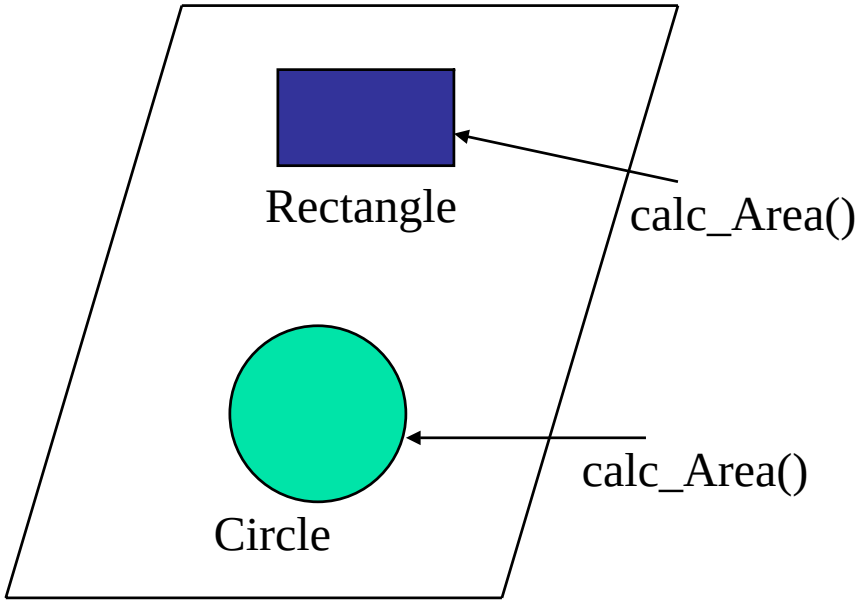
オブジェクト指向のよさ

- データ抽象
- ポリモルフィズム(Polymorphism)
- 継承と関連（集約）に基づく構造化
  - 継承：差分プログラミング
  - 関連：柔軟な変更が可能
  - 抽象クラス(Abstract Class)

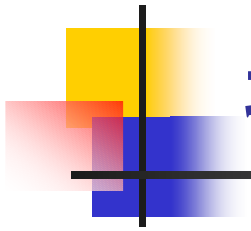
  → 再利用性を高める

抽象クラス(1)

例:図形の面積を計算する(calc_Area)




```
public class Rectangle {
    int position_x, position_y ;
    int w, h ;
    ...
    public double calc_Area() {
        [          ]
    }
}

public class Circle {
    ...
    public double calc_Area() {
        [          ]
    }
}
```



抽象クラス（2）

- 共通の性質をまとめる
- 抽象クラス：オブジェクトの生成機構がない

（例）

RectangleとCircleの共通の性質：

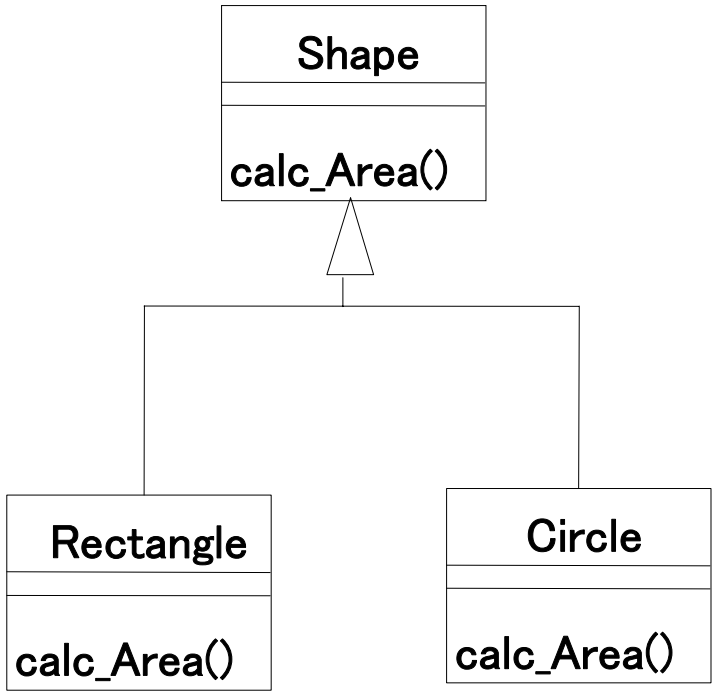
calc_Areaメソッドを持つ

calc_Areaメソッドの実装（コードの記述）はRectangle, Circleで行われる

- 抽象メソッド：実装されていない

  抽象メソッドをもつクラスは抽象クラスでなければならない

抽象クラス（3）



抽象クラス

```
public [abstract] class Shape {
    public [abstract] double calc_Area()
}
```

