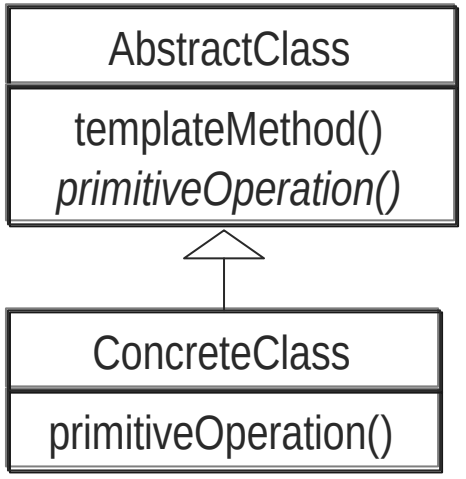
デザインパターンと その適用過程のモデル化

東京工業大学 学術国際情報センター
情報基盤部門

小林 隆志 tkobaya@gsic.titech.ac.jp

研究の背景

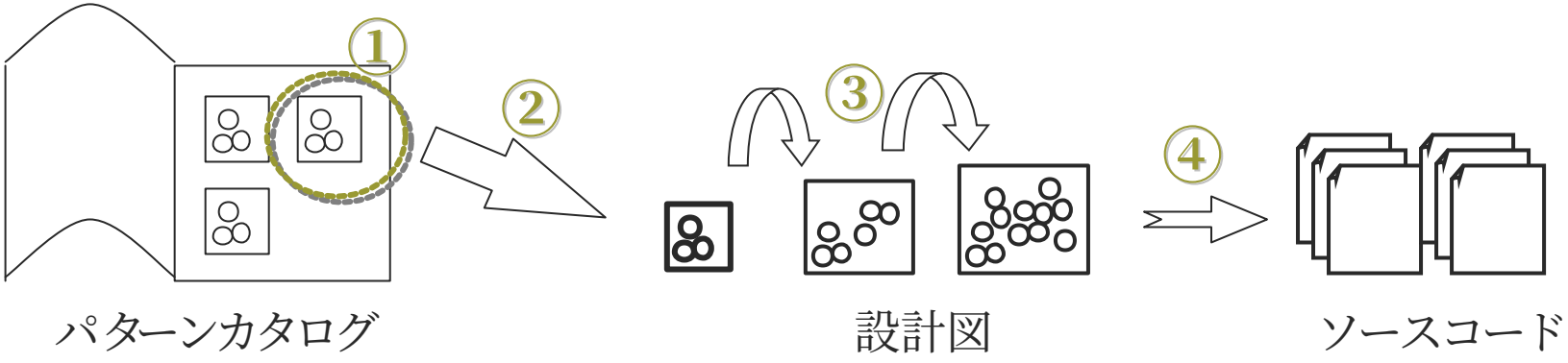
- デザインパターン
  - 設計時に頻繁に現れる構造を抽象化
  - GoFパターンが有名
  - 継承, 多態性を利用した
    再利用可能なクラス群
  - 利点と問題点
    - ○ 拡張に対して強い設計
    - × 拡張性のために, 複雑な機構を持つ



Template Methodパターン

デザインパターンの利用の問題

- デザインパターンの利用過程(リファクタリングは除く)
  - ① カタログの “構造”, “適用の可能性”を参考に選択
  - ② 選択したパターンの構造を, 適切にリネーム/マッピング
  - ③ 各問題にあわせるために, 抽象化されている部分を具体化
  - ④ カタログにある“仮想コード”を参考に**コード実装**



デザインパターンの利用の問題

- デザインパターンの利用過程での問題

① 選択

どのパターンを使用するべきか

②③ 適用(リネーム/マッピング, 具体化)

どこをどのように変更するか

組合わさったパターン間での依存関係の解決

適切な使用がなされているか

ここを支援

④ **コード実装**

仮想コードによる協調動作定義 の反映

適用を支援する既存研究

- 従来の研究
  - パラメータ化
    - × パターンのメカニズムを壊す変更が可能
    - × 組合わさった場合に, 影響を解析できない
    - ⇒ パターンの持つ情報を利用するためにモデルが必要
  - パターンの非形式的モデル化
    - × カタログの電子化 :計算機による支援はできない
  - パターンの宣言的形式モデル化
    - × パターンに対する操作を記述できない
    - × ツールに組み込むことが困難

