



Kế thừa, đa hình và
mẫu thiết kế



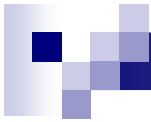
Tài liệu tham khảo

- Bruce Eckel, *Thinking in Patterns*
- Erich Gamma, *Design Patterns – Elements of Reusable Object-Oriented Software*



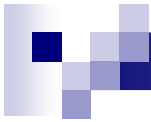
Mẫu thiết kế (Design Patterns)

- **Creational** - Thay thế cho khởi tạo tường minh, giảm phụ thuộc môi trường (platform)
- **Structural** - thao tác với các lớp không thay đổi được, giảm độ ghép nối và cung cấp các giải pháp thay thế kế thừa
- **Behavioral** - Che giấu cài đặt, che giấu thuật toán, cho phép thay đổi động cấu hình của đối tượng

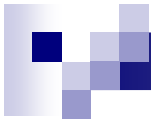


Singleton

- Cho phép khởi tạo duy nhất một đối tượng
- Ứng dụng trong điều phối tương tranh (điều khiển ngoại vi, quản lý CSDL, các luồng vào ra,...)



```
class Singleton {  
    private int i;  
  
    private static Singleton s = new Singleton(0);  
  
    private Singleton(int x) { i = x; }  
  
    public static Singleton getReference() {  
        return s;  
    }  
  
    public int getValue() { return i; }  
    public void setValue(int x) { i = x; }  
}
```



```
public class TestSingleton {  
    public static void main(String[] argv) {  
        Singleton sg;  
        sg = Singleton.getReference();  
        sg.setValue(10);  
        System.out.print(sg.getValue());  
  
        // Singleton s2 = new Singleton(0);  
    }  
}
```



Nhân bản đối tượng

- Có nhu cầu nhân bản các đối tượng
 - Khi truyền tham số để tránh sửa đổi đối tượng gốc
 - Sử dụng nhiều đối tượng “tương tự”
- Làm thế nào để nhân bản đối tượng mà không biết rõ kiểu (lớp) thực sự của nó?
 - Sử dụng copy constructor?
 - Sử dụng phương thức nhân bản?
 - Interface Cloneable và phương thức `clone()`




Ví dụ: nhân bản đối tượng

```
class GraphicTool {  
    Graphic list[];  
    ...  
    void add(Graphic g) { //add to list }  
  
    void copyPaste(i) {  
        // make a copy of list[i] and add to the list  
    }  
  
    ...  
}
```




Các lớp đều có copy constructor

```
class Graphic {  
    int x, y;  
    public Graphic(Graphic g) {  
        x = g.x;  
        y = g.y;  
    }  
    ...  
}  
class Circle extends Base {  
    int r;  
    public Circle(Circle c) {  
        super(c);  
        r = c.r;  
    }  
    ...  
}
```



```
class GraphicTool {  
    Graphic list[];
```

```
...
```

```
    void copyPaste(i) {  
        Graphic g = new Graphic(list[i]);  
        add(g);  
    }
```

```
...
```


```
}
```

```
    public class TestGraphicTool {  
        public static void main(String args[]) {  
            GraphicTool tool = new GraphicTool();  
            Point p = new Point(1, 1);  
            Circle c = new Circle(10, 10, 10);  
            tool.add(p);  
            tool.add(c);  
            tool.copyPaste(1);  
        }  
    }
```



Phương thức clone()

```
class Graphic {  
    int x, y;  
    Graphic(int m, int n) { x= m, y = n;}  
    public Graphic(Graphic g) { x = g.x; y = g.y; }  
    public Graphic clone() {  
        return new Graphic(this);  
    }  
}
```