抽象メソッド

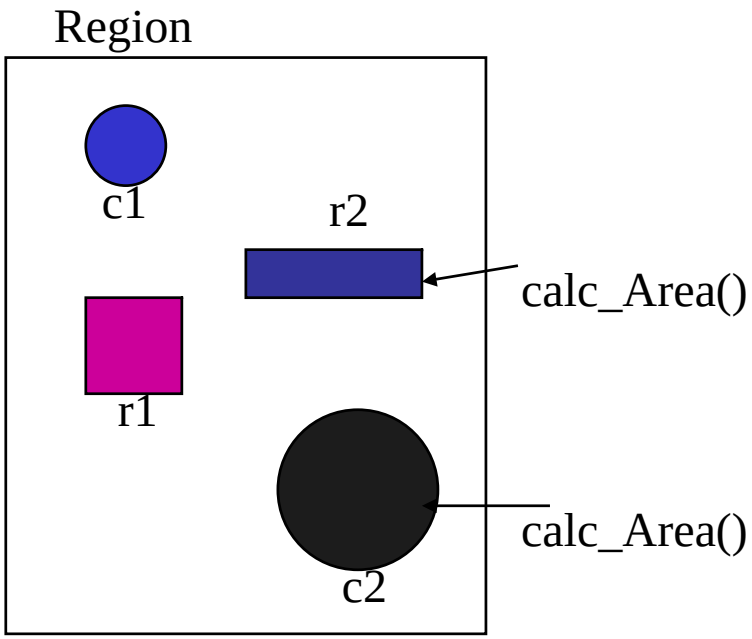
```
public class Rectangle [extends] Shape {
    ...
    public double calc_Area() {
        return (double) (w*h) ;
    }
}
```

実装

継承を用いた実装

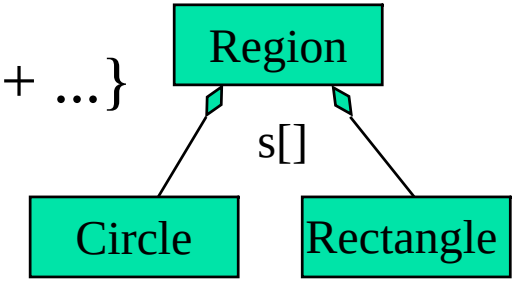
抽象クラスからパターンへ（1）

（例）ある領域にあるRectangleとCircleの面積の総和

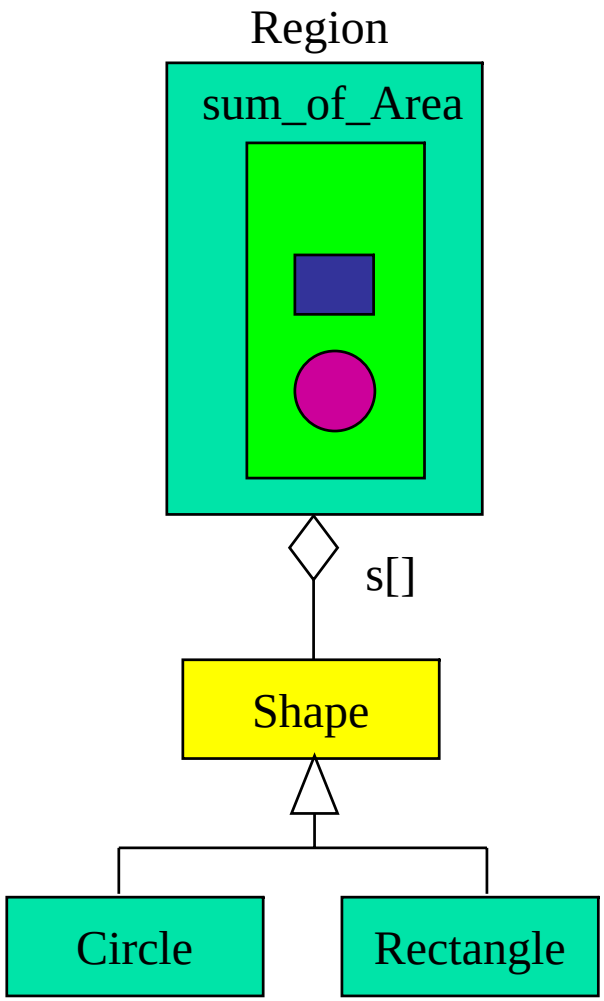


c1.calc_Area ()+ r2.calc_Area() +
r1.calc_Area() + c2.calc_Area()

```
Public class Region {
   Shape s[] ;

  double public sum_of_Area() {
     double sum = 0 ;
     for (int i=0 ; i < s.length ; i++) {
        /* if s[i] is a rectangle ...
             if s[i] is a circle ..
        */
        sum = sum + ...}
     return sum ;
   }
}
```