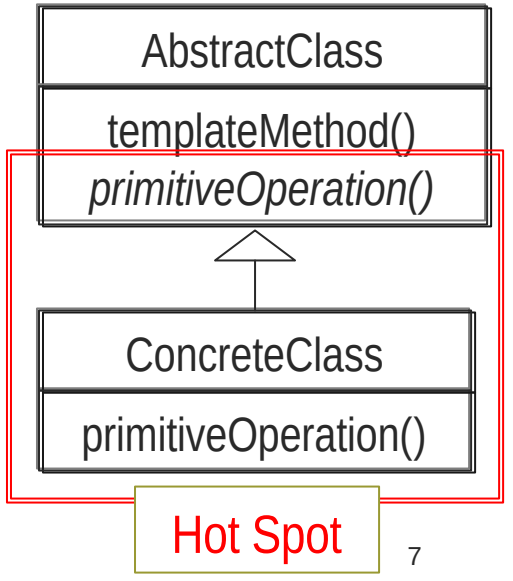
本研究のアプローチ

- サブゴール
  - 簡潔, 形式的, 操作的 であるモデルの作成
  - モデルを用い, パターン定義を記述
  - パターン記述を利用できる 支援ツールを実装
  - 組合わされたパターンへの対応
  - パターンのメカニズム/挙動確認

パターンの適用過程

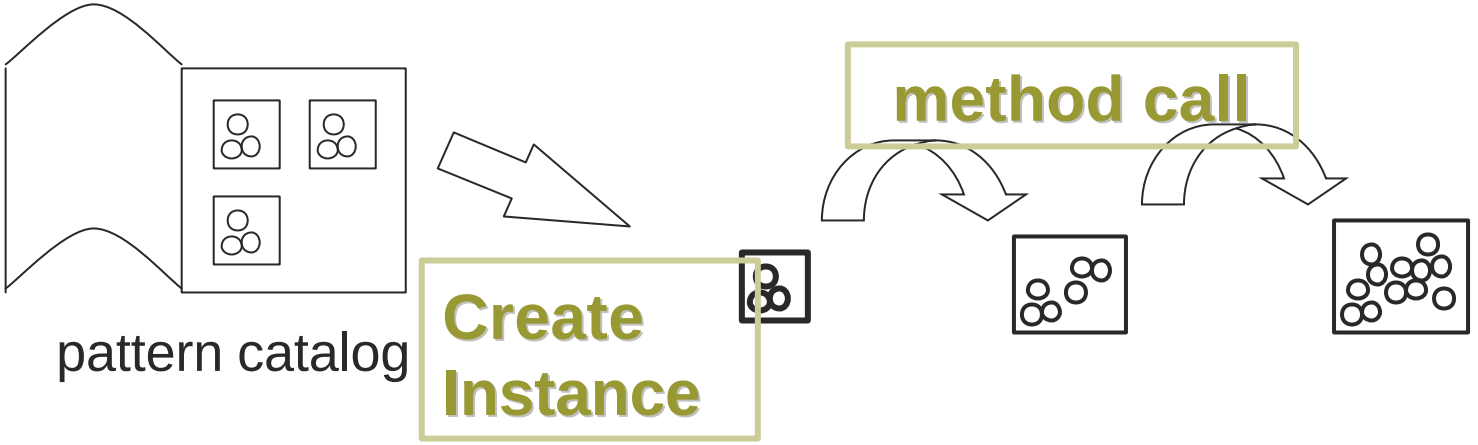
- パターンを構成するものは 大きく分けて2つ
  - Frozen Spot: 変更が不要(変更してはいけない)部分
  - Hot Spot : 問題に合わせ変更する必要がある抽象化された部分
- 適用過程

  = Frozen Spot 部の構築 + Hot Spot部の具体化
  - 例 :TemplateMethod パターン
    - AbstractClass の作成
    - TemplateMethod ()の作成
    - 内部で呼び出す PrimitiveOperation()の作成
    - サブクラスを作成しPrimitiveOperation()を作成



パターンのモデリング(1/2)

- Frozen/Hot Spot, 具体化方法はパターン固有
- パターンをクラスとして捉える
  - 適用 = パターンのインスタンスを作成
    + メソッドでホットスポットをうめる



パターンのモデリング(2/2)

