オブジェクト指向技術 第14回
― オブジェクト指向の実践 ―

立命館大学 情報理工学部
丸山 勝久

maru@cs.ritsumei.ac.jp

講義内容

- ソフトウェアパターン
- デザインパターン
  - 第13回から続く
- リファクタリング

GoFのデザインパターン

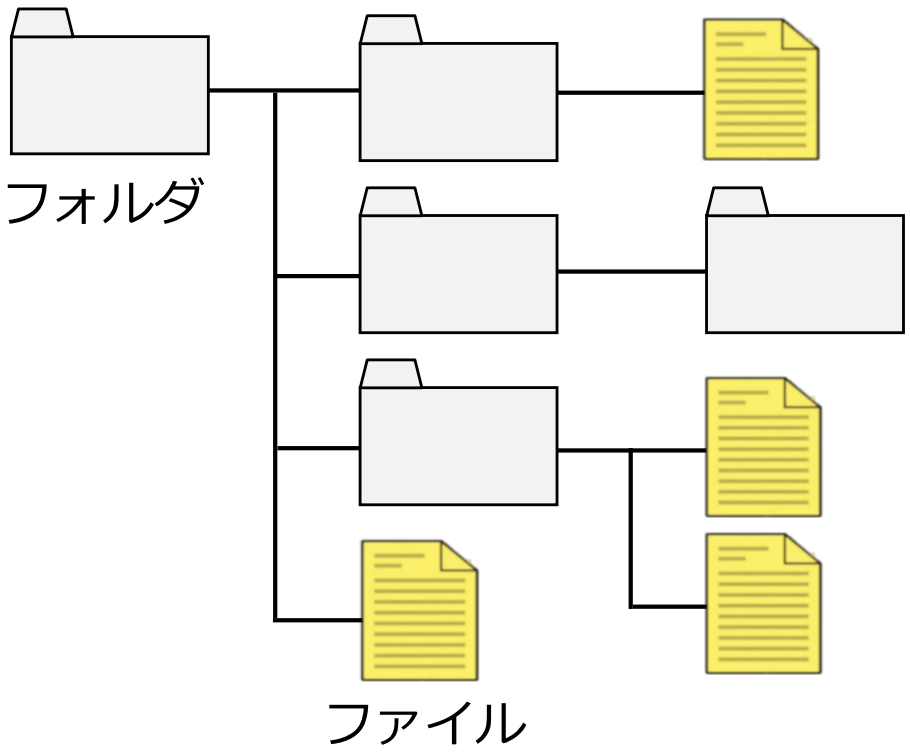
生成に関する	Abstract Factory	関連する部品を組み合わせて製品を作る
	Builder	複雑なインスタンスを組み立てる
	Factory Method	インスタンス生成をサブクラスに任せる
	Prototype	コピーしてインスタンスを作る
	Singleton	たった1つのインスタンス
構造に関する	Adapter	一皮かぶせて再利用
	Bridge	機能の階層と実装の階層を分ける
	Composite	容器と中身の同一視
	Decorator	飾り枠と中身の同一視
	Facade	シンプルな窓口
	Flyweight	同じものを共有して無駄をなくす
	Proxy	必要になってから作る

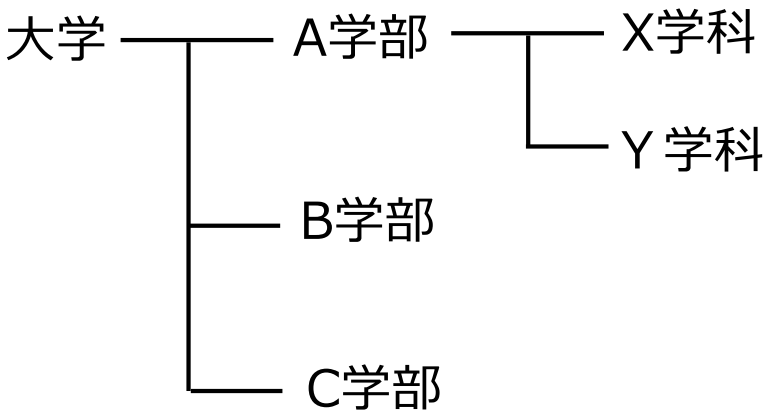
振る舞いに関する	Chain of responsibility	責任のたらい回し
	Command	命令をクラスにする
	Interpreter	文法規則をクラスで表現
	Iterator	1つ1つ数え上げる
	Mediator	相手は相談役1人だけ
	Memento	状態を保存する
	Observer	状態の変化を通知
	State	状態をクラスとして表現
	Strategy	アルゴリズムを切り替える
	Template Method	具体的な処理をサブクラスに任せる
	Visitor	構造を渡り歩きながら仕事をする