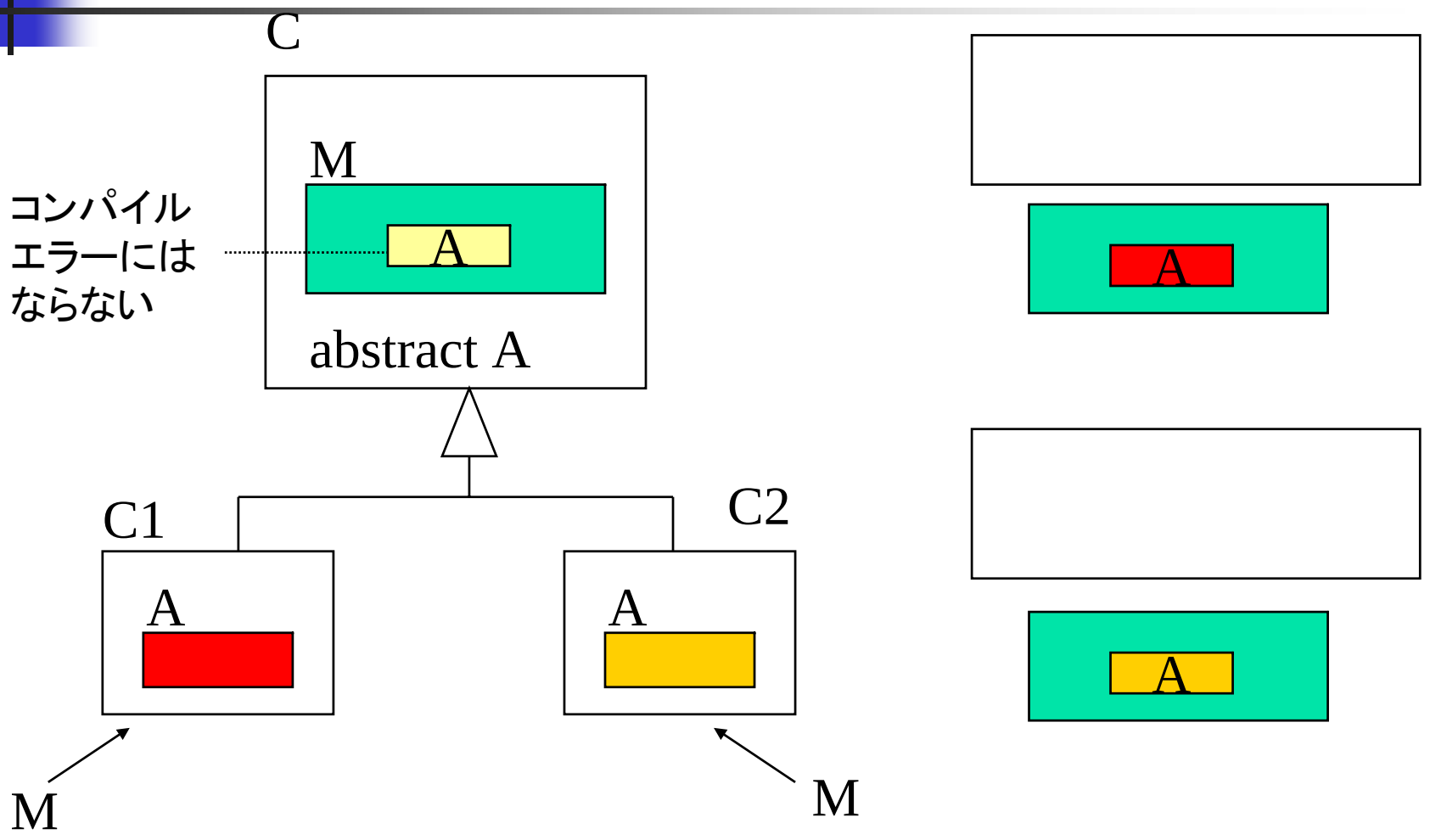


抽象クラスからパターンへ（2）



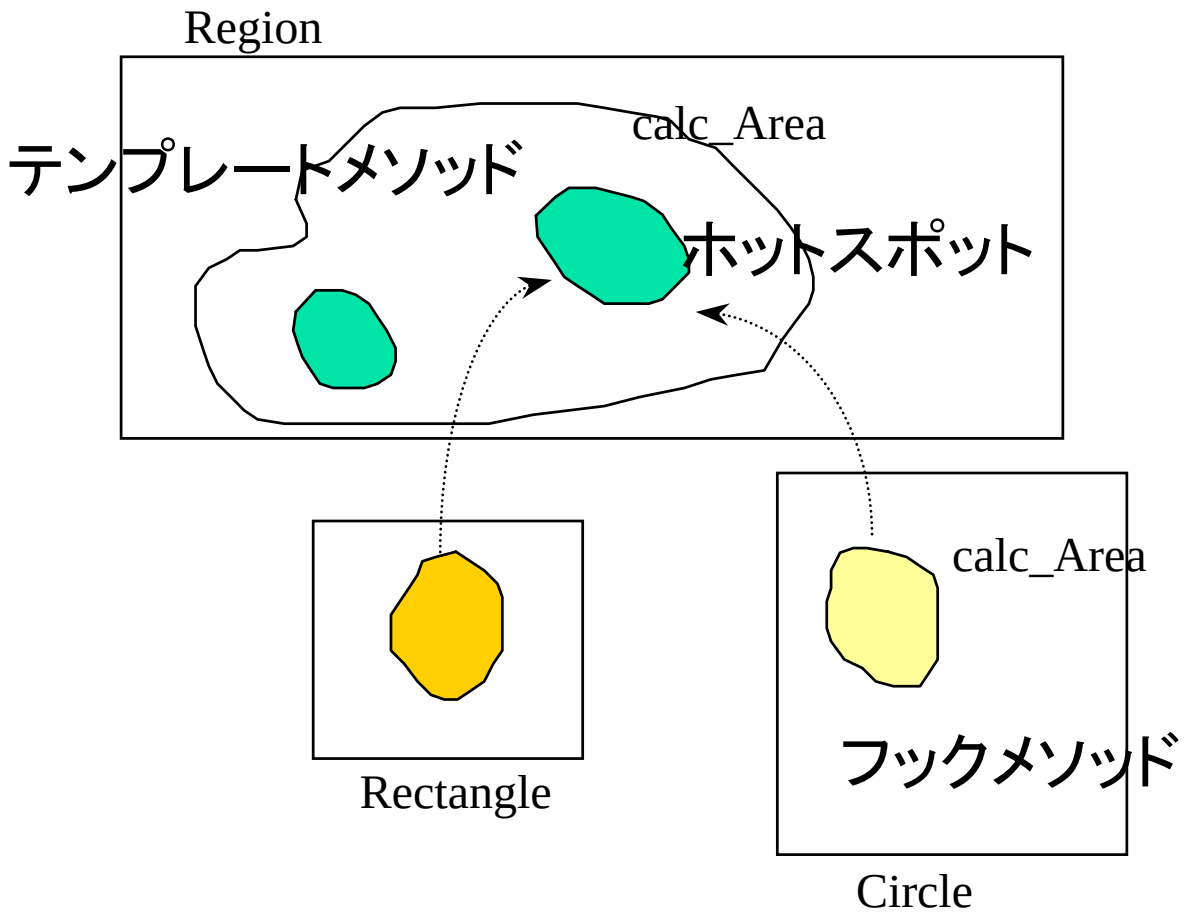

```
public class Region {

    Shape s[] ;

 public double sum_of_Area() {

        double sum = 0 ;

        for (int i=0 ; i < s.length ; i++) {

        }

        return sum  ;

}
```