- パターンをクラスとして
  オブジェク指向モデリング

  要素　　⇒ 属性

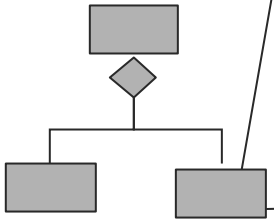
  FrozenSpot ⇒ コンストラクタ

  具体化　⇒ メソッド

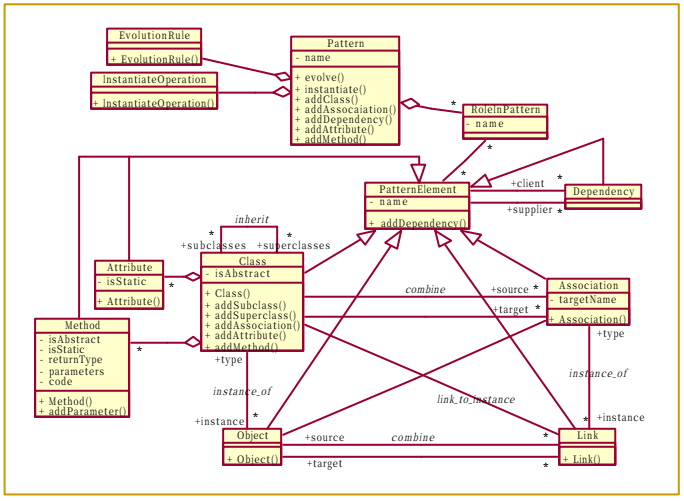
  組合わせ ⇒ 集約

- 適用 = パターンのインスタンスを作成
  + メソッドでホットスポットをうめる

```
Pattern#1{
  Class A;
  Class B;
  ……
  Pattern#1{
    ..FrozenSpotの生成..
  }
  Inst_Op#1(…){
    ..HotSpotへの変更..
  }
}
```



