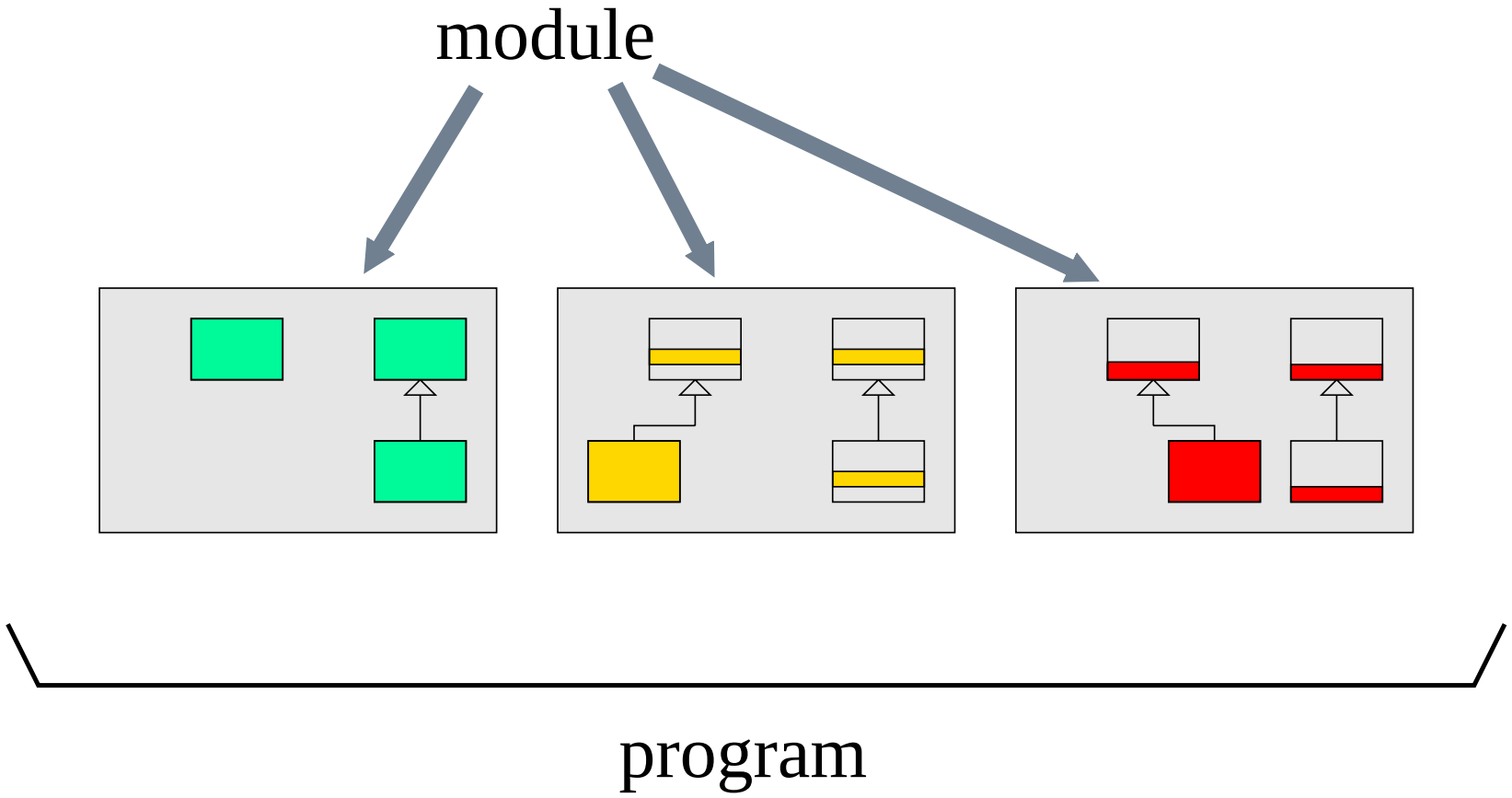
Abstract Factory 部品群を追加可能

Builder 生成処理を追加可能

…

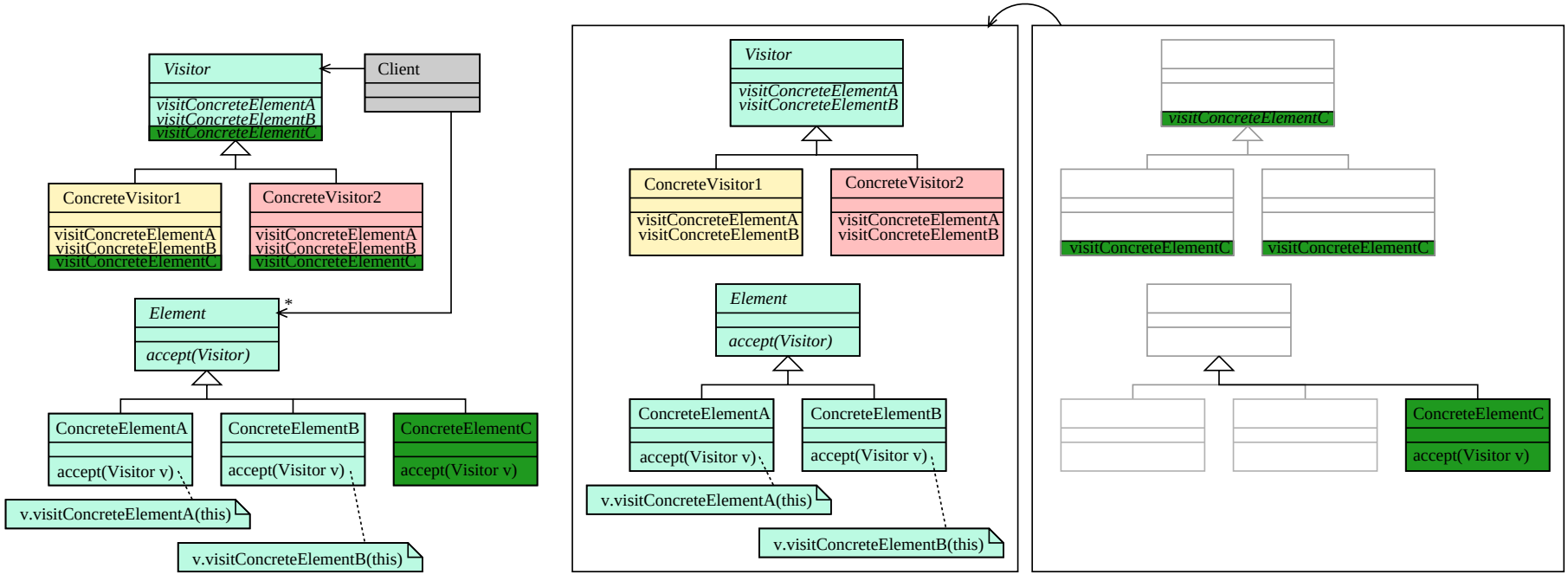
Visitor オペレーションを追加可能

観点をモジュールにできる



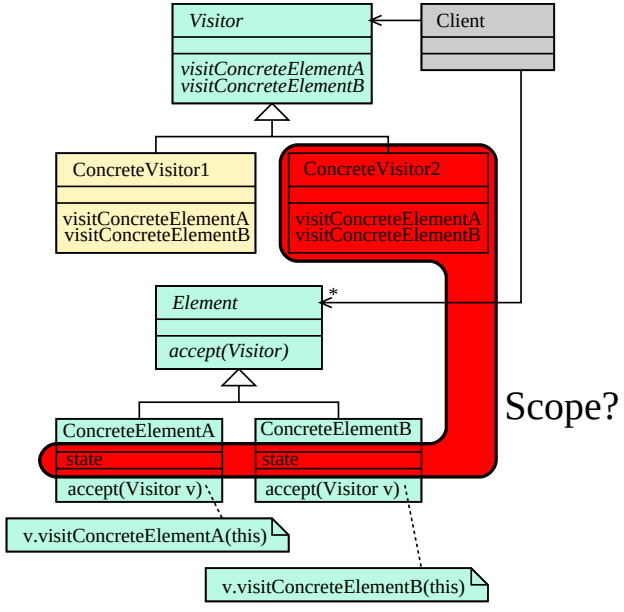
拡張手段の増加

既存のモジュールが定義したクラスを変更できる

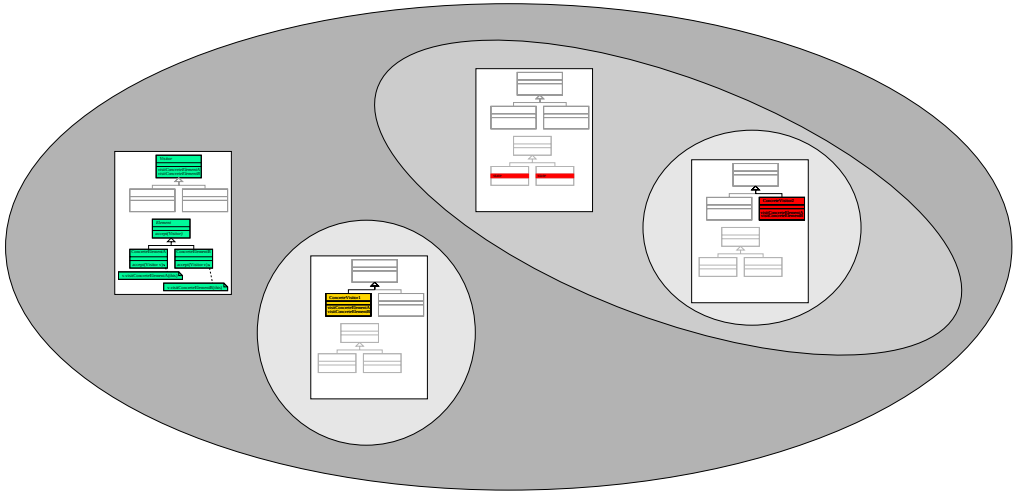