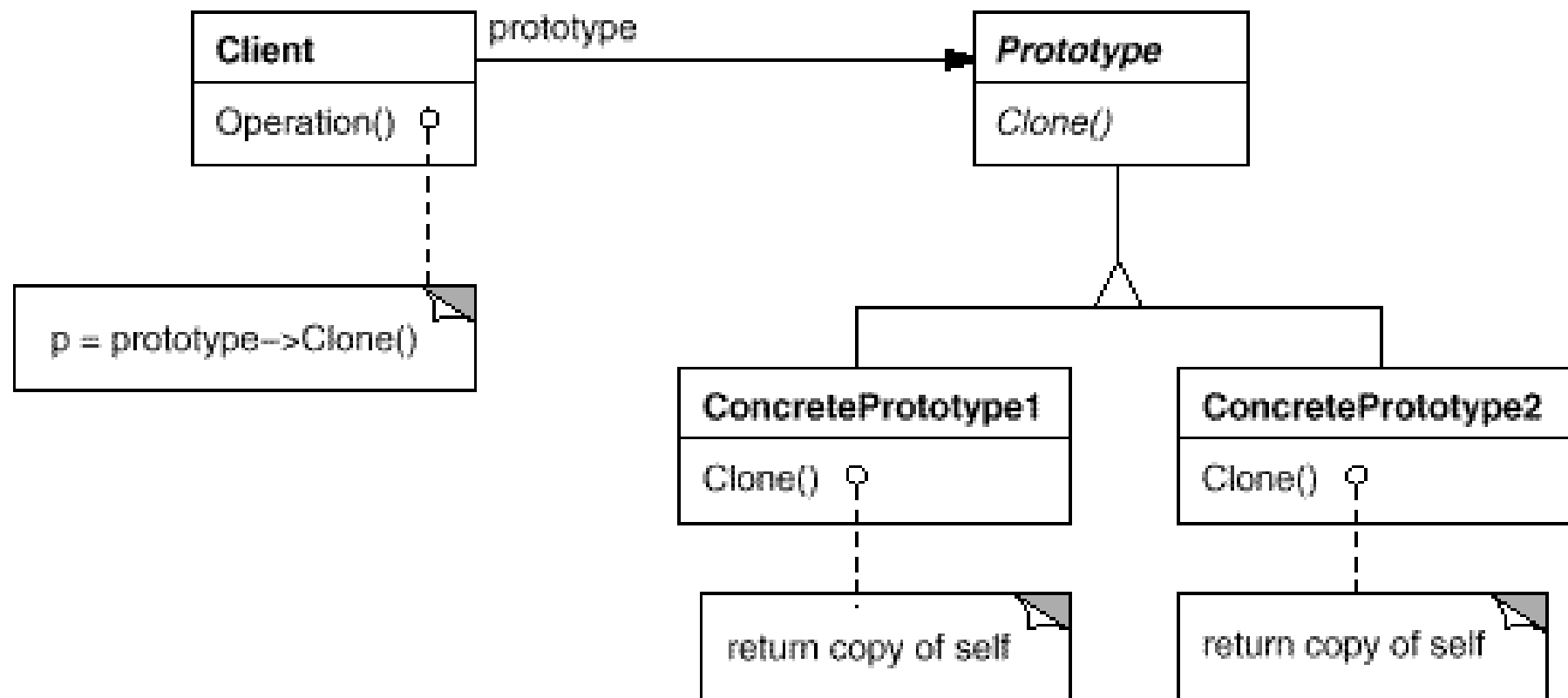
```
class Circle extends Graphic {  
    int r;  
    public Circle(Circle c) {  
        super(c);  
        r = c.r;  
    }  
  
    public Circle clone() {  
        return new Circle(this);  
    }  
}
```