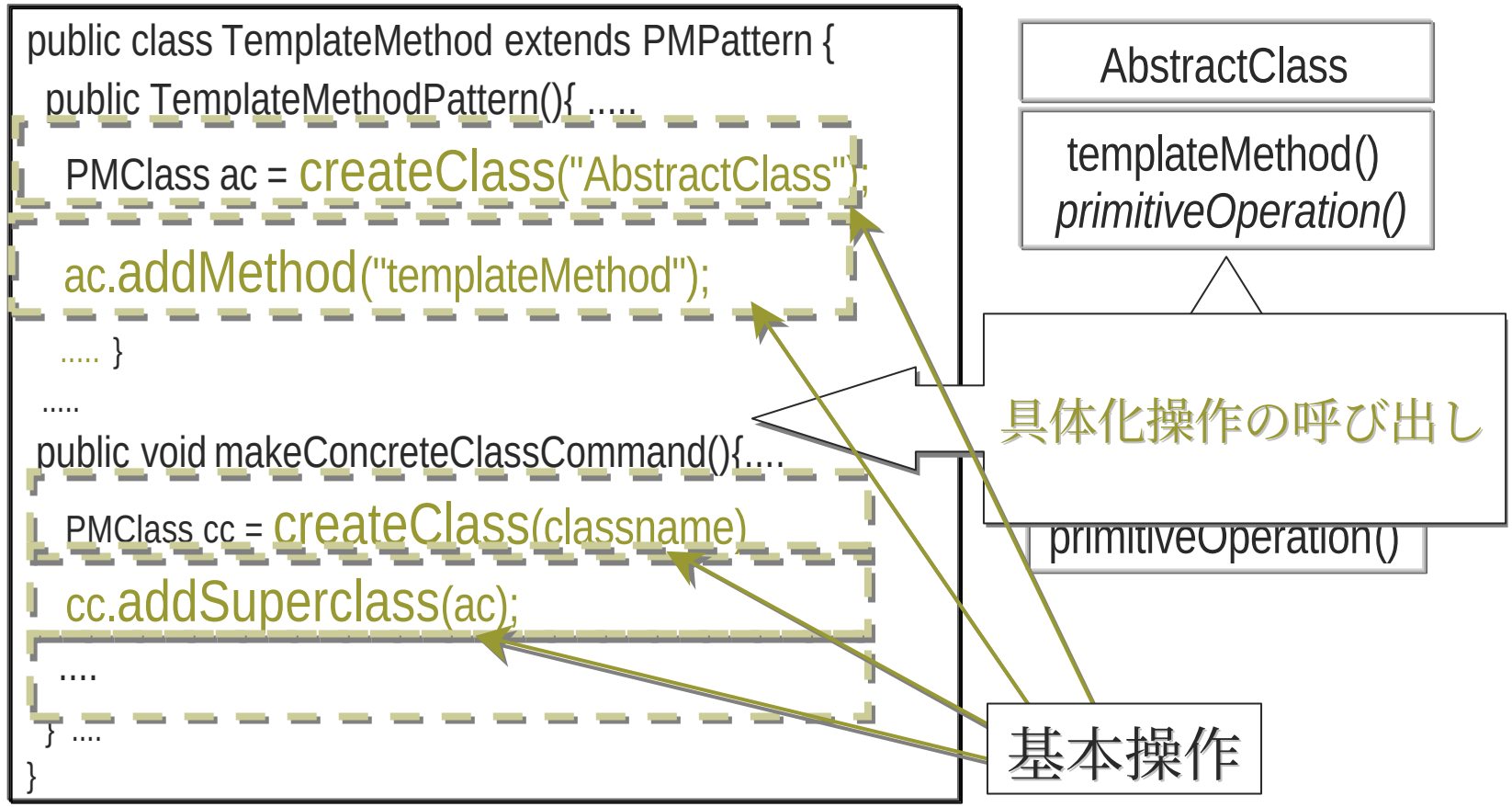
本研究のモデル



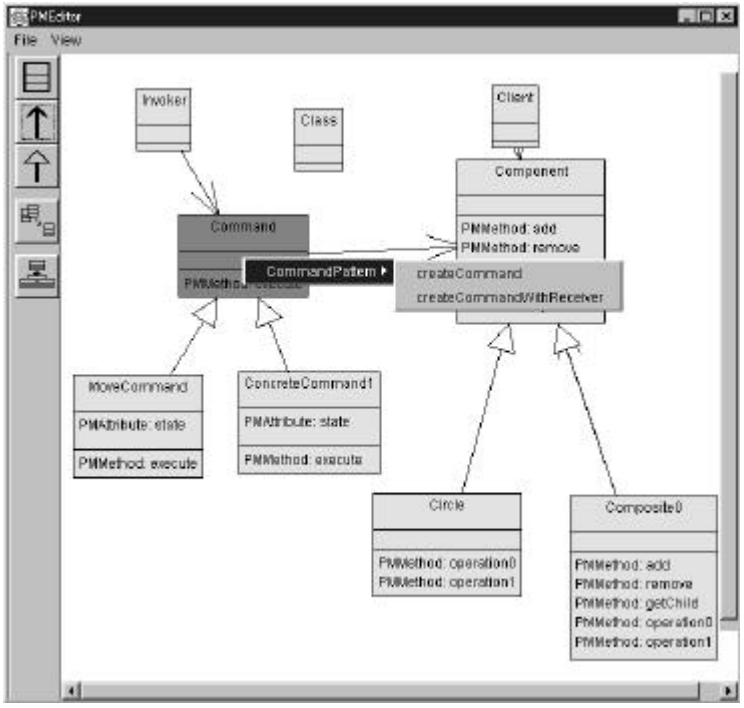
- 特徴
  - 適用時の具体化操作の表現
  - UMLのメタモデルより簡単な構造
- 利点
  - 適用方法の形式化 =
    蓄積可能 ＋ 計算機支援
  - クラス図内にパターン情報を保持
- 記述言語にはJavaを採用
  - 基本操作を実装
- GoFの23個のうち 22個を記述

TemplateMethodパターンの記述

```
public class TemplateMethod extends PMPattern {
  public TemplateMethodPattern(){ .....
    PMClass ac = createClass("AbstractClass");
    ac.addMethod("templateMethod");
    ..... }
  .....
 public void makeConcreteClassCommand(){....
    PMClass cc = createClass(classname)
    cc.addSuperclass(ac);
    ....
  } ....
}
```

