オブジェクト指向技術 第13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

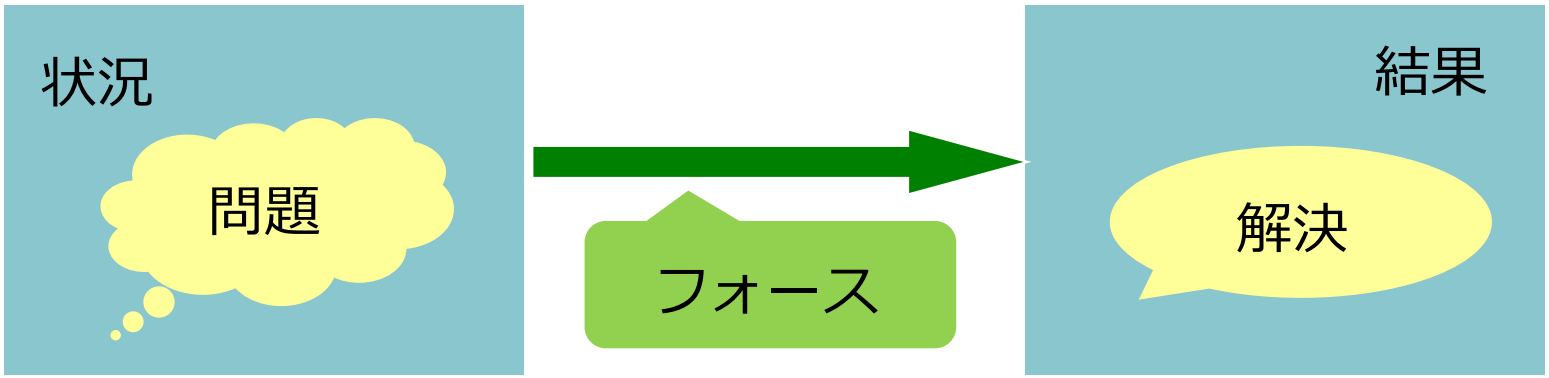
- ソフトウェアパターン
- デザインパターン
  - 第14回に続く
- リファクタリング

ソフトウェアパターン(pattern)

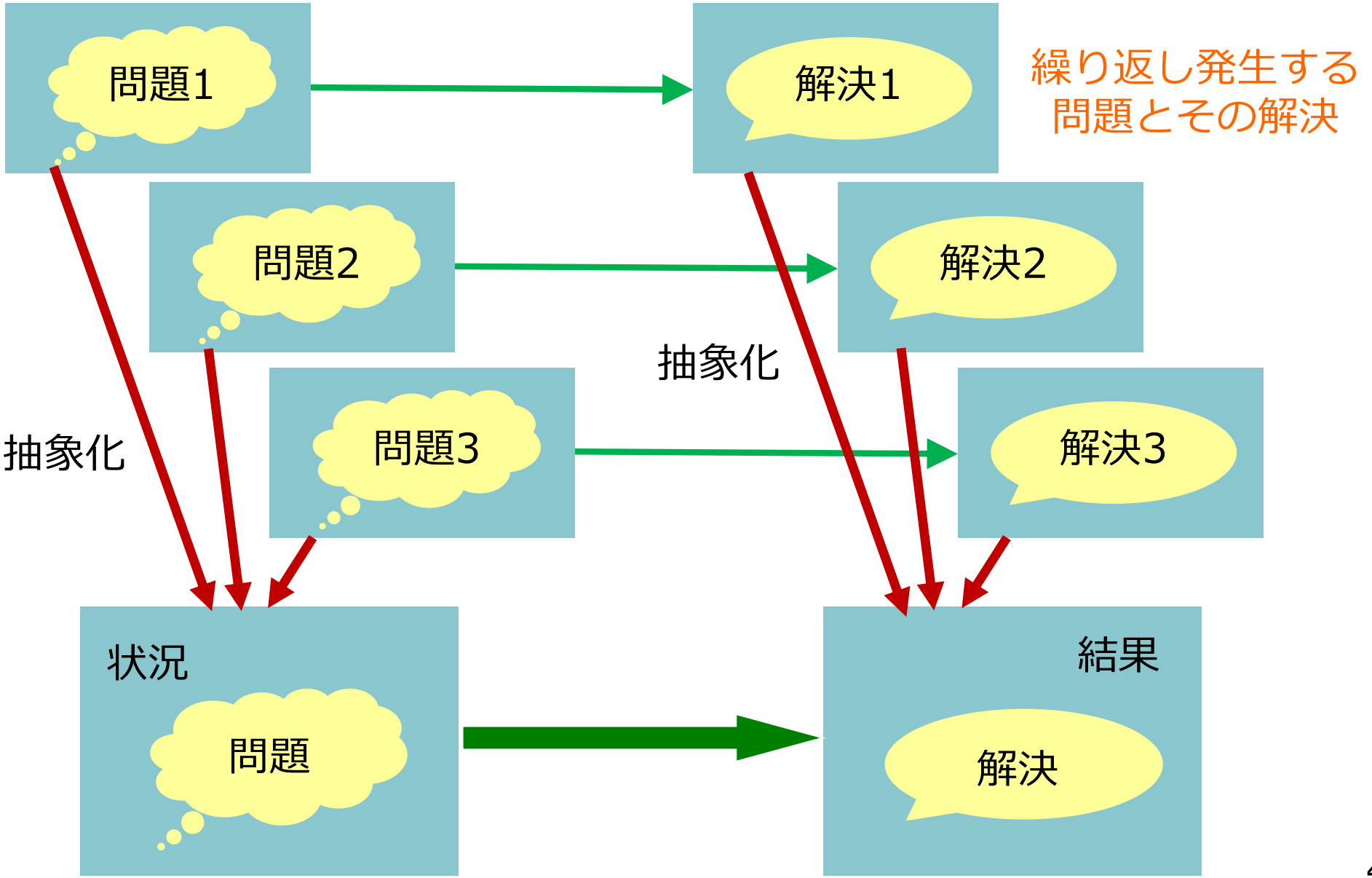
- 開発者の経験や知見を再利用できるように体系化したもの
- ソフトウェア開発において繰り返し現れる問題に対する解法・指針・知見に名前を付与してテンプレート化
  - パターンはそのまま組み込む再利用部品ではない
- ソフトウェア開発の各局面で頻繁に現れる構造や原則
- 熟練開発者たちの過去の成功例
- コミュニケーションを効率的に行うための標準的な語彙
  - パターンは名前をもつ