GoF (Gang of Four) = Erich Gamma, Richard Helm, Ralph Johnson, John Vlissidies

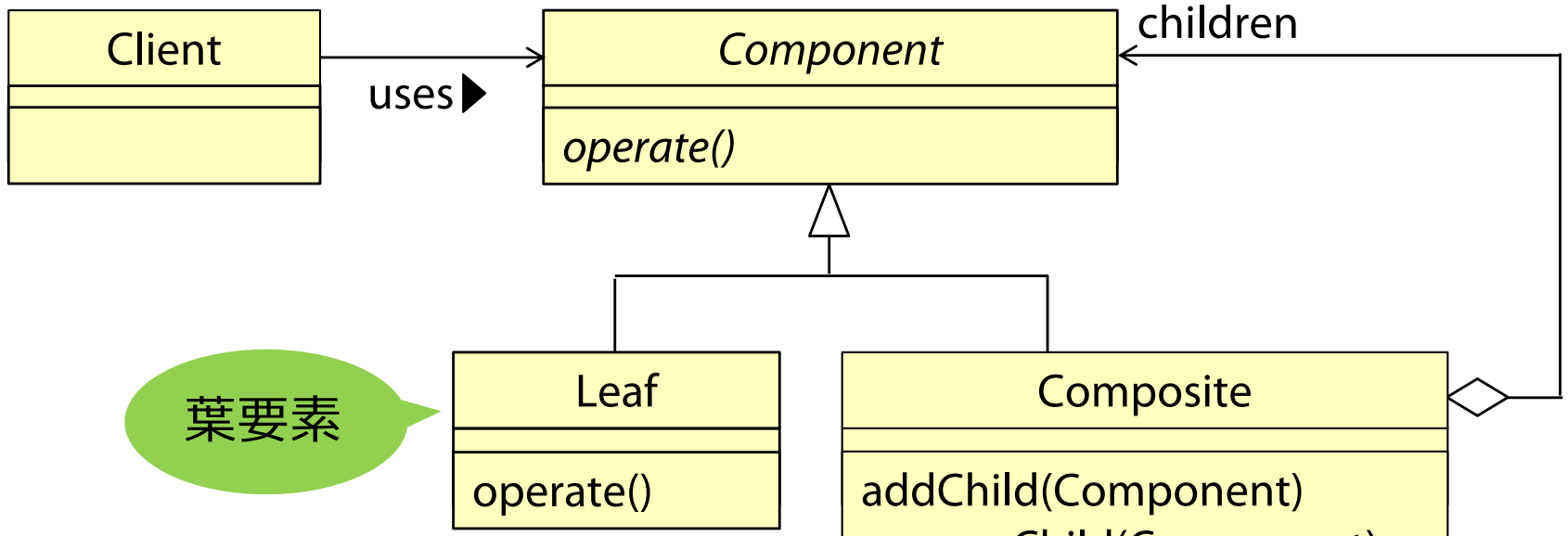
Compositeパターン

- 階層構造(木構造等)を著感的に表現したい
  - ディレクトリ構造，組織構造





Compositeパターンの構造



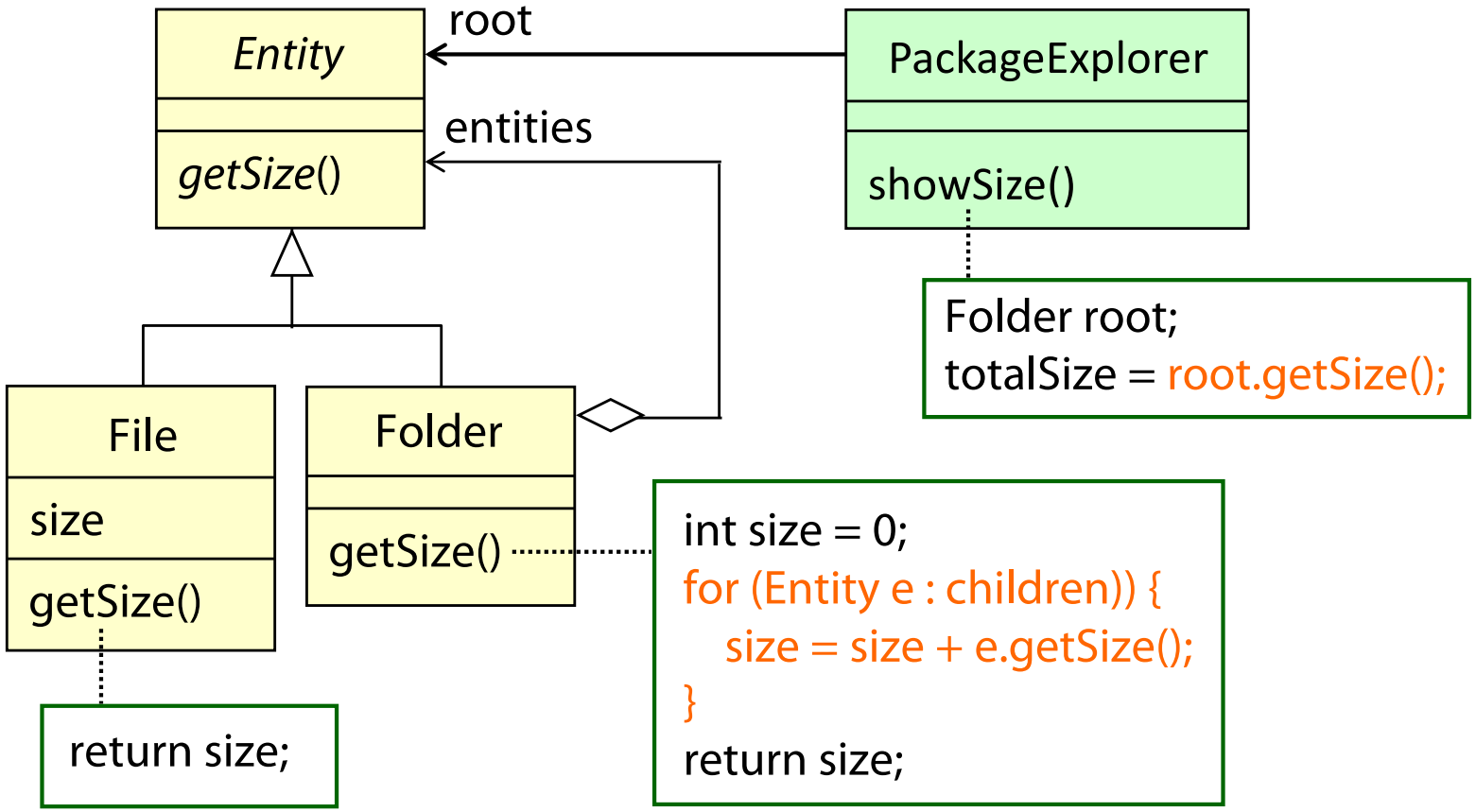
木構造において子供を持たない

木構造において子供を持つ

- ObserverはCompositeとLeafをComponentとして同一視
- addChild()とremoveChild()の対象はComponent
- getChildren()は子要素の集まりを返す