クラス独立な情報隠蔽機構

クラスをまたがったスコープをつくれる



Scope?



実装モジュール選択による静的な多態性

リンク時にモジュールを選択できる

- ○ 型安全性が高まる (downcast が減る)
- ○ クラス数が減少して単純化
- × 動的なことはできない

関連研究: Unit Model

Static and Dynamic Structure in Design Patterns[ICSE2002]

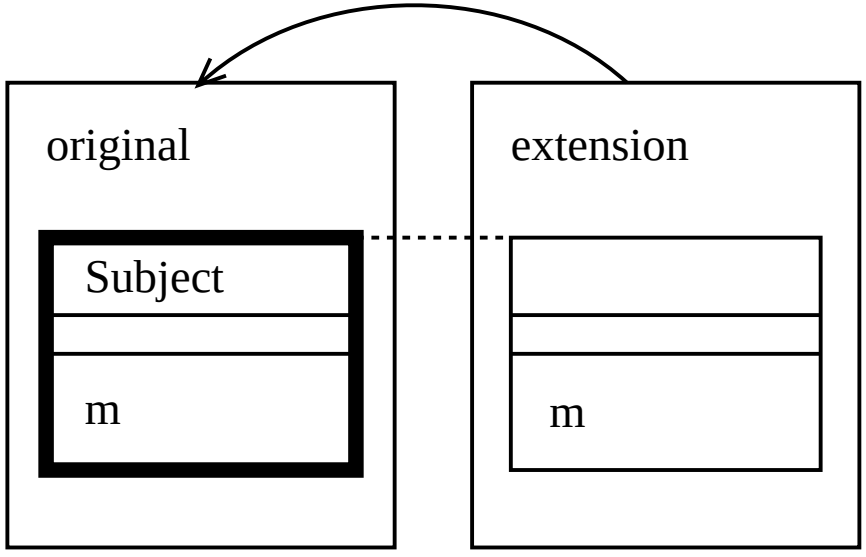
- **23 種すべてについてパターンを静的な部分と動的な部分に分離**
- **静的な部分をモジュール結合によって実装**

MixJuice **でも実装モジュール選択は静的な性質を利用する**

メソッド拡張 - Observer

```
module original {
   define class C {
      define void m() {...}
   }
}
module extension
      extends original {
   class C {
      void m() {
         original();
         イベント通知
} } }
```