パターン記述

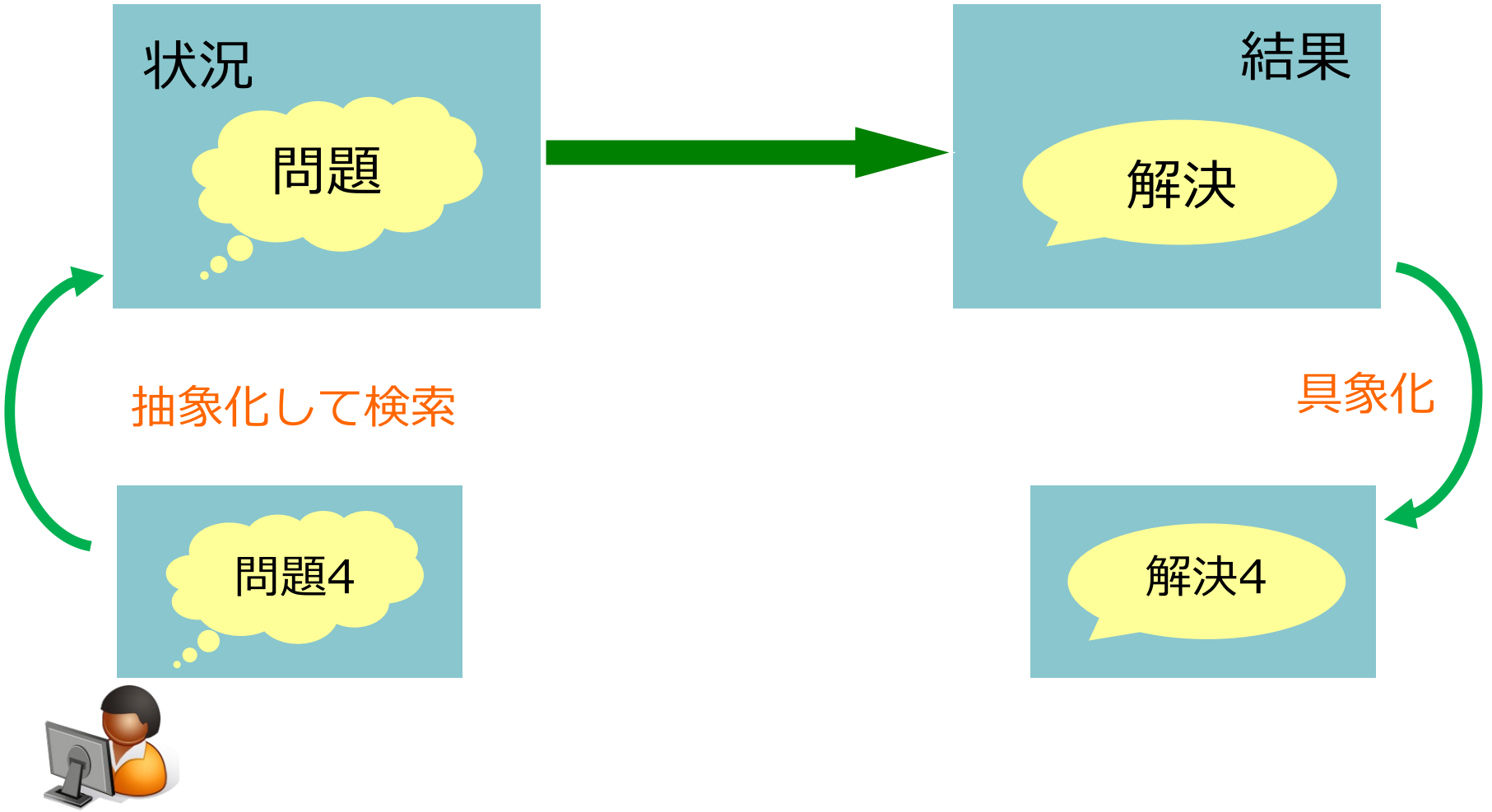
- どのような状況(context)で利用するか
- どのような問題(problem)に適用できるか
- どのように問題を解決(solution)するか
- どのような結果(consequence)や
  効果(利点や欠点)を与えるか



パターンの発見



パターンの活用



ソフトウェアパターンの種類(1/2)

- ビジネスパターン(business pattern)
  - ビジネス領域で必要となる典型的なデータや概念，ビジネスに登場する典型的な人物・もの・活動，ビジネスプロセス
- アナリシスパターン(analysis pattern)
  - 典型的な分析モデルの集まり
- アーキテクチャパターン(architectural pattern)
  - システム全体の構造や振る舞いに関する良い設計
  - ソフトウェアシステムの基礎的かつ典型的な構造
- デザインパターン(design pattern)
  - システムの構成要素の構造や振る舞いに関する良い設計
  - 特定のプログラミング言語に非依存
- コーディングパターン(Coding patterns)
  - イディオム，良く知られたアルゴリズム，コーディング規約，命名規則など

ソフトウェアパターンの種類(2/2)

- テストパターン(testing pattern)
  - テスト方法やテスト計画の優れた指針
- 保守パターン(maintenance pattern)
  - ソフトウェアプロダクトの改訂や改善に関する優れた指針や変更方法
- プロセスパターン(Process pattern)
  - ソフトウェア開発における人員，プロジェクト，プロダクトを管理する優れた指針
- セキュリティパターン(security pattern)
  - 特定の文脈において繰り返し発生するセキュリティの問題と，それらに対する実績のある一般的な解法
- アンチパターン(anti pattern)
  - 過去の失敗から学んだ教訓
  - 解決の悪い見本や「すべきでないこと」の集まり

デザインパターン

- 変更の繰り返しによる良い設計を集めたもの
- オブジェクト指向設計は簡単ではない
  - 変更容易性，保守性，拡張性，…
    - 将来起こりうる問題や要求に対応するのは大変
- ライブラリやフレームワークによる再利用は
  特定のプログラミング言語に大きく依存
  - コードや設計そのものを再利用できる機械は限られている
  - 設計の知見を再利用する場面の方が多い