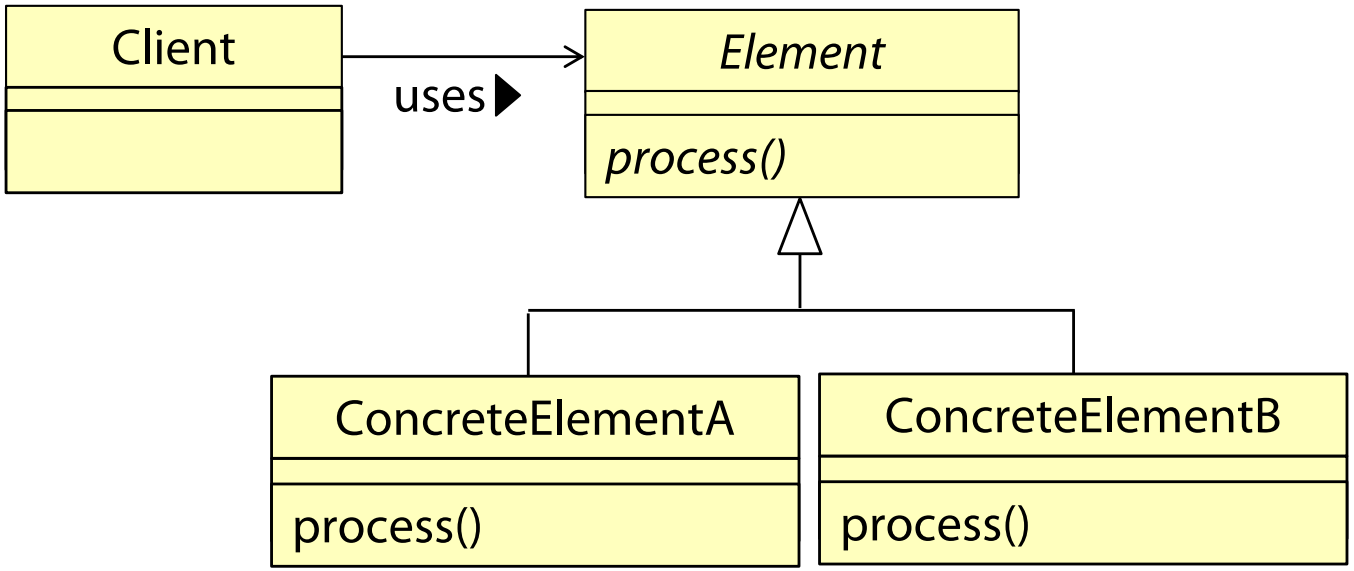
Composite Patternの特徴・注意点

- 様々な再帰構造に適用可能
- LeafかCompositeかに関わらず共通の処理は同じメソッド呼び出しで行うことができる



Visitorパターン

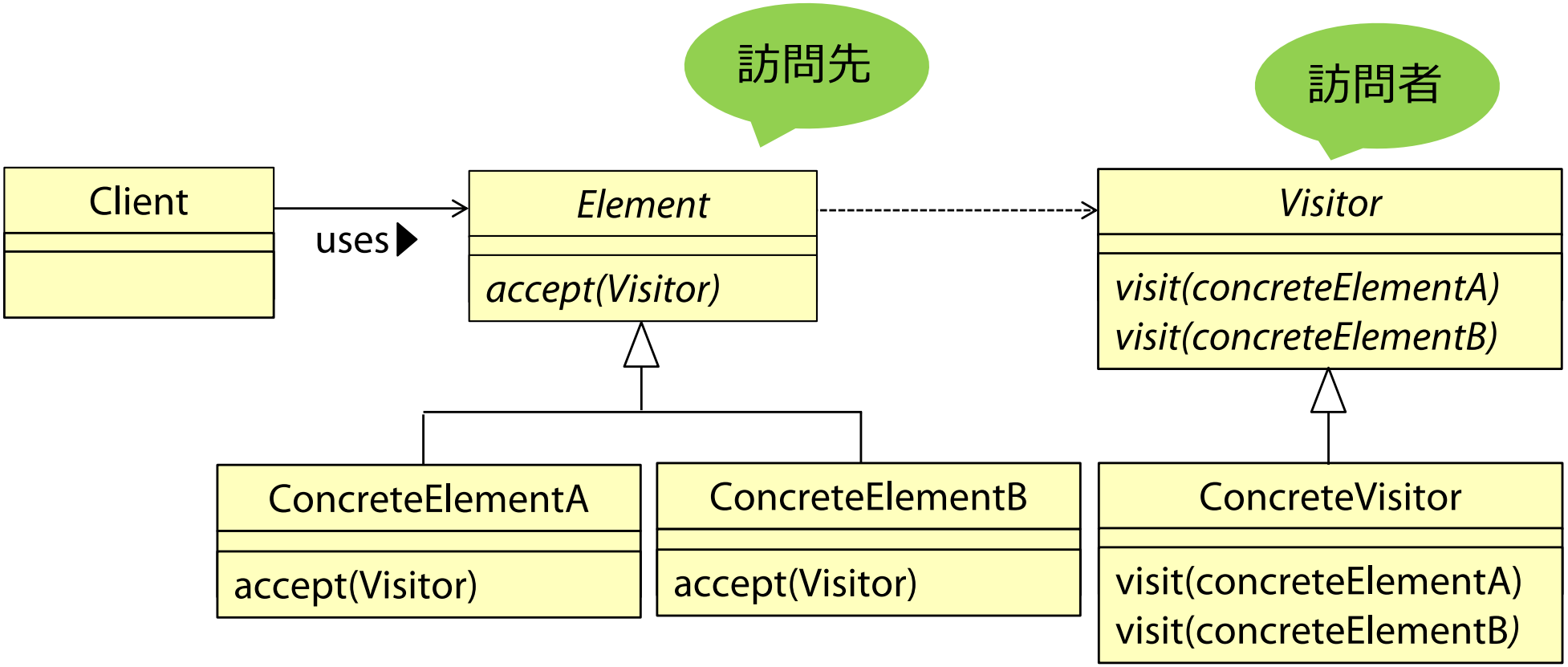
- 複雑なデータ構造の中を渡り歩きながら(走査しながら)
  各要素に応じた処理を行う



Elementにおける処理が分散する

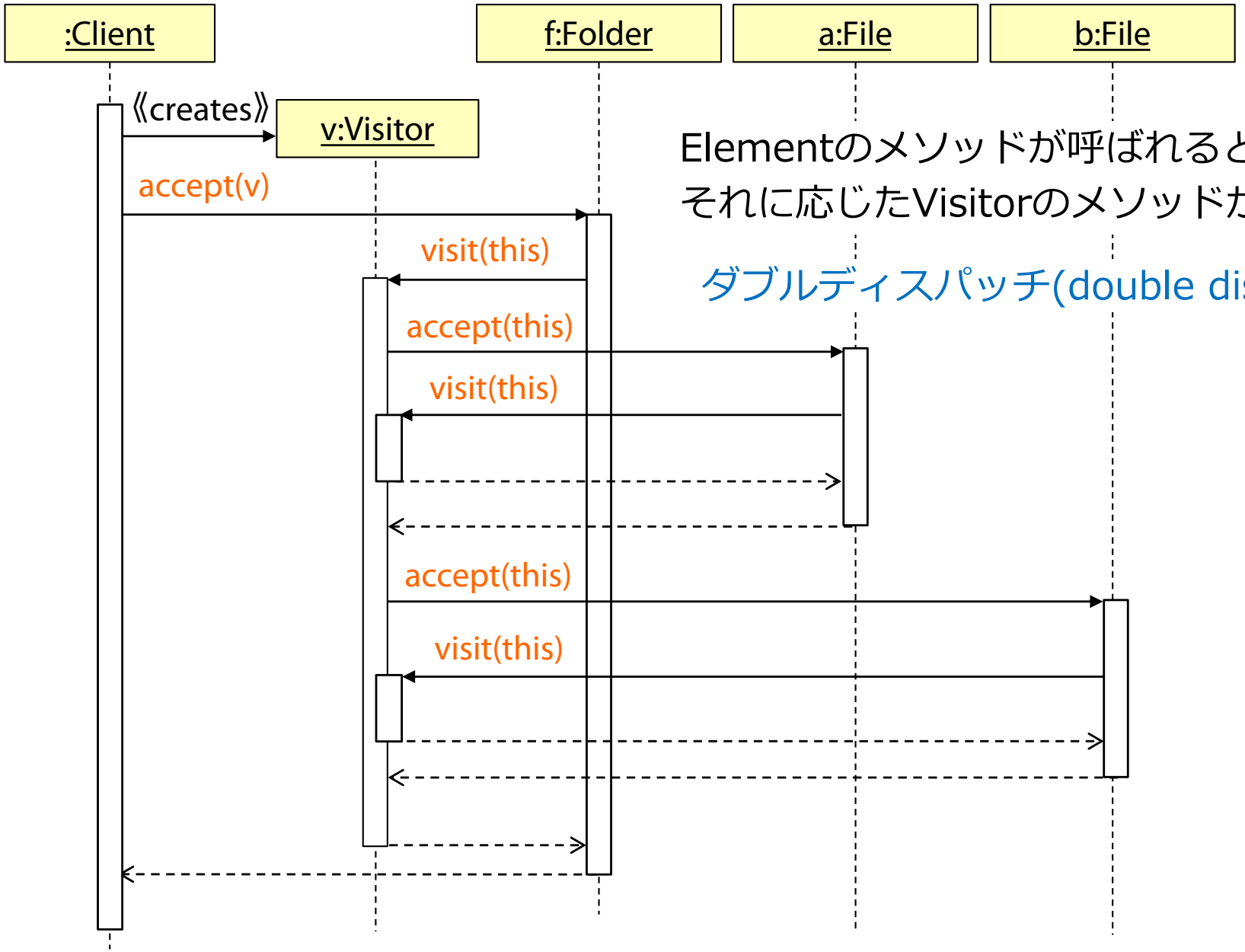
データ構造が複雑な場合，Clientが走査に責任を持つのは面倒

Visitor Patternの構造



- Elementの処理はVisitorのvisit()に集める
  - メソッドオーバーロードを利用
- どのElementをどの順番に走査するのかは，Visitorで定義する

Visitorパターンの振る舞い



Elementのメソッドが呼ばれると，
それに応じたVisitorのメソッドが呼ばれる

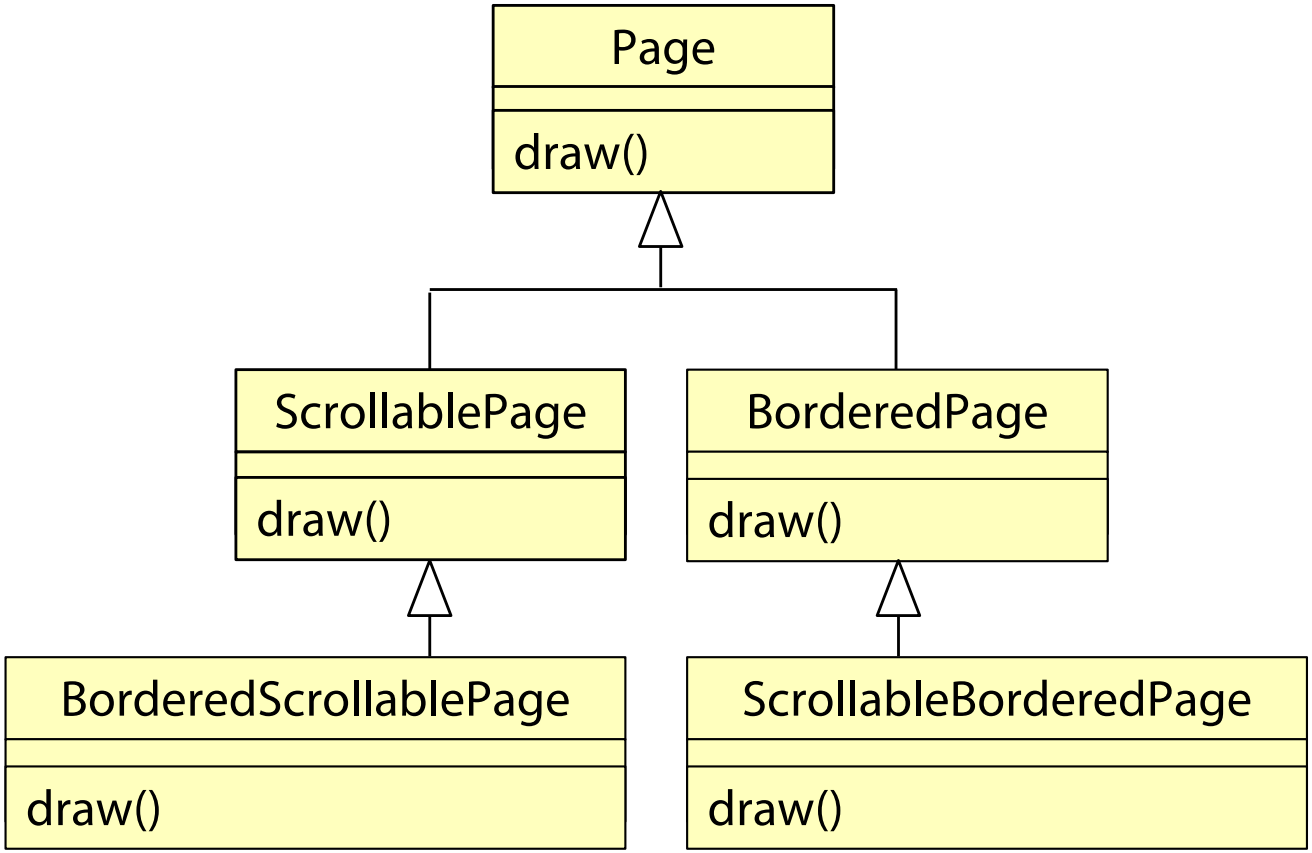
ダブルディスパッチ(double dispatch)