メソッド拡張 - Observer



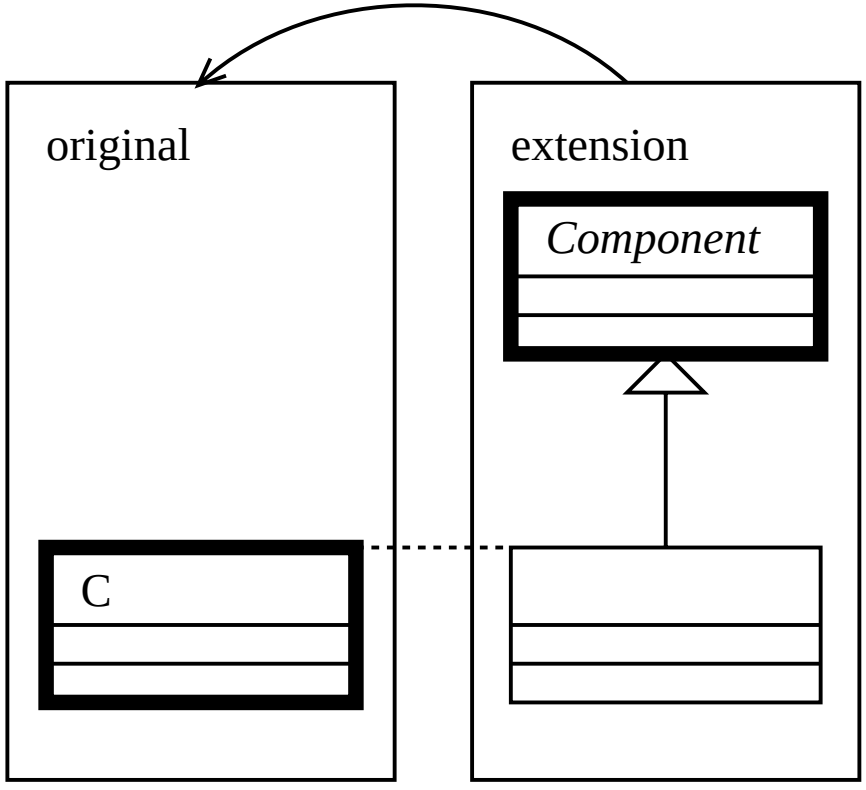
メソッド追加 - Chain of Responsibility

```
module original {
  define abstract class C {...}
  define class A extends C {...}
  define class B extends C {...}
}
module extension extends original {
  class Handler {
    define abstract void m();
  }
  class A { void m() {...} }
  class B { void m() {...} }
}
```

スーパーインターフェース追加 - Compoiste

```
module original {
   define class C {...}
}
module extension
      extends original {
   define interface Component {
      ...
   }
   class C implements Component {
      ...
   }
}
```

スーパーインターフェース追加 - Compoiste



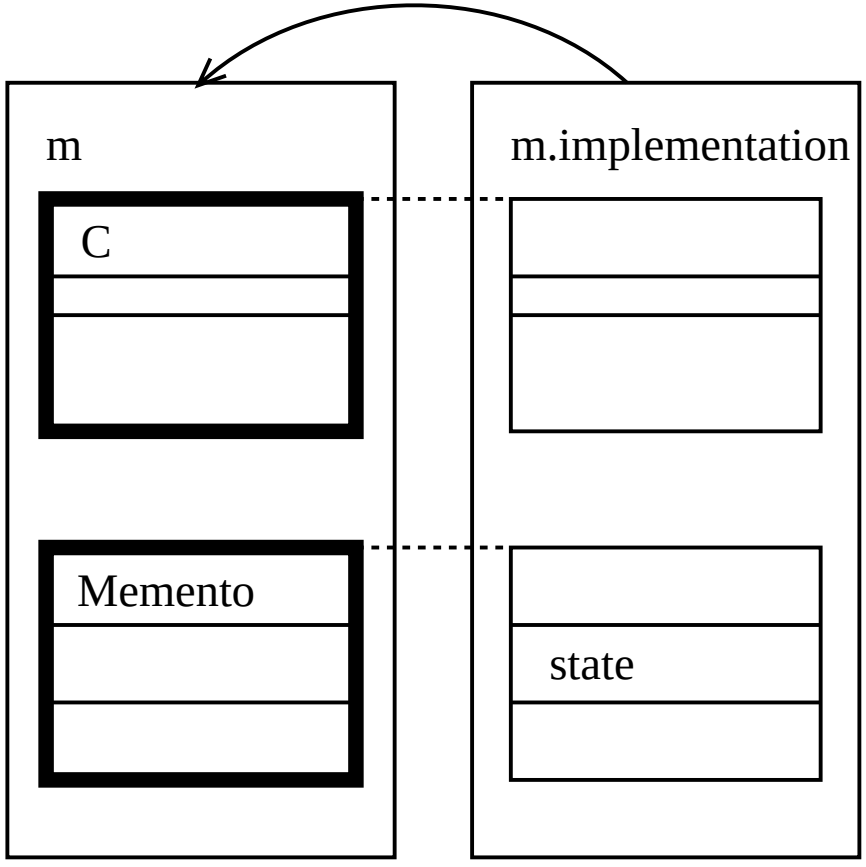
名前空間分離 - Memento

```
module m {
   define class C { ... }
   define class Memento {}
}

module m.implementation
      extends m {
   class C { ... }

   class Memento {
      int state;
   }
}
```