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

GoFのデザインパターン

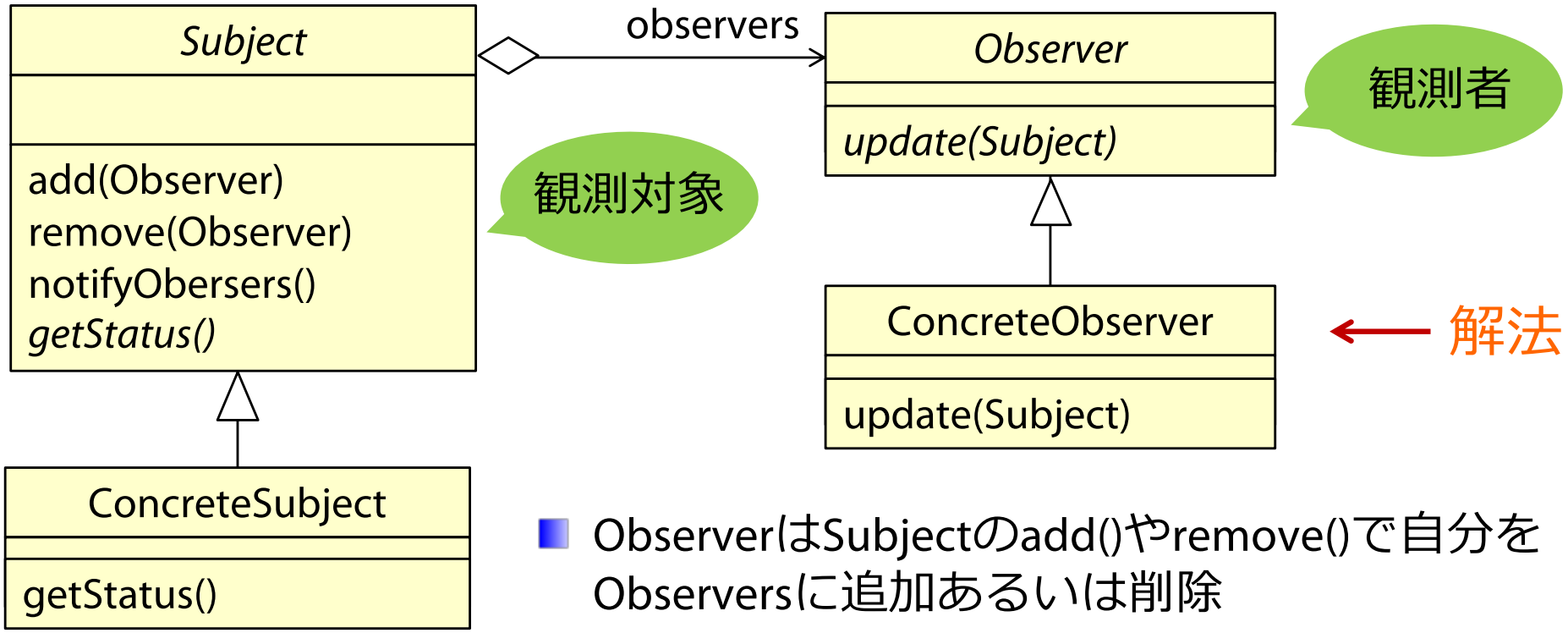
生成に関する	Abstract Factory	関連する部品を組み合わせて製品を作る
	Builder	複雑なインスタンスを組み立てる
	Factory Method	インスタンス生成をサブクラスに任せる
	Prototype	コピーしてインスタンスを作る
	Singleton	たった1つのインスタンス
構造に関する	Adapter	一皮かぶせて再利用
	Bridge	機能の階層と実装の階層を分ける
	Composite	容器と中身の同一視
	Decorator	飾り枠と中身の同一視
	Facade	シンプルな窓口
	Flyweight	同じものを共有して無駄をなくす
	Proxy	必要になっ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

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

Observerパターン ← パターンの名前

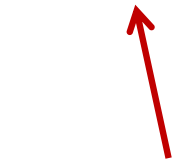
- あるインスタンスの状態が変化したときに
  変化を自動的に通知する仕組みを実現したい ← 状況と問題



- ObserverはSubjectのadd()やremove()で自分を
  Observersに追加あるいは削除
- Subjectは特定のイベントが発生したとき
  登録されているObserverのupdate()を呼び出す

Observerパターンの特徴・注意点

- イベント発生時にupdate()が呼ばれることを保証できる
- イベントに関する情報はupdate()の引数として
  渡すことができる
  - 特定のConcreteSubjectに特化した情報をSubjectに持たせるのは好ましくない
- 複数種類のConcreteObserverを用意すれば
  同じイベント通知に対して異なる処理を実現するのが容易
- 複数のObserverが登録されているとき
  どのような順番で呼ばれるかは原則として制御できない



結果や効果

Iteratorパターン

- 配列やコレクション内部の要素に対して
  内部のデータ構造を隠蔽した上で
  特定の順番で処理を行いたい

```
for (int index = 0; index< data.length; index++) {
  System.out.println(array[index]);
}
```