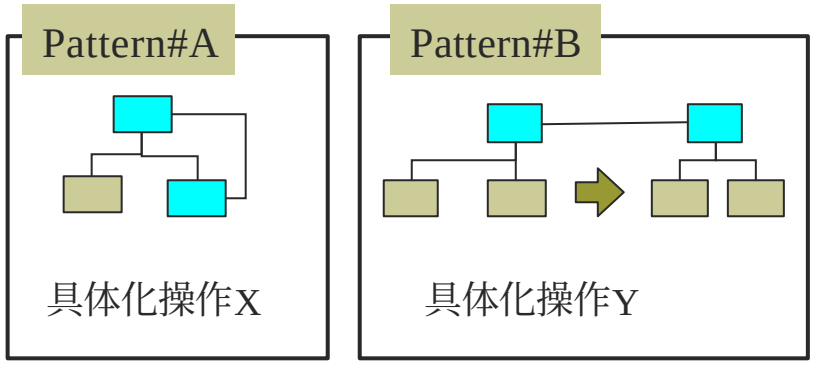


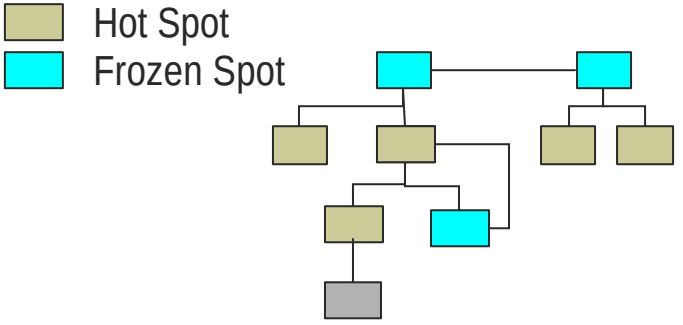
支援ツールの実装



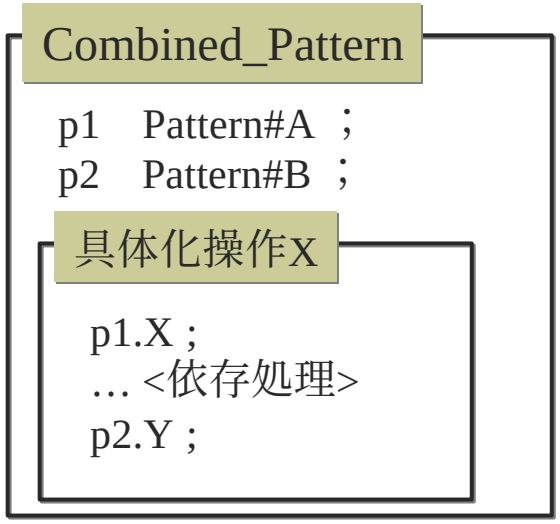
- クラス図エディタ
- パターンのインスタンスを生成
- 具体化操作の呼び出し

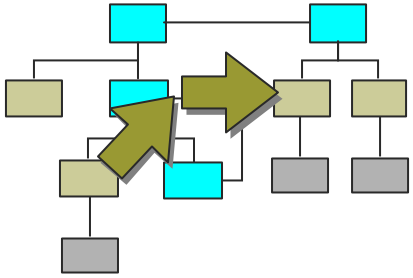
パターンの組み合わせ





依存する具体化ができない

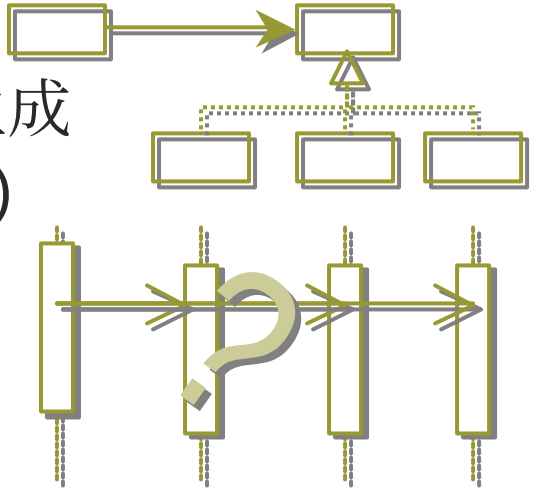




依存関係のカプセル化

メカニズム/挙動の確認

- 問題点
  - 動的束縛で実行されるメソッドが動的に変化する.
- 仮想コードの意味するもの
  - メソッドの呼び出し, インスタンス生成
  - プログラムの意味 (繰り返し, 分岐)
- これらを モデルに関係として組込む
  - ActionSemanticsの利用を検討中
- シーケンス図による確認



現段階での達成度

- サブゴール
  - 簡潔, 形式的, 操作的 であるモデルの作成
  - モデルを用い, パターン定義を記述
  - パターン記述を利用できる 支援ツールを実装
  - 組合わされたパターンへの対応
  - パターンのメカニズム/挙動確認

まとめ

- 目標 :デザインパターンの利用を支援
- 利用 = 選択＋適用＋実装
- 適用を支援するための OOモデルを作成
- 記述言語にJavaを採用し, 基本操作を実装
- GoFの23個のうち22個を記述
- パターン記述を利用した, 支援ツールの実装
- パターンの組合わせに関する支援
- パターンのメカニズム/挙動に関する考察