Visitorパターンの特徴・注意点

- 訪問時の処理をVisitorに集めることができる
  - データ構造と処理の分離
- 複数のVisitorを用意することで，様々な辿り方を切り替えが可能
  - 木構造だと行きがけ順 pre-order, 通りがけ順 in-order, 帰りがけ順 post-order 等
  - 条件に応じて，次に辿るElementをきめ細かく制御可能
- Elementを修正せずに，Elementに対する処理を拡張可能
  - ConcreteVisitorの追加は容易
  - ConcleteElementの追加は面倒
  - 通常，データに対する処理内容よりもデータ構造の方が固定的

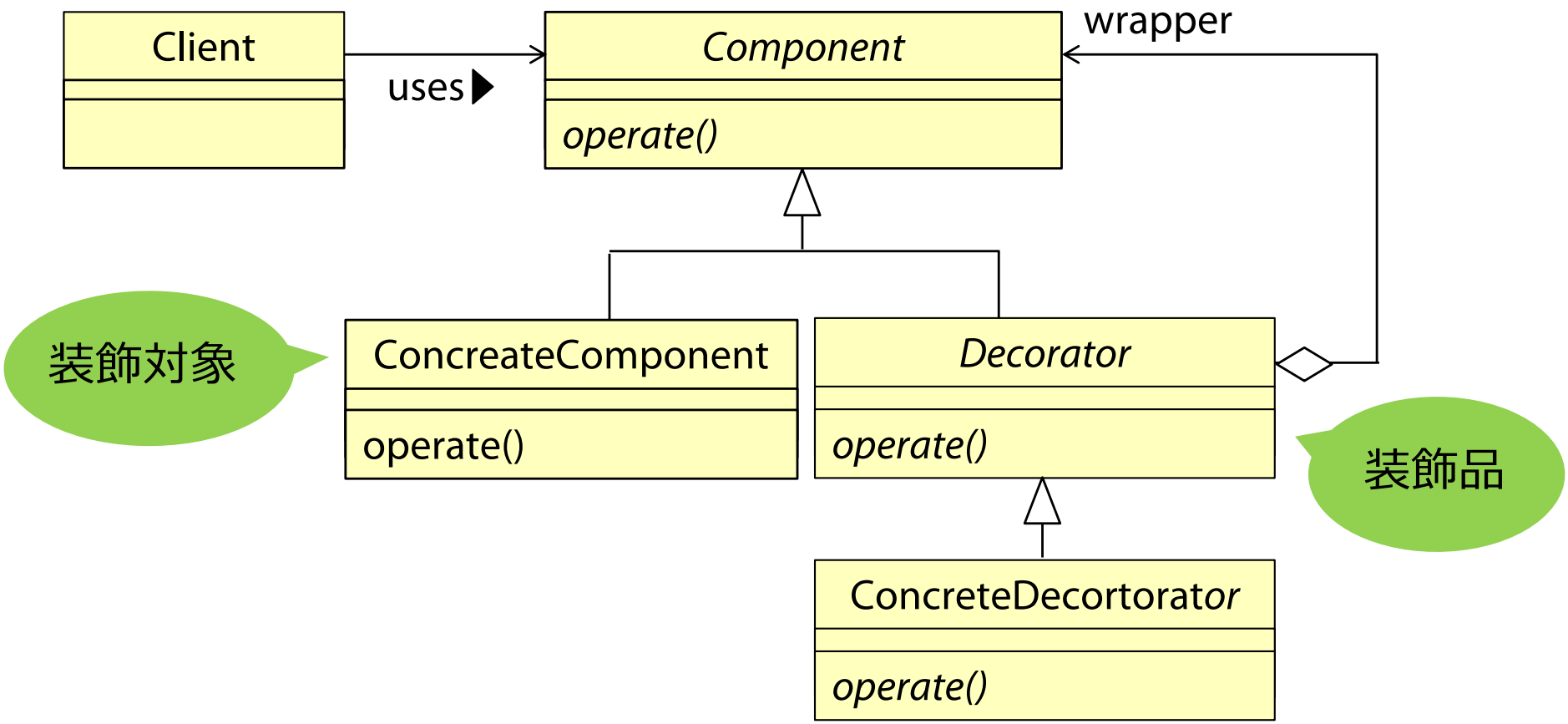
Decoratorパターン

- 様々な責任を動的に付与したい



継承を利用して静的に機能拡張を行うと
継承階層が不格好になる

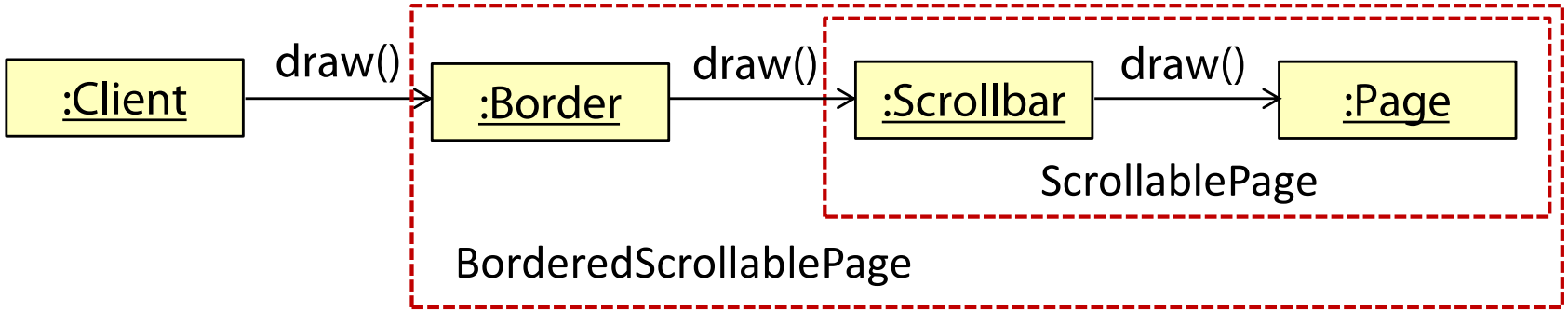
Decoratorパターンの構造



- Componentは装飾されるConcreteComponentと
  装飾品であるDecoratorが共通のインタフェースを提供する

Decoratorパターンの特徴・注意点

- 拡張したい機能(責任)を動的に付与できる
  - 装飾による機能拡張
  - Clientから，装飾されたインスタンスはすべてComponentに見える

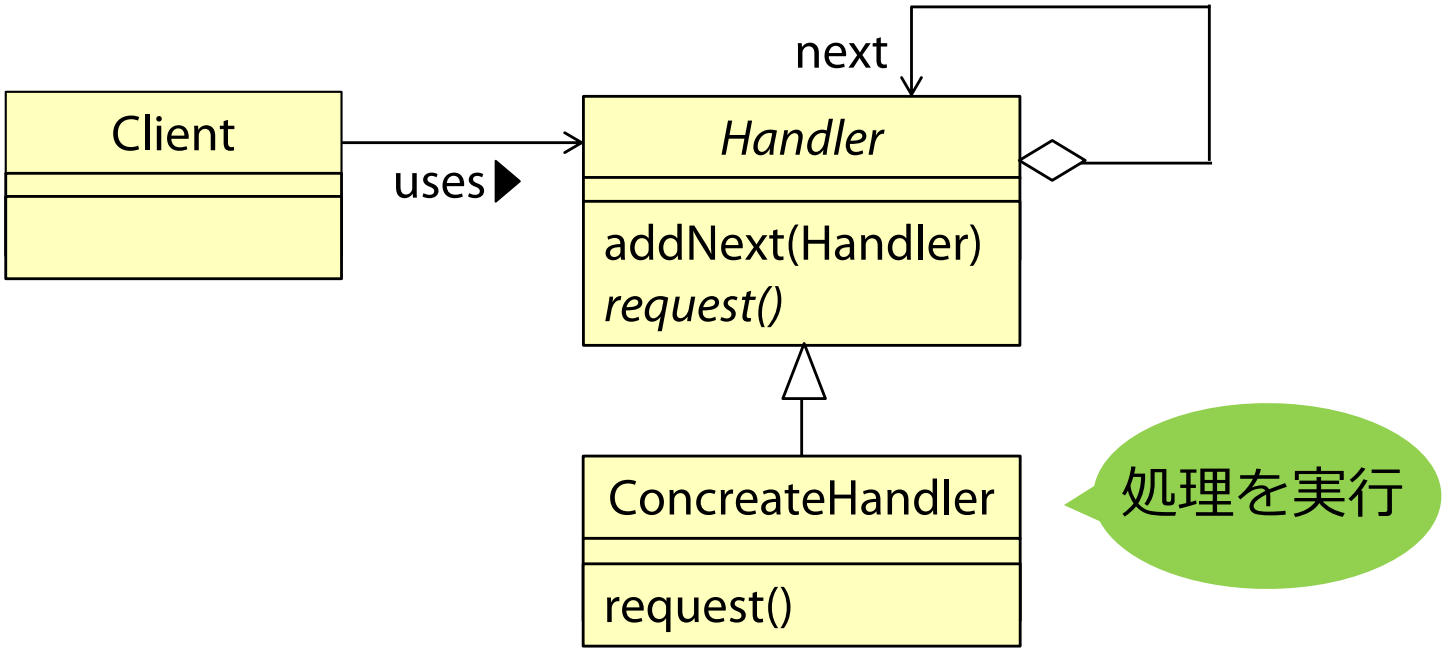


- 装飾品を装飾対象から切り離して実装できる
- 装飾品は，積み重ねの順番に依存しないように実装しなければならない
- 装飾品インスタンスの生成が多段になり面倒

```
BufferedReader reader = new BufferedReader(new FileReader(filepath)));
```