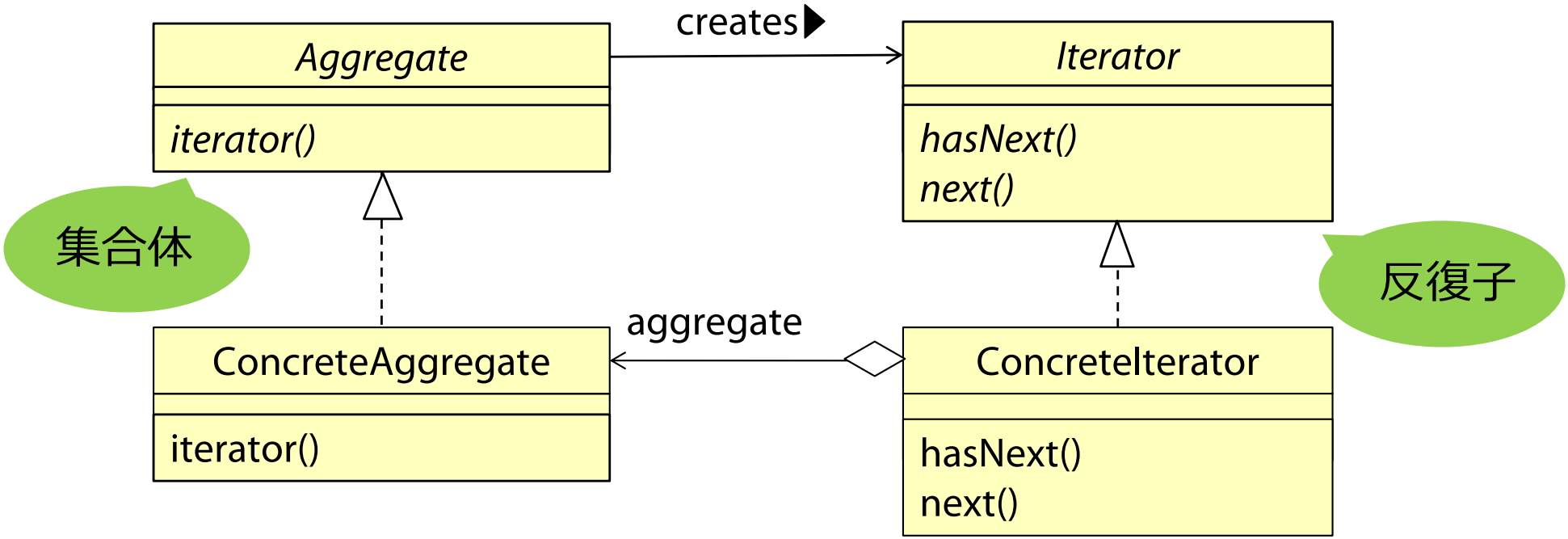
配列の場合

```
for (int index = 0; index< data.size(); index++) {
  System.out.println(data.get(index));
}
```

ArrayListの場合

内部のデータ構造に応じて，異なるコードを記述する必要がある

Iteratorパターンの構造



- ConcreteAggregateは iterator()が呼ばれると専用のIteratorを生成して返す
- hasNext()は次の要素が存在する場合にtrueを返す
- next()は次の要素を返し，着目点を次の要素へ進める

```
Iterator it = data.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

Iteratorパターンの特徴・注意点

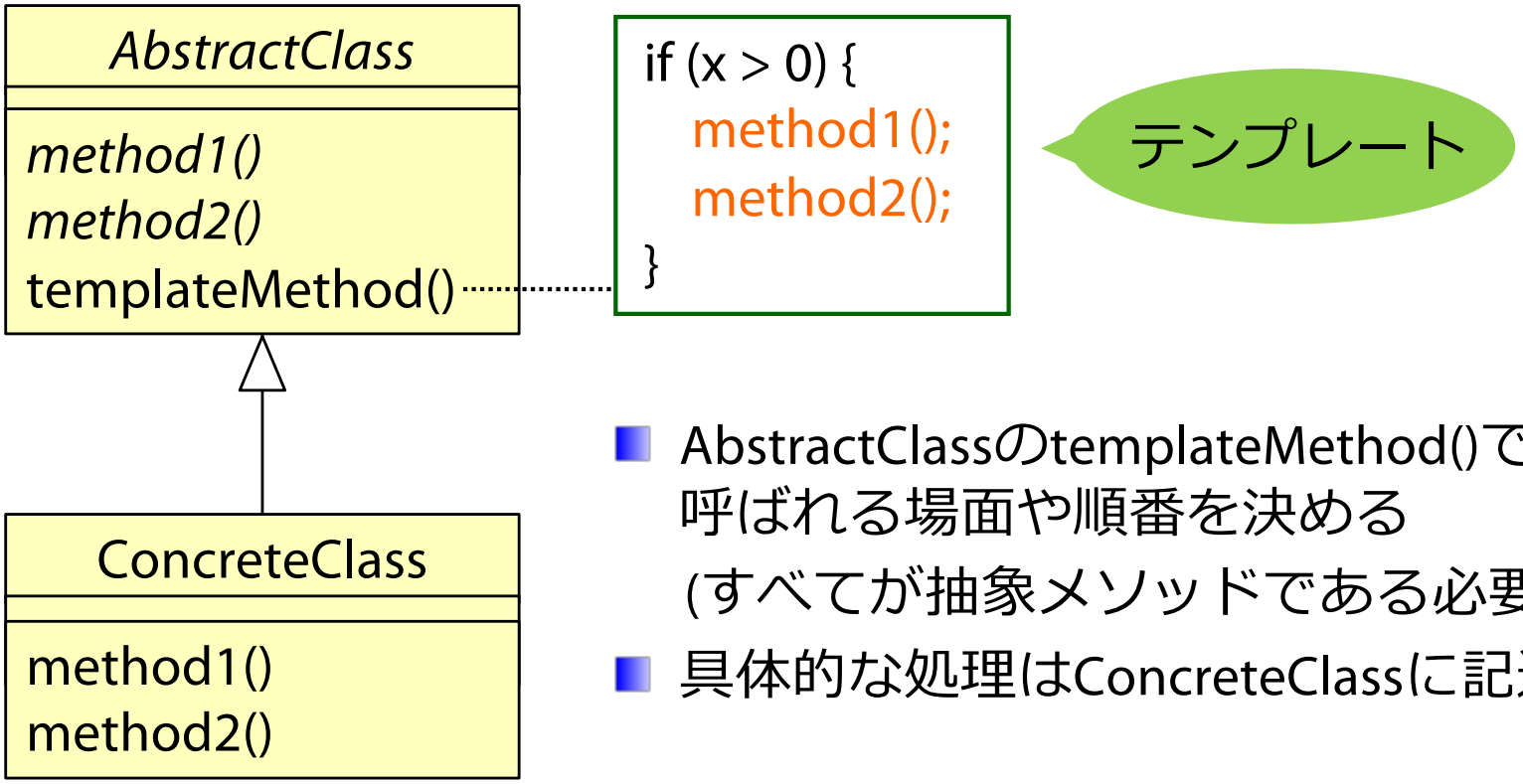
- 集合体の実装に関わらず同じように繰り返し処理ができる

```
Iterator it = data.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

- 集合体の「取り出し方」を再利用できる
  - 先頭から順番に1つずつ取り出す
  - 末尾から逆順に1つずつ取り出す
  - 先頭から順に1つ飛ばしで取り出す
  - 奇数だけ取り出す 等
- 処理の途中で集合体に変更を加えるのは危険
  - 原則禁止
  - Javaでは，ConcurrentModificationExceptionが発生するときがある