パターンへ（1）

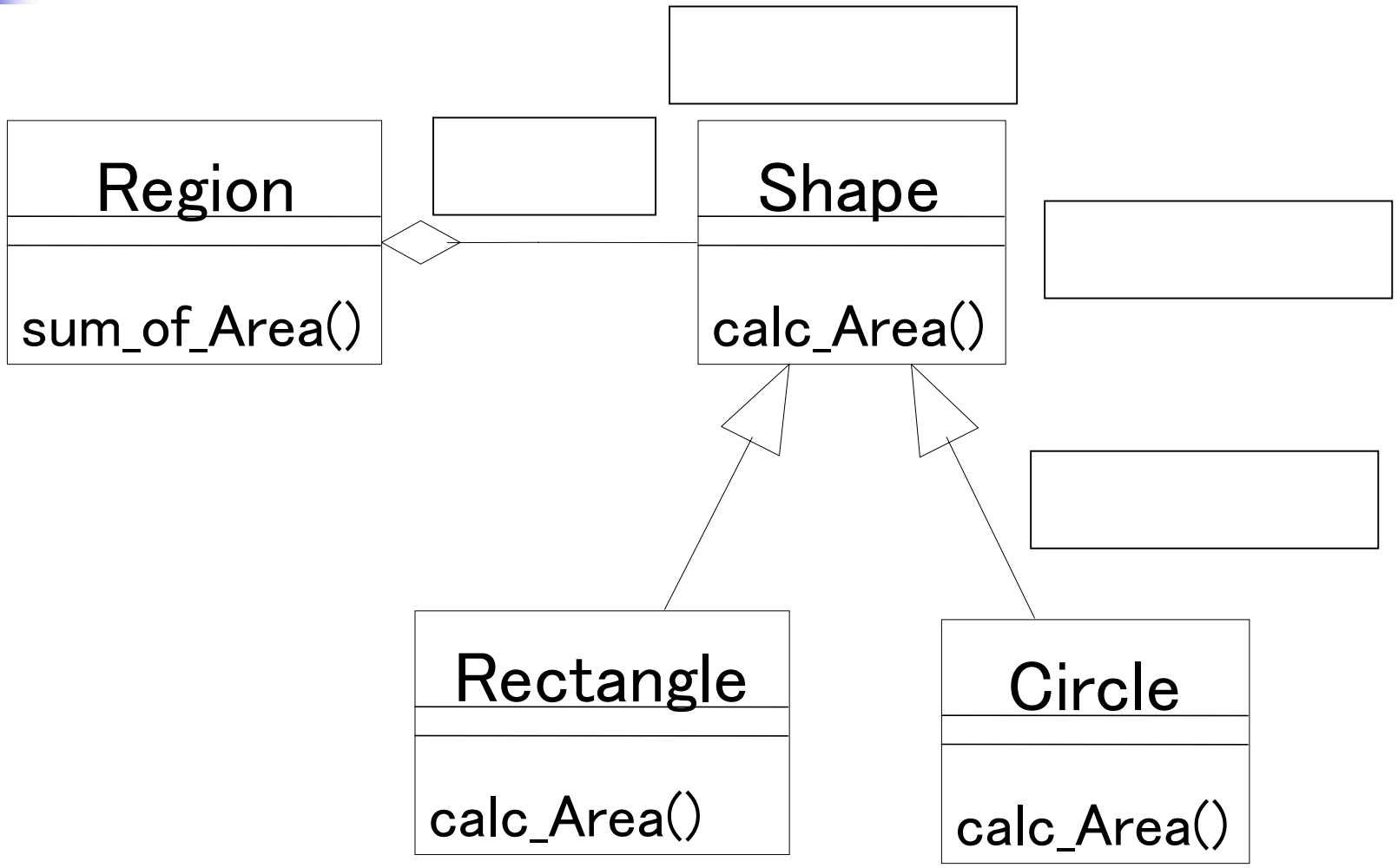


Aの実装をオブジェクトによって切り替える

パターンへ（2）


Region
calc_Area
テンプレートメソッド
ホットスポット
Rectangle
calc_Area
フックメソッド
Circle

パターンへ（3）


Region
sum_of_Area()
Shape
calc_Area()
Rectangle
calc_Area()
Circle
calc_Area()