Chain of Responsibility(CoR)パターン

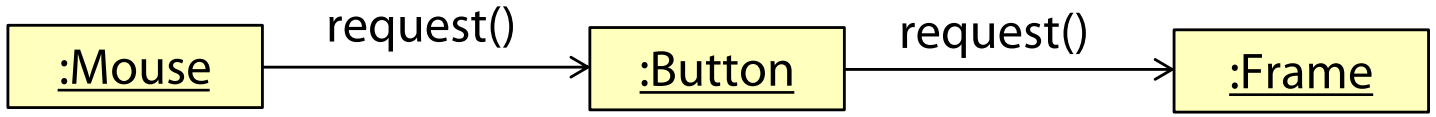
- 複数のオブジェクトに要求を処理する機会を与えたい



- Handlerは，要求に対して自分で処理できるなら処理し，できないならnextに転送する
- Clientは先頭のハンドラに要求を送信

CoRの特徴・注意点

- 連鎖(chain)の形態を定義
  - 要求の処理に関する順序関係を動的かつ柔軟に変更可能



- 要求を発生させるオブジェクトと要求を処理するオブジェクトの結合を弱める
  - Clientは実際にどのHandlerで要求が処理されるか知らなくて良
  - 要求の送信元が要求の送信先を探す手間が不要
- 直接依頼する方が実行が早い
- 要求が処理されない可能性がある
  - 処理が実行されないときの対応を考えておく

リファクタリング (refactoring)

- 既存のソフトウェアの外部的な振る舞いを変えることなく，設計や実装を改善する作業
  - コードの理解性や保守性の向上を目的
- リストラクチャリングの一種
  - コード変換の目的や内容がカタログとしてまとめられている

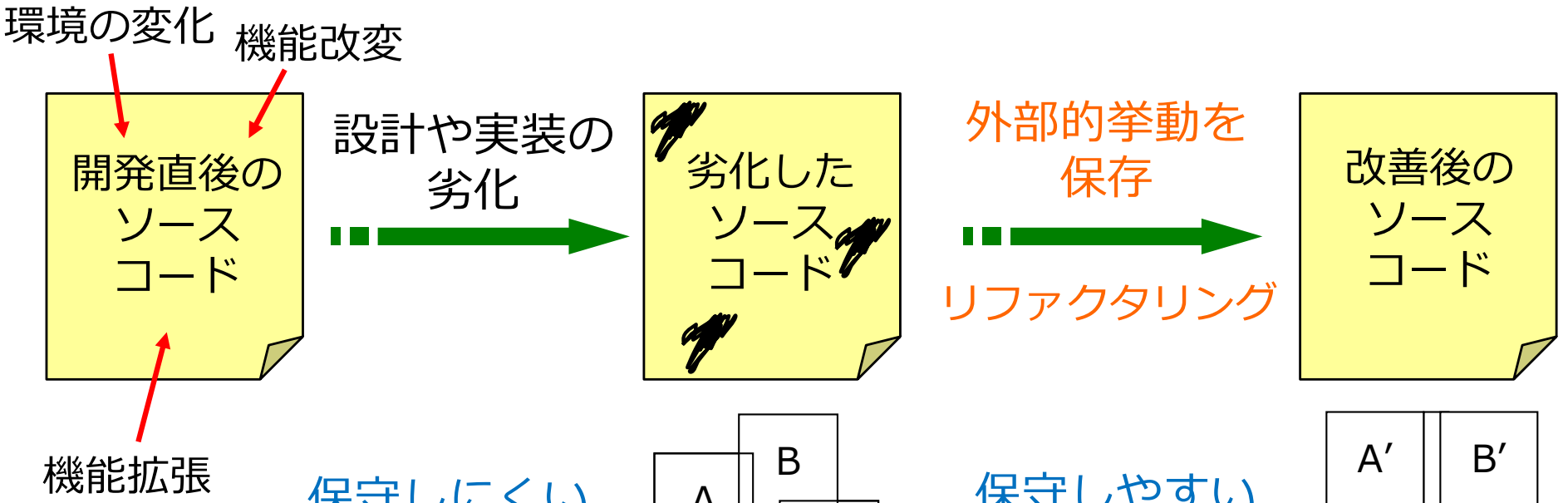
re = 繰り返し

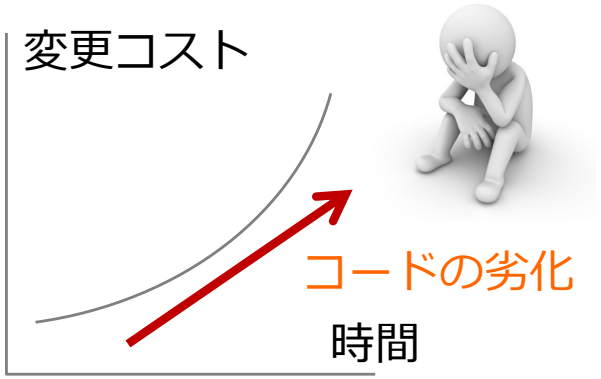
factoring = 因数分解

$$f_1(x) = x^2 + 3x + 2$$
$$f_2(x) = (x + 2)(x + 1)$$
$$f_3(x) = ((x + 1) + 1)(x + 1)$$
$$f_4(x) = (x + 1)^2 + (x + 1)$$

$$\forall x\, f_1(x) = f_2(x)$$
$$\forall x\, f_2(x) = f_3(x)$$
$$\forall x\, f_3(x) = f_4(x)$$