Template Methodパターン

- 複数の処理がまったく同じ手順である，あるいは一部だけが異なるとき
- 処理の枠組み(テンプレート)を先に定め，具体的な処理を後から作成したい



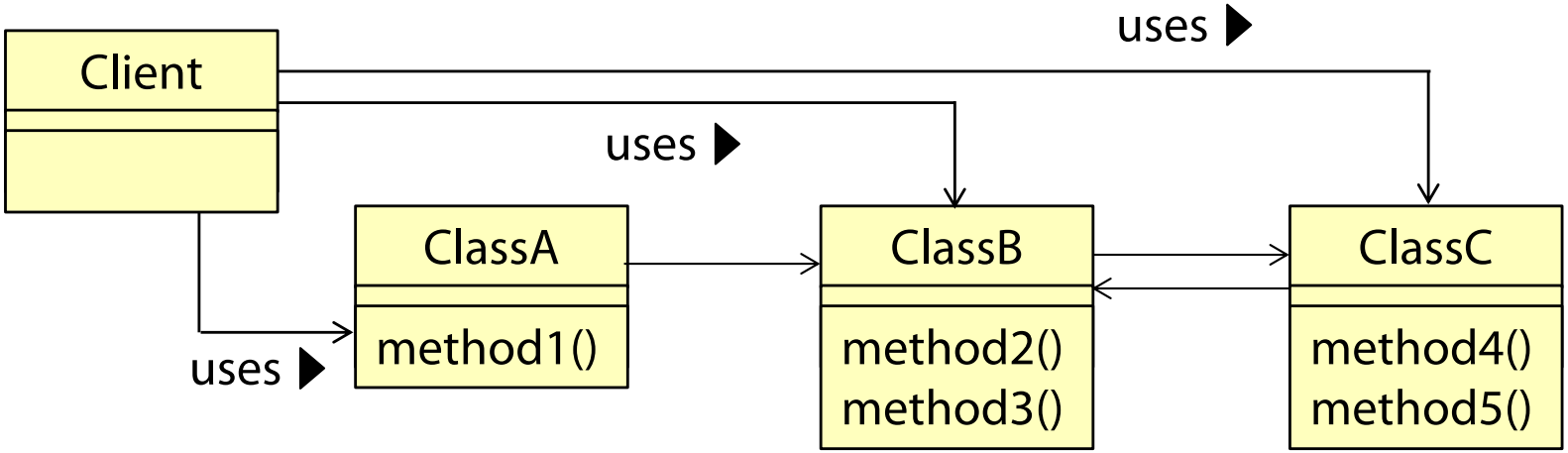
- AbstractClassのtemplateMethod()でメソッドが呼ばれる場面や順番を決める
  (すべてが抽象メソッドである必要はない)
- 具体的な処理はConcreteClassに記述

Template Methodパターンの特徴・注意点

- ロジックを共通化できる
  - 同じような処理を何度も書かなくて良い
    - テンプレートメソッドにだけ記述
  - もしコピー＆ペーストでコードを複製すると
    一箇所に変更があった場合，すべてに対して同じ変更を加える必要
- トップダウンに設計できる
  - 親クラスを作った時点で，子クラスで実装しなければならない内容を明確化できる
- 制御の逆転の実現
  - 抽象メソッドによりホットスポットの位置を明示
- 何でも共通化すれば良いというわけではない
  - コードが分散するので，理解しにくくなる
  - Javaの場合，動的束縛になるのでコストがかかる