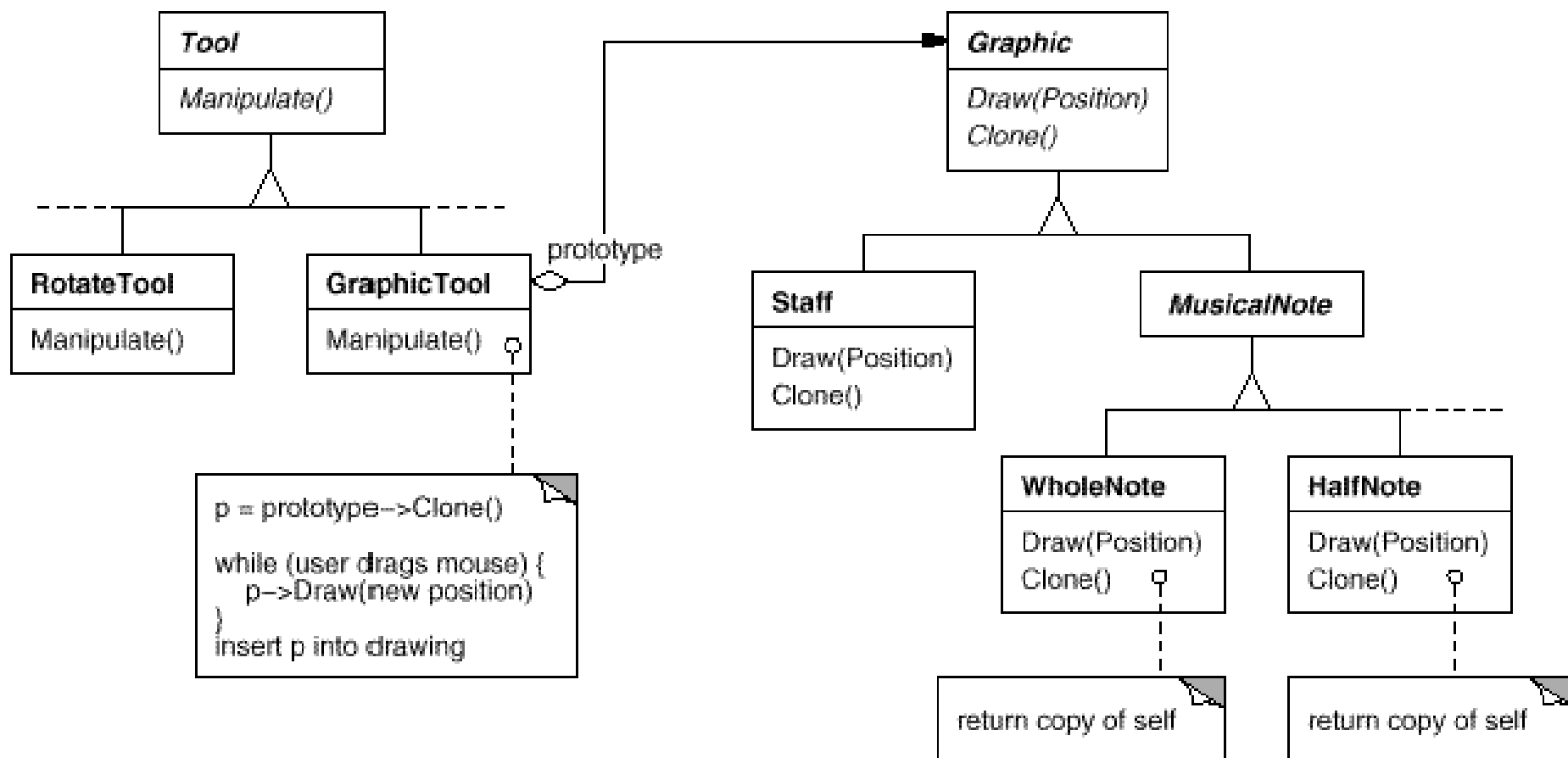
Local copy sử dụng clone()

```
class GraphicTool {  
    Graphic list[];  
    ...  
    void copyPaste(i) {  
        Graphic g = list[i].clone();  
        add(g);  
    }  
    ...  
}
```

```
public class TestGraphicTool {  
    public static void main(String args[]) {  
        GraphicTool tool = new GraphicTool();  
        Point p = new Point(1, 1);  
        Circle c = new Circle(10, 10, 10);  
        tool.add(p);  
        tool.add(c);  
        tool.copyPaste(1);  
    }  
}
```

Prototype





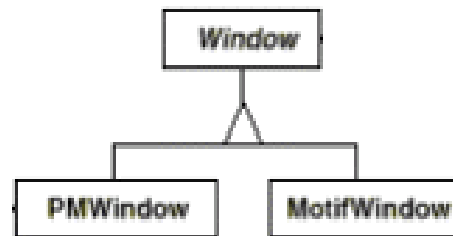


Abstract Factory

- Một chương trình cần có khả năng chọn việc sử dụng một trong một vài họ các lớp đối tượng
- Ví dụ, giao diện đồ họa cần chạy được trên một vài môi trường khác nhau
- Mỗi môi trường (platform) cung cấp một tập các lớp đồ họa riêng:
 - WinButton, WinScrollBar, WinWindow
 - MotifButton, MotifScrollBar, MotifWindow
 - pmButton, pmScrollBar, pmWindow