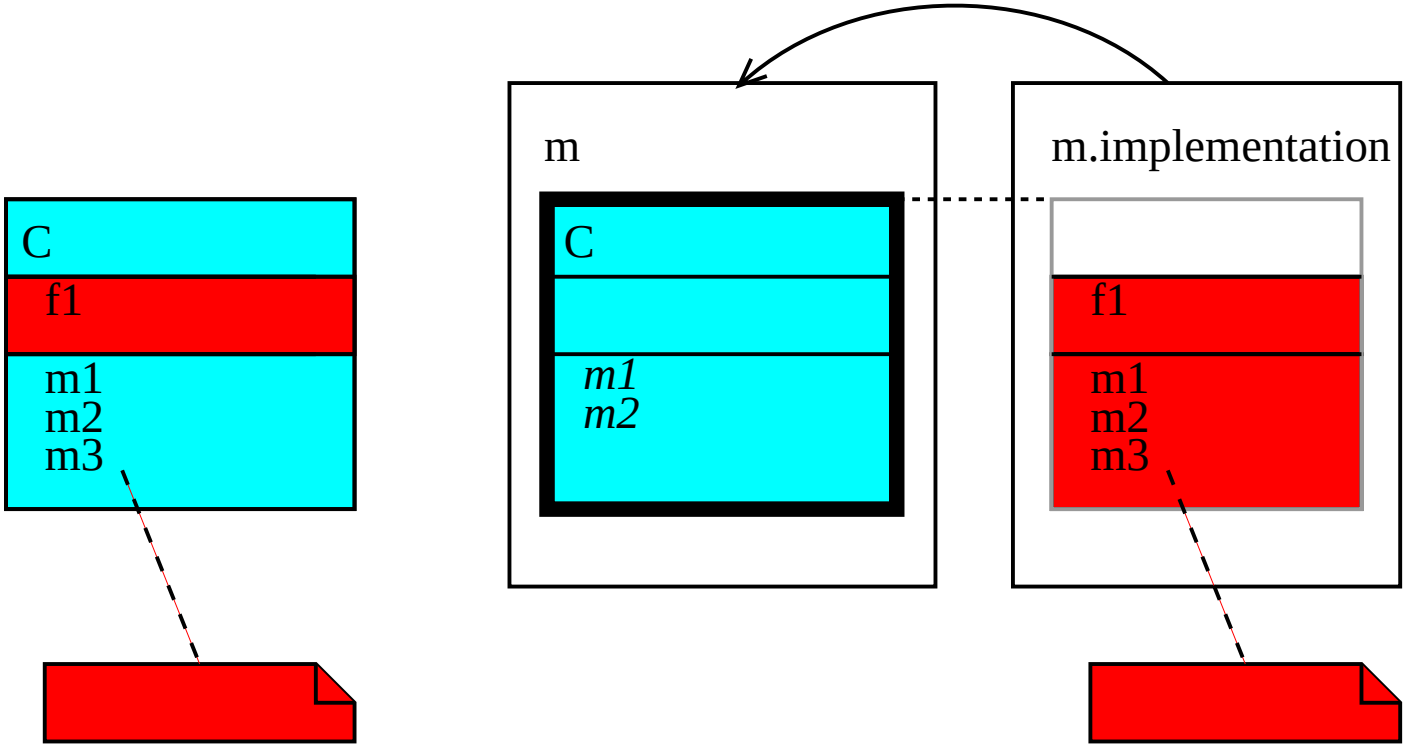
名前空間分離 - Memento



仕様モジュールと実装モジュールの分離 - Visitor

```
module m {
  define class ConcreteElement {
    // public methods
    define abstract void m1();
    define abstract void m2();
  }
}
module m.implementation extends c
  class ConcreteElement {
    // public methods
    void m1() {...}
    void m2() {...}
    // protected fields and metho
    int f1;
```

仕様モジュールと実装モジュールの分離 - Visitor



実装モジュール選択 - Strategy

```
module m {
 define class C {
  define abstract void strategy()
 }
}
module m.strategyA extends m {
 class C { void strategy(){...}}
}
module m.strategyB extends m {
 class C { void strategy(){...}}
}
```

実装モジュール選択 - Strategy