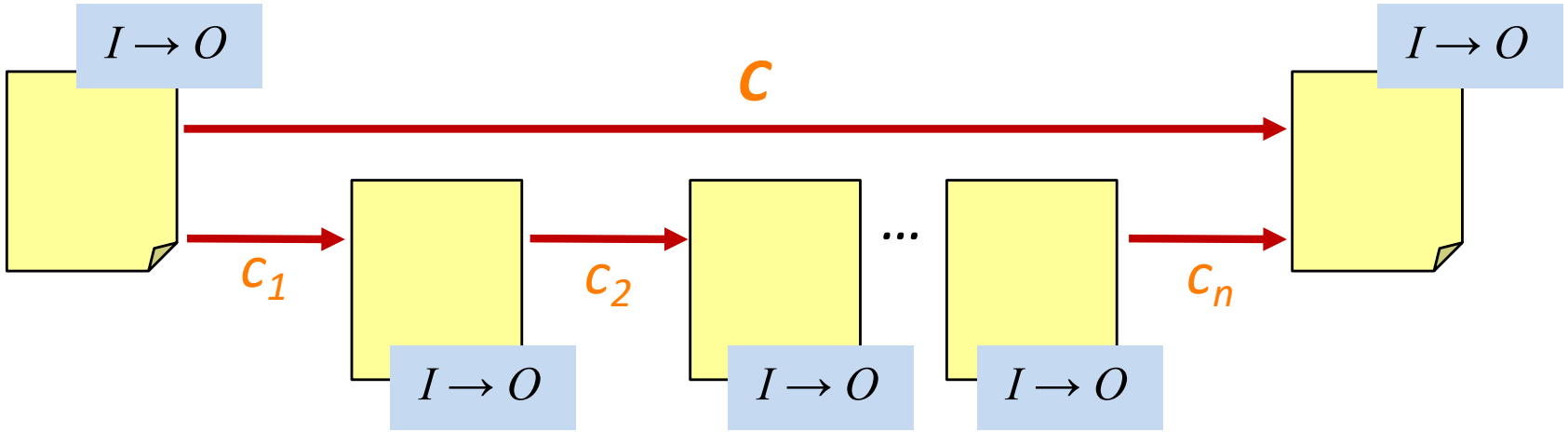
リファクタリングの概要





リファクタリングの安全性(1/2)

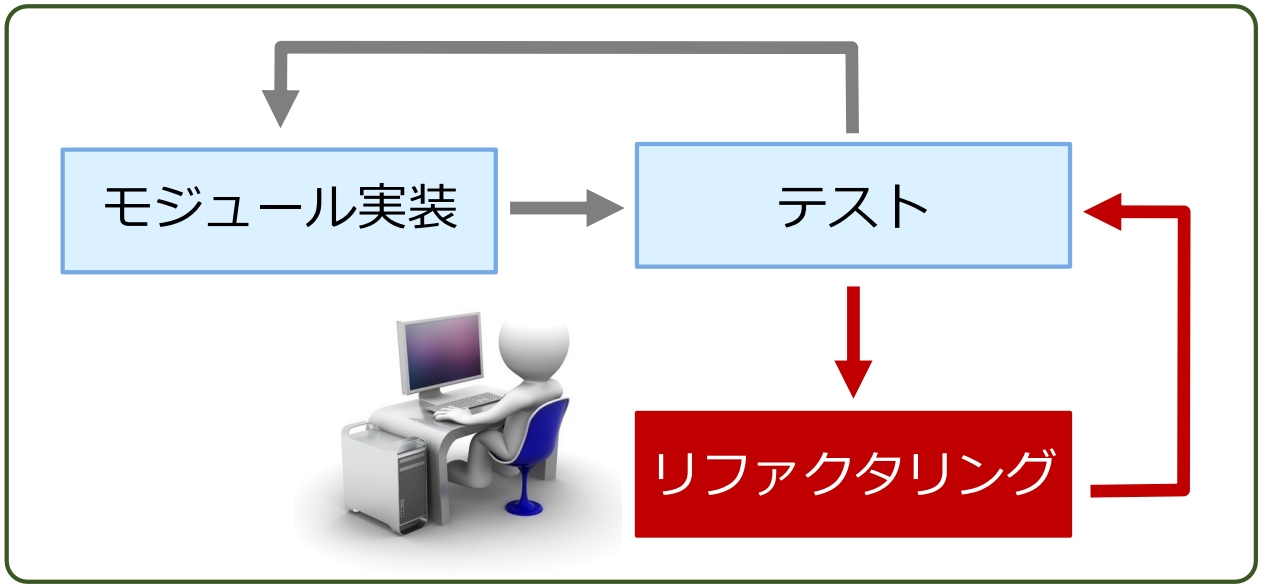
- 同一の入力値集合に対して，リファクタリング前後で出力値集合が同一
  - 外部的振る舞い ≒ 観測できる入出力の関係
- 大きな設計変更を小さなコード変換の繰り返しで実現する

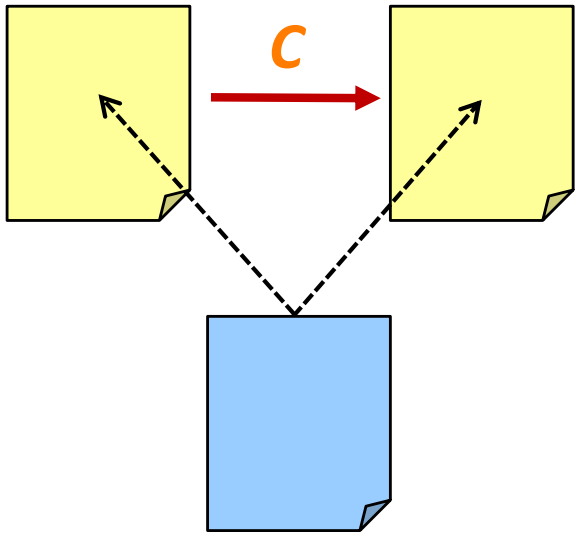


$c_1, c_2, \ldots, c_n$ を外部的振る舞いを保存するコード変換とすると，
C のコード変換は外部的振る舞いを保存

リファクタリングの安全性(2/2)