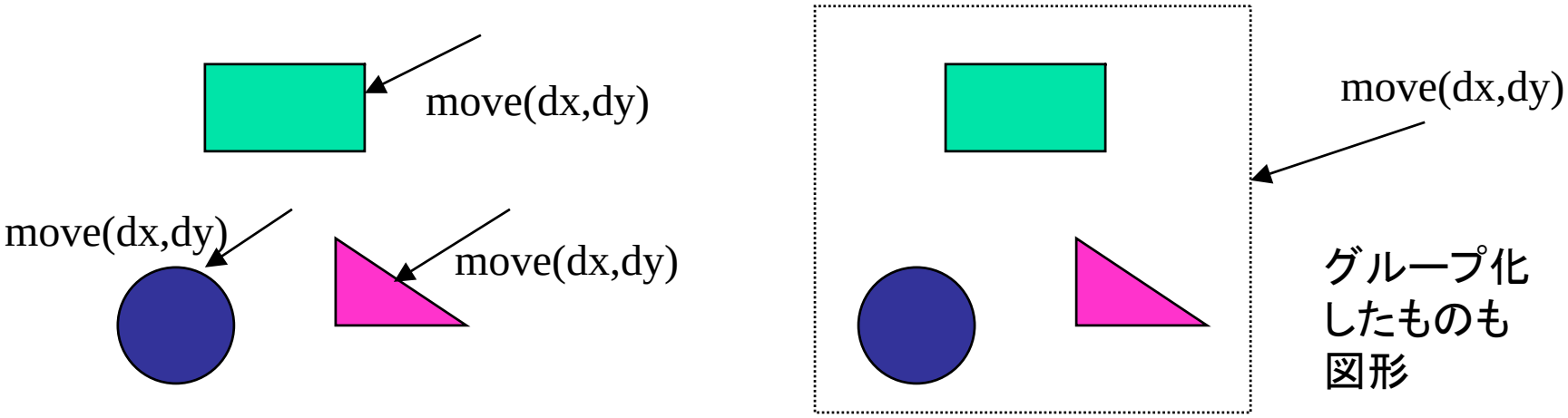
パターンへ（4）

サブクラス（継承）の利用

抽象クラス：インスタンスの生成機構を持っていない
共通の性質をあつめたもの

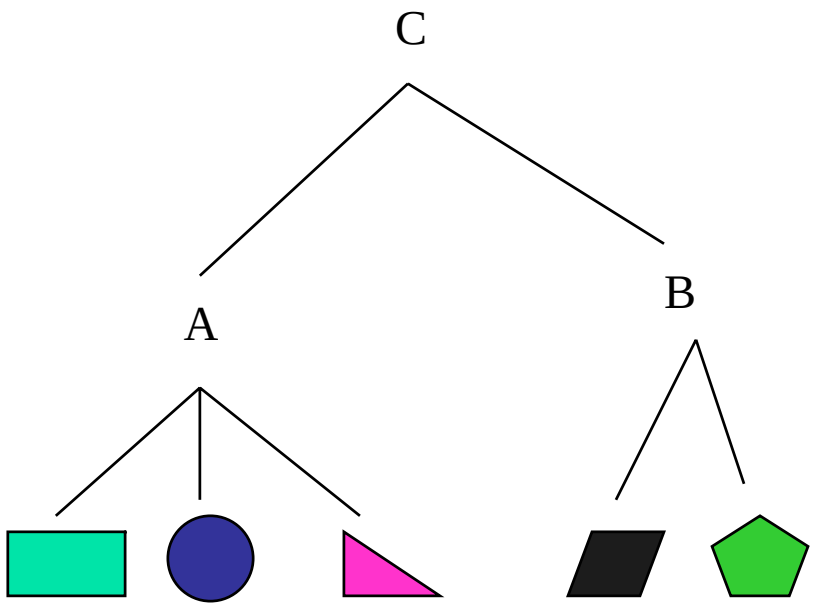
他のオブジェクトとの関係：属性，インスタンス変数

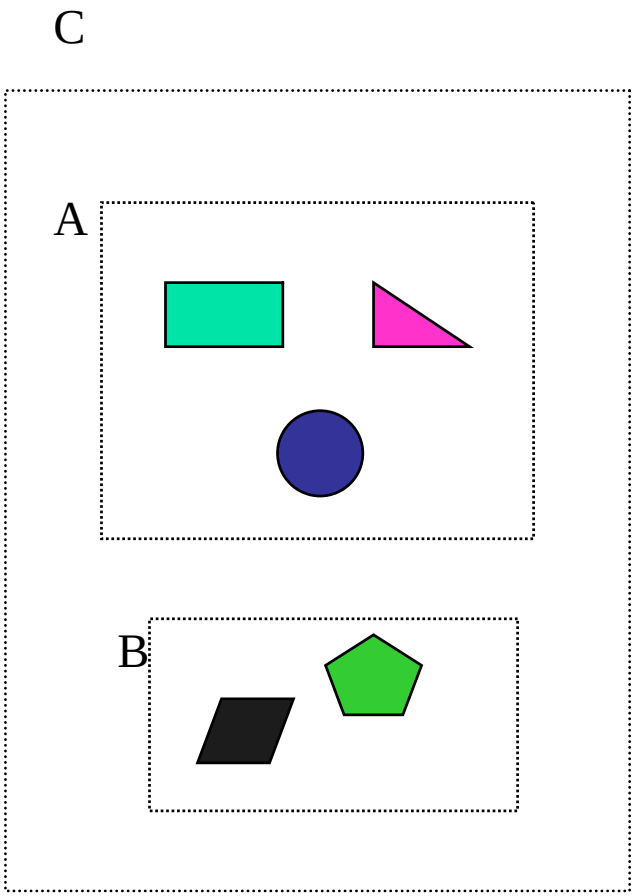
(例）図形のグループ化



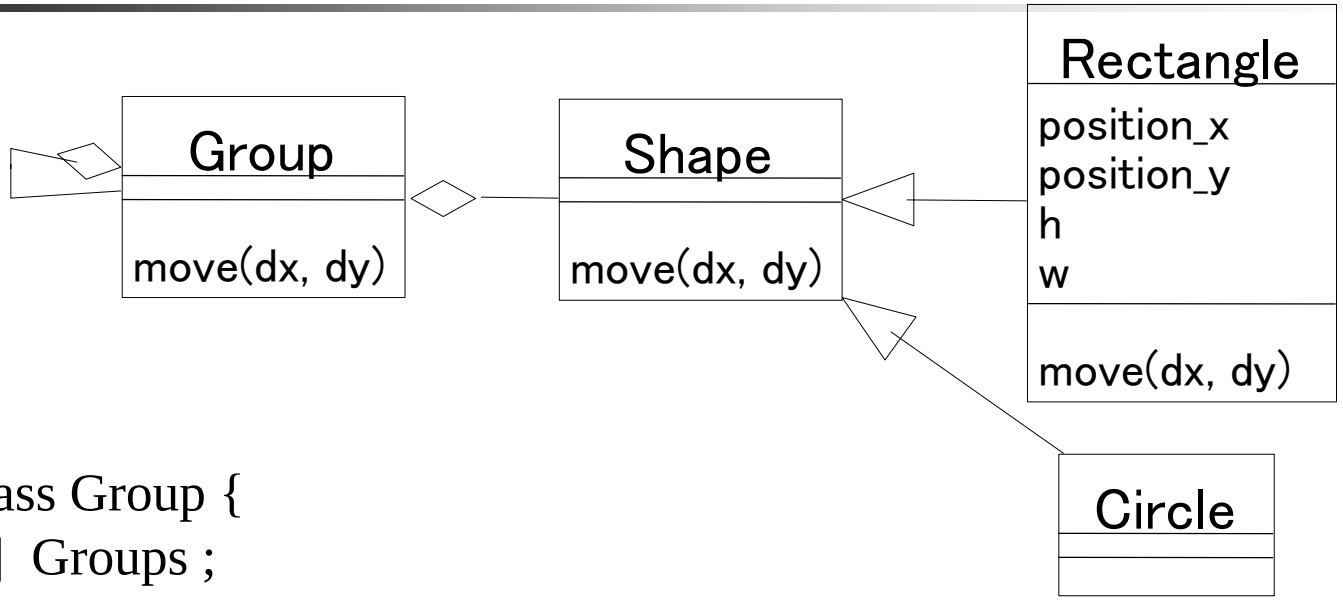
パターンへ（５）：Composite Pattern

グループ化したものも図形：階層構造