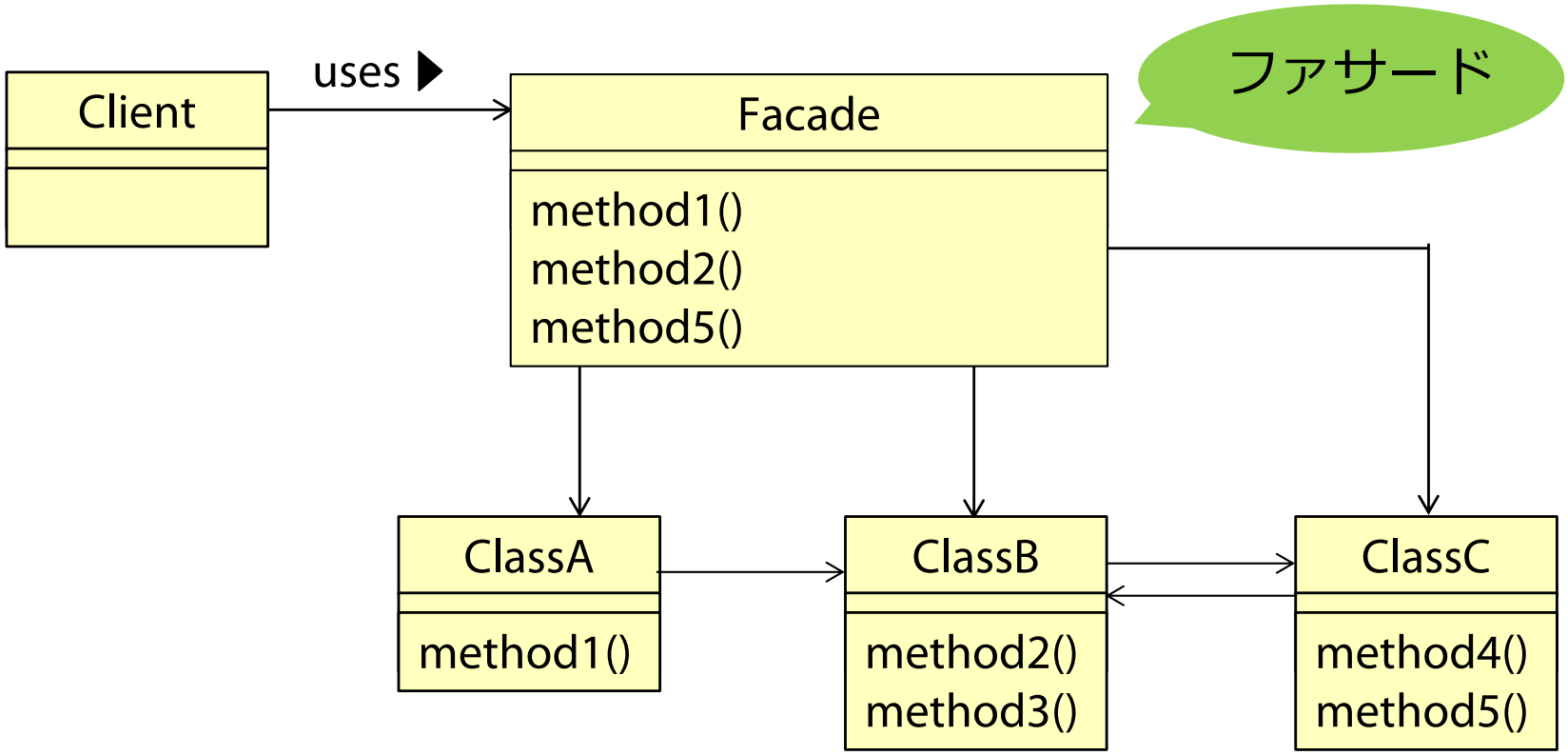
Façade(ファサード)パターン

- 互いに関係を持つ複数のクラスを利用する窓口を用意することで，複雑な処理を利用しやすくしたい



Clientが複雑な処理を直接実行 

Façadeパターンの構造



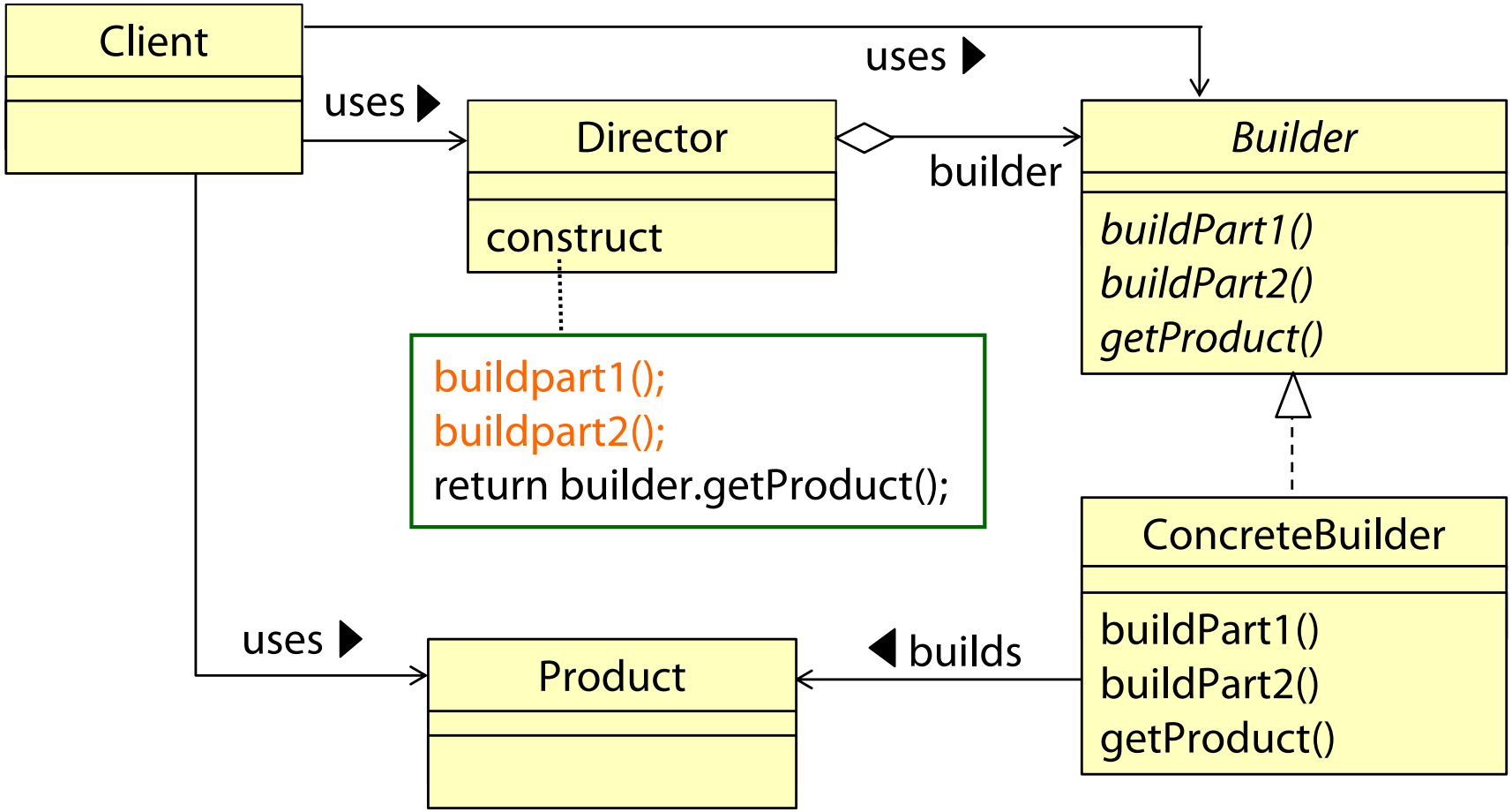
- Clientから利用するメソッドのみをFacadeで提供
- ClientはFacadeを窓口として利用

Façadeパターンの特徴・注意点

- インタフェースを削減する
  - 機能を利用する時に，どこを呼べば良いかを明示
  - 内部の詳細をしらなくても，処理が実行できる
- オブジェクト間の依存関係が多いときに適用すると直接的な依存関係を減らすことができる
- Facade自身は実装クラスを隠蔽するわけではない
  - 実装をパッケージアクセスにすることで隠蔽することもある