- テスト結果が変わらなければ，外部的振る舞いを保存しているとみなす





同一の
テストケースを利用

十分なテストを実施することで変換前後の
外部的振る舞いの保存を保証

注意

$f_1(x) = x(x+1)$ $f_1(0) = 0$

$f_2(x) = x(x+2)$ $f_2(0) = 0$

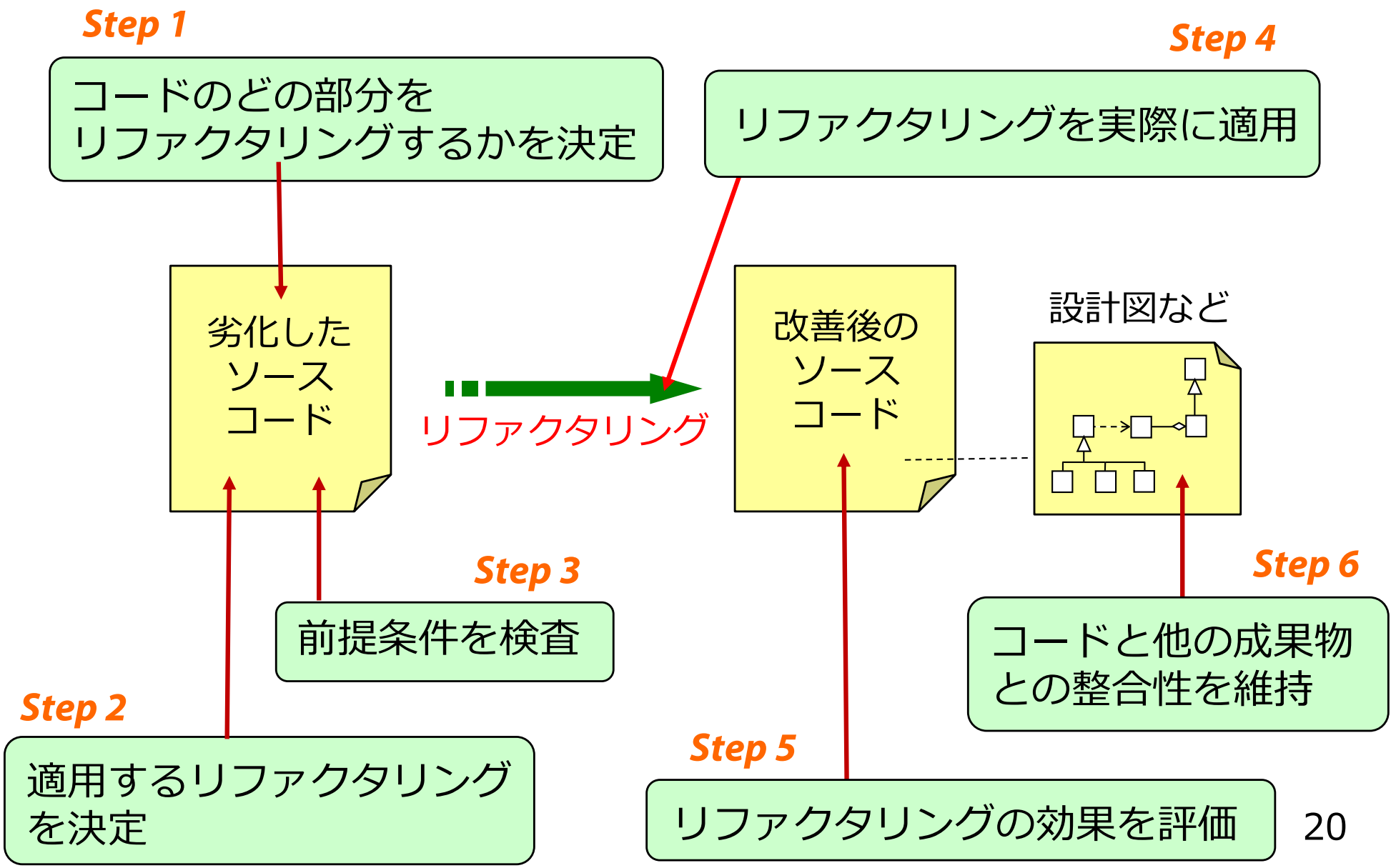
テストが
十分でないと

リファクタリングの手順

- プログラムの作成
- テストの作成
  - 振る舞いの保存を保証するために必須
- テストの実施
- リファクタリングの適用
- テストの実施
  - テストに失敗すればリファクタリング適用の取り消し等の対応を行う
  - 繰り返しテストを実施するため，テストの自動化が必須

Test-Driven開発(TDD)では，
プログラムを作成する前にテストを作成

リファクタリングプロセス



リファクタリングの原則

- いつリファクタリングを適用すべきか
  - 新しい機能を追加するとき
  - バグを修正するとき
  - コードレビューを実施しているとき
  - 機能が完成したとき
- リファクタリング中に機能拡張やバグ修正はしない
  - リファクタリング中に外部的挙動を変化させないため

なぜリファクタリングをするのか

- 保守作業や改良により設計が劣化
  - ソフトウェアは本質的に変化(修正，変更)し続ける
  - 保守性の観点
    - 見慣れない設計や難解なコードを理解および変更しやすく
    - 対象コードに関する理解を確認するために，同じ動作で内容の異なるコードを見ることが可能
- 将来のクラスやクラス間の関係を完全には予測不可能
  - はじめから完全なソフトウェアを構築するのは困難
  - 生産性の観点
    - 設計作業を適当な時期に打ち切り，実装作業に移行可能
    - 既存コードから再利用モジュールの抽出可能

不吉な臭い

- 既存のコードにおいて問題を発見するための手がかり
- 潜在的な問題や欠陥に関する兆候(warning sign)
  - 問題かどうかは不明であるが，調べてみる価値があるもの
  - リファクタリングをいつ適用すべきかの明確な基準はない
- 不吉な臭いの例
  - 不可思議な名前
  - 重複したコード
  - 長いメソッド
  - 長い引数リスト
  - グローバルなデータ
  - 変更可能なデータ
  - 変更の偏り
  - 変更の分散
  - 属性・操作の横恋慕
  - データの群れ
  - 基本データ型への執着
  - 重複したスイッチ文
  - 疑わしき一般化
  - 巨大なクラス
  - 相続拒否
  - コメント

リファクタリングカタログ

- 名前
  - リファクタリングを表現する短く，適切な名前
  - 設計者や開発者の語彙になる
- スケッチ (sketch)
  - リファクタリングの内容を思い出すために使う
  - 適用前後のコードの変化を示す
- 動機
  - 適用すべき理由、適用を避けるべき状況
- 手順
  - 実施手順の記述
- 例
  - 実際の適用例

リファクタリング操作の例

他 多数あり

メソッドの抽出	メソッドのインライン化	フェーズの分離
変数の抽出	変数のインライン化	変数のカプセル化
変数名の変更	パラメータオブジェクトの導入	メソッド宣言の変更
メソッド群のクラスへの集約	レコードのカプセル化	コレクションのカプセル化
オブジェクトによるプリミティブの置き換え	問い合わせによる一時変数の置き換え	仲介人の除去
クラスの抽出	クラスのインライン化	委譲の隠蔽
アルゴリズムの置き換え	メソッドの移動	フィールドの移動
ループの分離	デッドコードの削除	変数の分離
フィールド名の変更	参照から値への変更	値から参照への変更
条件記述の分解	条件記述の統合	特殊ケースの導入
ガード節による入れ子の条件記述の置き換え	ポリモーフィズムによる条件記述の置き換え	アサーションの導入
問い合わせと更新の分離	パラメータによるメソッド統合	フラグパラメータの削除
setterの削除	メソッドの引き上げ	メソッドの押し下げ

メソッドの抽出

Extract Method

```
class Customer {
  public String statement() {
    String results = “Rental Record: ”;
    double totalAmount = 0;
    for (Rental rental : rentals) {
      double thisAmount = 0;
      switch ( … ) {
        …
      }
      totalAmount += thisAmount;
    }
    results += ….
}
```

リファクタリング前

```
class Customer {
  public String statement() {
    String results = “Rental Record: ”;
    double totalAmount = 0;
    for (Rental rental : rentals) {
      double thisAmount = amountFor(rental);
      totalAmount += thisAmount;
    }
    results += ….
  }
  private double amountFor(Rental rental) {
    double thisAmount = 0;
    switch ( … ) {
      …
    }
    return thisAmount;
  }
}
```

リファクタリング後

メソッドのインライン化

Inline Method

```
class Customer {
  public String statement() {
    String results = “Rental Record: ”;
    double totalAmount = 0;
    for (Rental rental : rentals) {
      double thisAmount = amountFor(rental);
      totalAmount += thisAmount;
    }
    results += ….
  }
  private double amountFor(Rental rental) {
    double thisAmount = 0;
    switch ( … ) {
      …
    }
    return thisAmount;
  }
}
```

リファクタリング前

```
class Customer {
  public String statement() {
    String results = “Rental Record: ”;
    double totalAmount = 0;
    for (Rental rental : rentals) {
      double thisAmount = 0;
      switch ( … ) {
        …
      }
      totalAmount += thisAmount;
    }
    results += ….
}
```