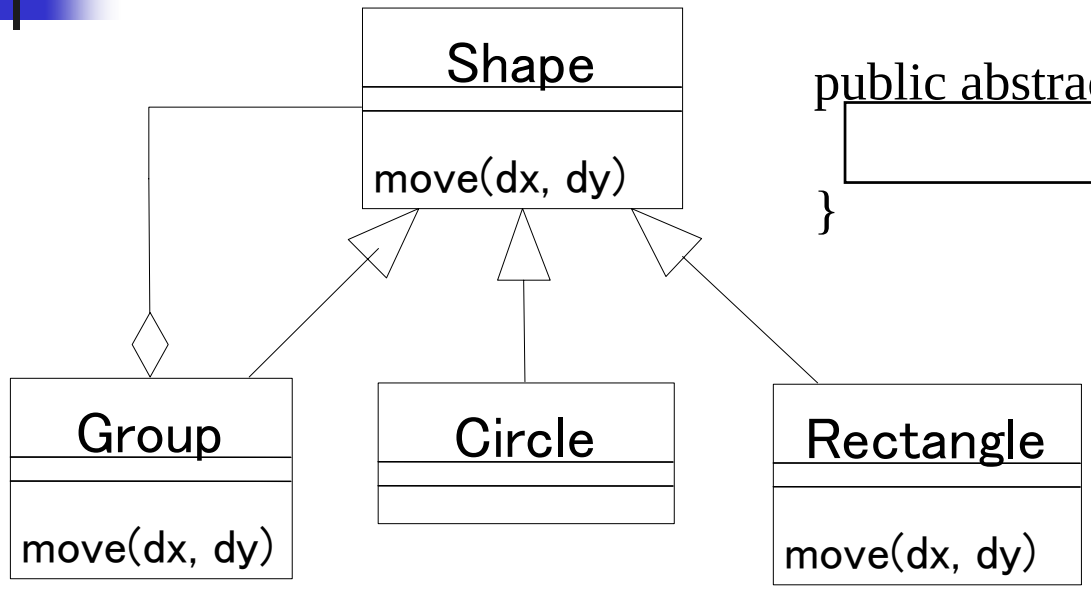
パターンへ（6）:Composite Pattern




```
public class Group {
 Group[ ]  Groups ;
 Shape[ ]  Shapes ;
 public void move(int dx, int dy) {
 }
}
```

```
public class Rectangle
      extends Shape{
  ...
  public void move(int dx,int dy) {
    position_x = position_x + dx ;
    position_y = position_y + dy ;
  }
}
```