Builderパターン

- 複雑な構造を持つインスタンスの生成の際，
  生成手順をいくつかに分割して実現



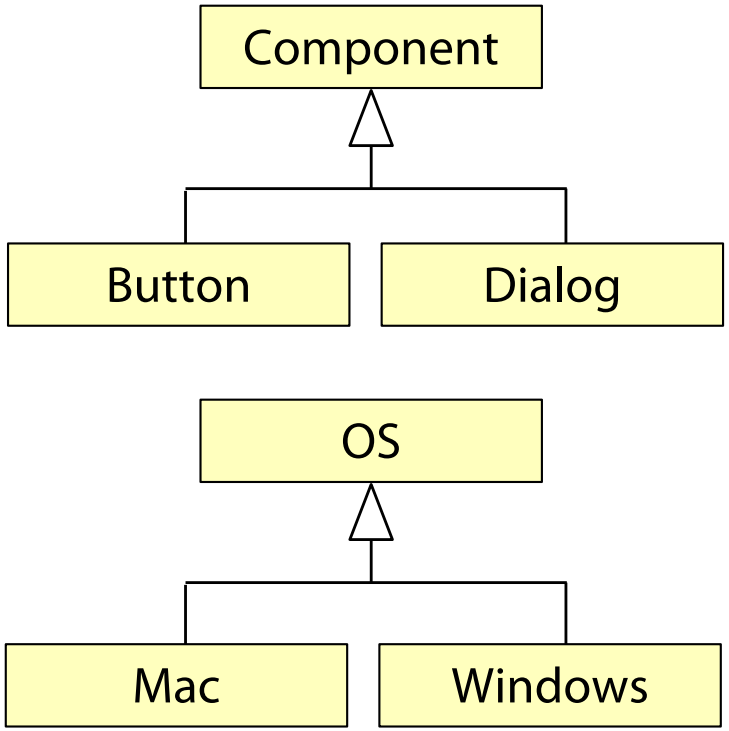
- ObserverはDirectorのconstruct()がProductを生成

Bulidерパターンの特徴・注意点

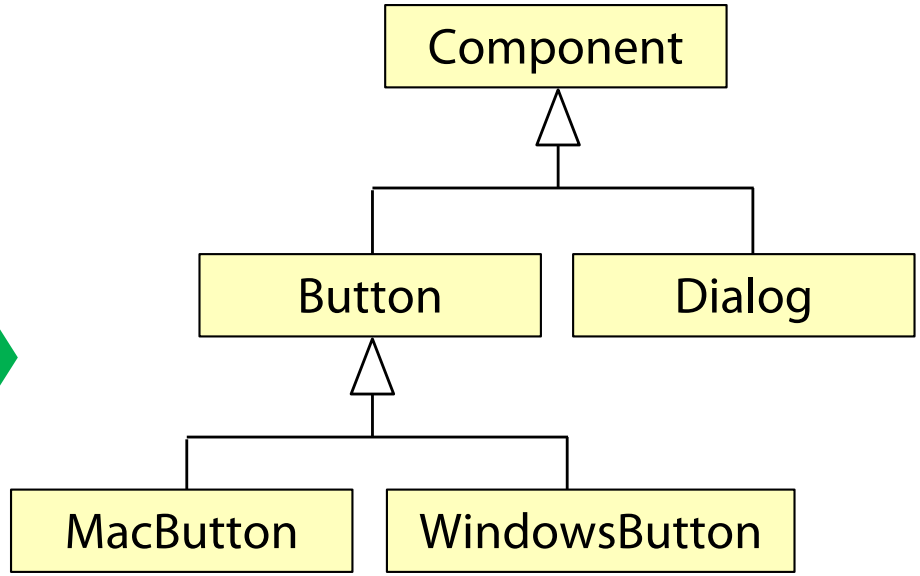
- Product内部の変更が容易になる
  - Product内部の変更はConcreteBuilderを更新することで吸収(ClientやDirectorに影響しない)
- Productを生成するためのコードを局所化
  - インスタンス生成に関わるコードをConcreteBuilderに隠蔽
- 各ConcreteBuilderごとに異なる製品クラスを作成しても良い
- DirectorはFacadeの一種
- Productの生成の構築に必要なメソッド群をすべて提供する必要がる
  - 将来の拡張も想定しなければならない

Bridgeパターン

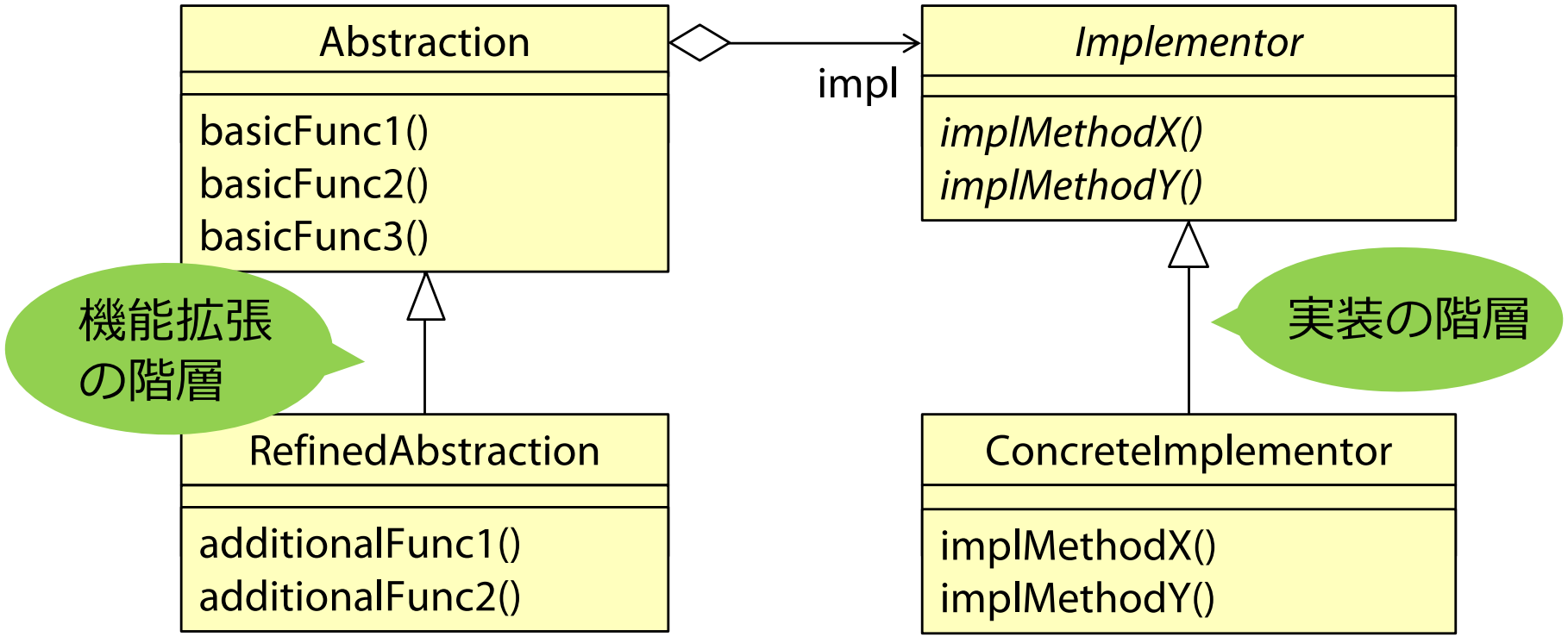
- 複数の観点で継承階層が現れるとき
- 機能のクラス階層(機能拡張が目的の継承)と
  実装のクラス階層(抽象クラスの実装が目的の継承)を
  分離したい