リファクタリング後

メソッドの移動

Move Method

```
class Customer {
  public String statement() {
    …
    double thisAmount = amountFor(rental);
    …
  }
  private double amountFor(Rental rental) {
    double thisAmount = 0;
    switch (rental.getMovie().getPriceCode()) {
     case Movie.REGULAR:
      thisAmount = …
      …
  }
}
```

```
class Rental {
 …
}
```

リファクタリング前

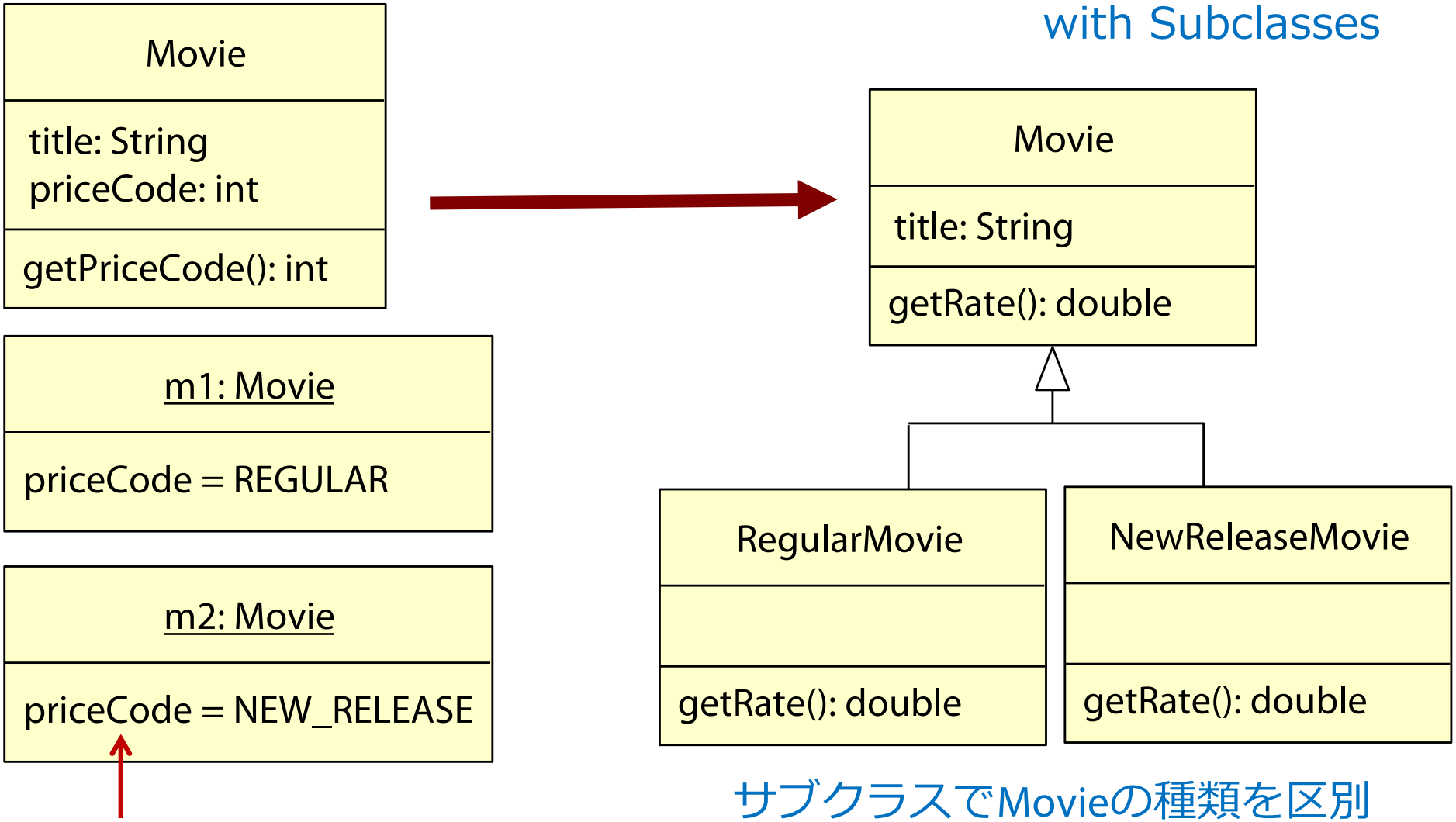
```
class Customer {
  public void statement() {
    …
    double thisAmount = amountFor(rental);
    …
  }
  private double amountFor(Rental rental) {
     return rental.amountFor();
  }
}
```

```
class Rental {
  public double amountFor() {
    double thisAmount = 0;
    switch (getMovie().getPriceCode()) {
    …
  }
}
```

リファクタリング後

タイプコードの置き換え

Replace Type Code
with Subclasses



タイプコード(type code)で
Movieの種類を区別

サブクラスでMovieの種類を区別

ポリモーフィズムによる条件記述の置き換え

Replace Conditional with Polymorphism

```
class Rental {
  Movie movie;

  public double amountFor() {
    thisAmount = 0;
    switch (movie.getPriceCode()) {
      case Movie.REGULAR:
        thisAmount = daysRented;
        break;
      case Movie.NEW_RELEASE:
        thisAmount = daysRented * 3;
        break;
    }
  }
}
```

Replace Type Code with Subclasses

```
class Rental {
  Movie movie;
  public double amountFor() {
    thisAmount = daysRented * movie.getRate();
  }
}
```

動的束縛

```
abstract class Movie {
  abstract double getRate();
}
```

```
class RegularMovie extends Movie {
  double getRate() { return 1; }
}
```

```
class NewReleaseMovie extends Movie {
  double getRate() { return 3; }
}
```

リファクタリングの実践

- 個々の変換手順を覚える必要はない
  - 変換はカタログを見ながら実施すればよい
  - リファクタリングツールによる自動コード変換
- 不吉な匂いを見つけられるスキルは重要
  - 匂いが見つけられなければリファクタリングは始まらない
  - デザインパターンとの関係を考えるのが有効
    - パターン指向リファクタリング
- リファクタリングに過度に期待しない
  - 最初から作り直した方が良い場合もあり
    - アーキテクチャの悪臭は挙動を保存するコード変換だけでは除去できない
  - リファクタリングを生かすためには，適度な初期設計が必要
    - プレファクタリング(pre-factoring)

パターン指向リファクタリング

- リファクタリングにより既存の設計を
  ソフトウェアパターンに変換
  - パターンがリファクタリングの到達点を提供
  - パターンは設計の目指す先，リファクタリングとはどこか別の所からそこへ到達する方法
- 27個のリファクタリング
  - 12個の不吉な匂い
  - リファクタリングを繰り返し適用することで，
    - パターンに置き換える(To)
    - パターンに近づく(Towards)
    - パターンから離れる(Away from)

まとめ(1/2)

- Compositeパターンでは，葉要素とそれ以外を同一視することで，階層構造を直感的に表現する
- Visitorパターンでは，複雑なデータ構造の中を走査しながら各要素に応じた処理を行う
- Decoratorパターンでは，様々な責任を動的に付与する
- Chain of Responsibility(CoR)パターンでは，要求を処理するためのオブジェクトを鎖状に連結し、
  複数のオブジェクトに処理の機会を与える

まとめ(2/2)

- リファクタリングとは，既存のソフトウェアの外部的な振る舞いを変えることなく，設計や実装を改善する作業である
- リファクタリングは，リストラクチャリングの一種である
- リファクタリングでは，大きな設計変更を小さなコード変換の繰り返しで実現することで外部的振る舞いの保存を保証
- リファクタリングにおいては，テストの実行結果で外部的振る舞いが保存されていることを確認するのが一般的である
- リファクタリングでは繰り返しテストを実施するため，テストの自動化が必須である
- 不吉な臭いとは，既存のコードにおいて問題を発見するための手がかりや潜在的な問題や欠陥に関する兆候である
- リファクタリングをするのではなく，最初から作り直した方が良い場合がある
- パターン指向リファクタリングでは，パターンがリファクタリングの到達点を提供する