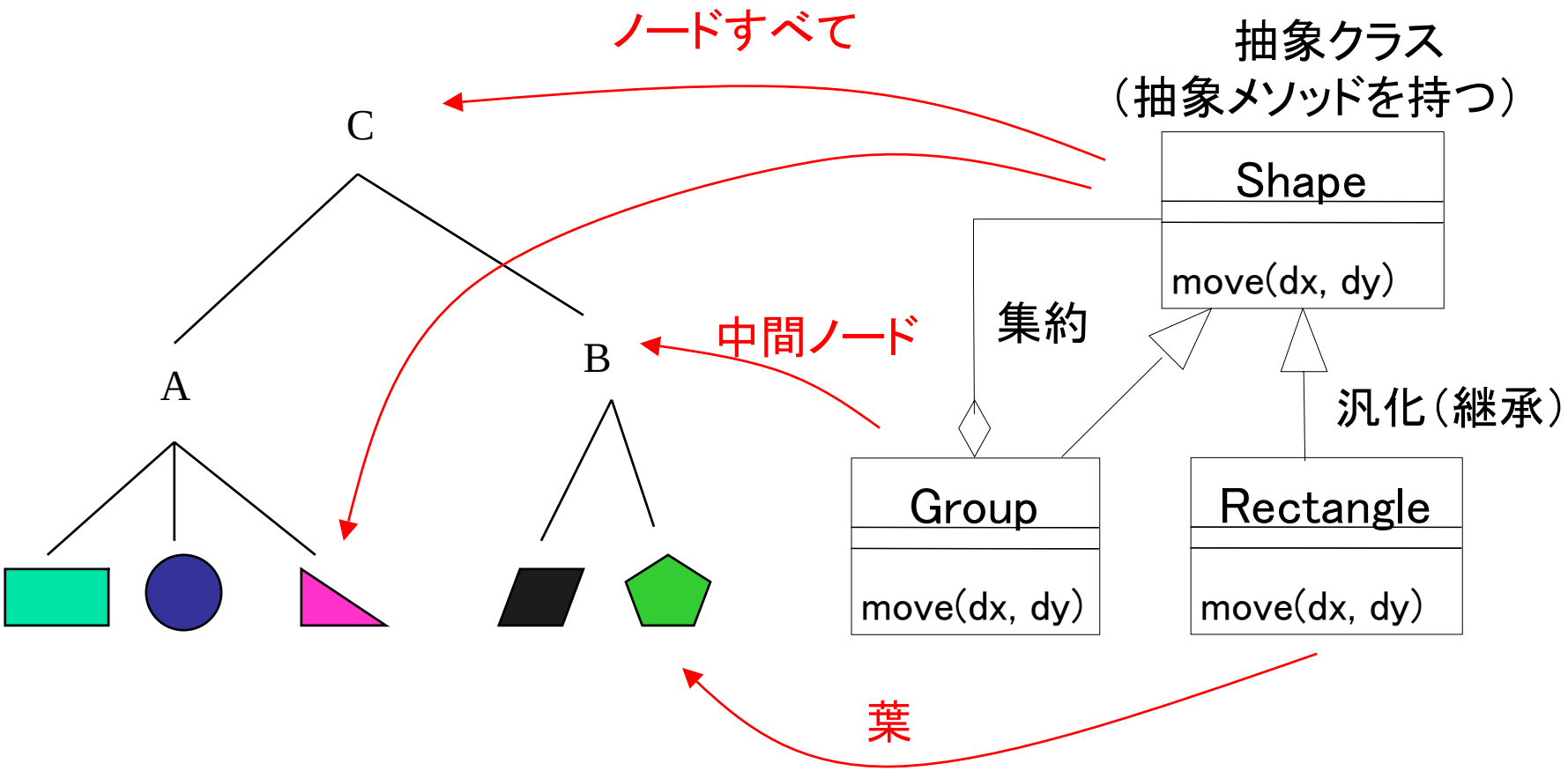
Composite Pattern (1)




```
public abstract class Shape {
                                        }
```

```
public class Group extends Shape {
  Shape[ ]  Shapes ;
 public void move(int dx, int dy) {
                                          }
}
```

Composite Pattern (2)



Composite Pattern (3)

オブジェクトが木構造（再帰構造）をしているとき