分離されていないと，
階層構造が複雑になる



Bridgeパターンの構造



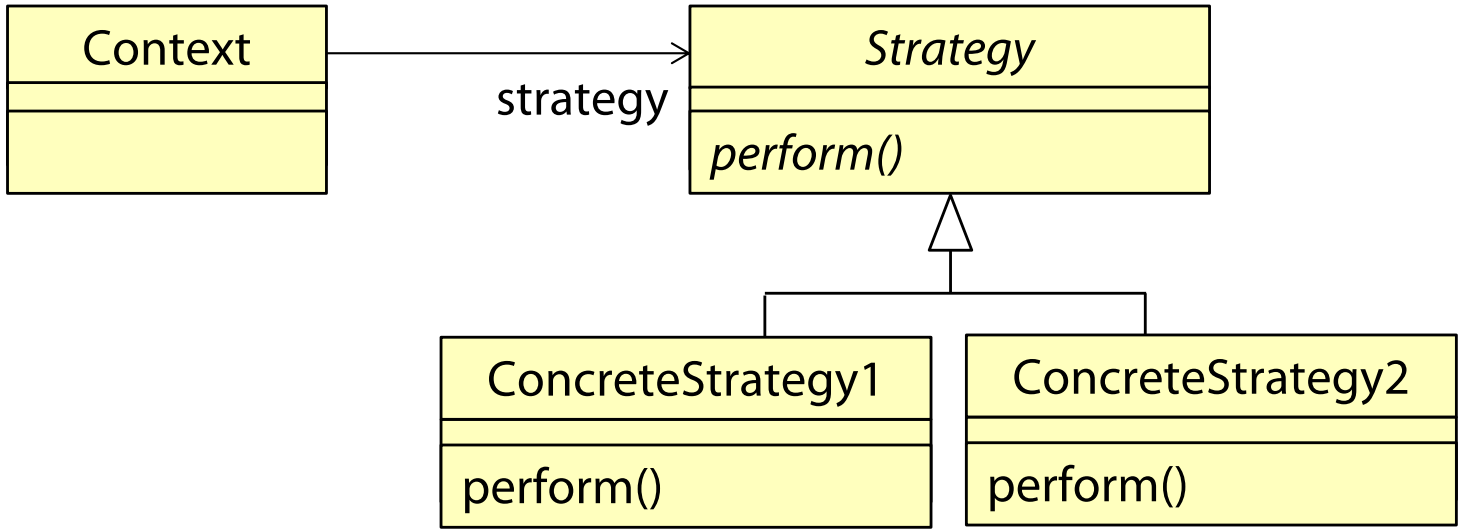
- Abstractionに対して，RefinedAbstractionで機能を拡張
- Implemenetorの子クラスでは具体的な処理を実装
- Abstraction, RefinedAbstractionでは
  Implementorのメソッドを組み合わせて機能を実現

Bridgeパターンの特徴・注意点

- Template Methodパターンのテンプレートメソッドを他クラス(Abstractionとその子孫)に切り出したものと捉えることができる
- インタフェースと実装のクラス階層を
  別々に用意することで，それぞれ独立に拡張可能
  - 実装の拡張がインタフェースに直接影響を与えない
- 外部から実装の詳細を隠蔽
  - 実装の階層は外から使わせない
- Abstractionが保持するImplementorインスタンスは後から切り替えることが可能
- クラス数が多くなりがちなので理解性に注意
  - 実装クラスには”Impl”を付ける等の命名規則やコメントにより若干改善

Strategyパターン

- アルゴリズムの切り換えを簡単に行いたい



- ObserverはStrategyに代入するインスタンスを変えることで，利用するアルゴリズムを切り替える

状態をクラスとして定義して，現時点の状態を切り替えるStateパターンがある
StrategyパターンとStateパターンは行動は同じであるが，目的が異なる

Strategy Patternの特徴・注意点

- 委譲を使うことで，ContextがConcreteStrategyを直接使うよりも疎結合
  - アルゴリズムの個別に定義できる
  - アルゴリズムが使用するデータをContextから切り離すことができる
- 条件判定との分離

```
data = readData();
if (n > 1000)
  quickSort();
else
  bubbleSort();
```

```
if (n > 1000)
  sorter = new QuickSort();
else
  sorter = new BubbleSort();
data = readData();
sorter.sort(data);
```

Strategy
パターンの適用

- すべてのConcreteStrategyが同じ入力を持つとは限らない
  - strategyMethodの一部の引数が使われず無駄になることがある

まとめ(1/2)

- ソフトウェアパターンは，開発者の経験や知見を再利用できるように体系化したものである
- ソフトウェアパターンは名前を持ち，コミュニケーションを効率的に行うための標準的な語彙となる
- ソフトウェアパターンは，どのような状況(context)で利用するか，どのような問題(problem)に適用できるか，どのように問題を解決(solution)するか，どのような結果(consequence)や効果(利点や欠点)を与えるかという観点でカタログ化されている
- デザインパターンは，変更の繰り返しによる良い設計を集めたものである

まとめ(2/2)

- Observerパターンでは，あるインスタンスの状態が変化したときに変化を自動的に通知する
- Iteratorパターンでは，配列やコレクション内部の要素に対して内部のデータ構造を隠蔽した上で特定の順番で処理を行う
- Template Methodパターンでは，処理の枠組みを先に定め，具体的な処理を後から作成する
- Façadeパターンでは，互いに関係を持つ複数のクラスを利用する窓口を用意する
- Builderパターンでは，複雑な構造を持つインスタンスをいくつかの手順に分割する
- Bridgeパターンでは，機能のクラス階層(機能拡張が目的の継承)と実装のクラス階層(抽象クラスの実装が目的の継承)を分離したい
- Strategyパターンでは，アルゴリズムの切り替えをインスタンスの切り替えにより