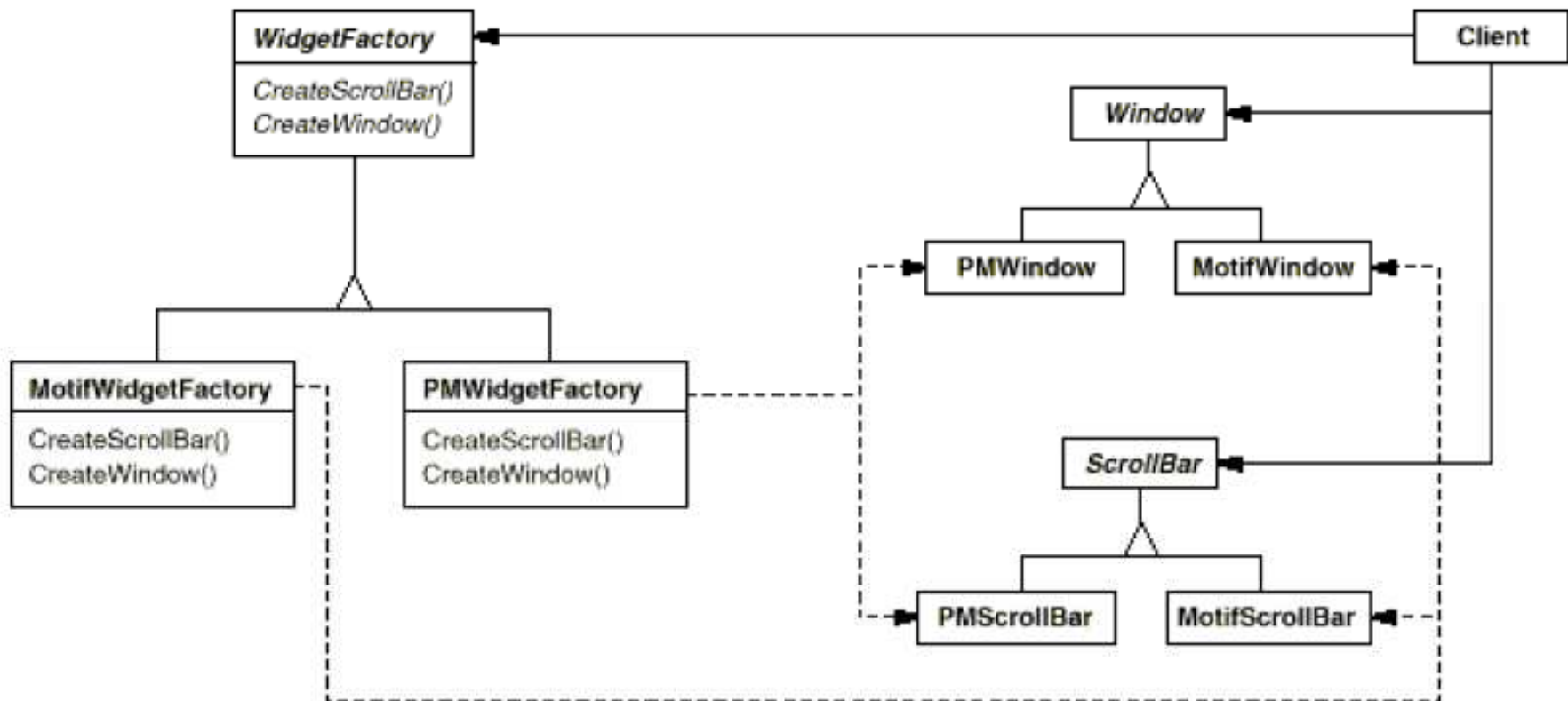
Yêu cầu

- Thống nhất thao tác với mọi đối tượng: button, window, ...
 - Dễ dàng - định nghĩa giao diện (interfaces):



- Thống nhất cách thức **tạo đối tượng**
- Dễ dàng thay đổi các họ lớp đối tượng
- Dễ dàng thêm họ đối tượng mới

Sơ đồ lớp





Giải pháp

- Định nghĩa Factory - lớp để tạo đối tượng:

```
abstract class WidgetFactory {  
    abstract Button makeButton(args);  
    abstract Window makeWindow(args);  
    // other widgets...  
}
```



Giải pháp (tt)

- Định nghĩa Factory chi tiết cho từng họ lớp đối tượng:

```
class WinWidgetFactory extends WidgetFactory
{
    public Button makeButton(args) {
        return new WinButton(args);
    }
    public Window makeWindow(args) {
        return new WinWindow(args);
    }
}
```

Giải pháp (tt)

- Chọn họ lớp muốn dùng:

```
WidgetFactory wf = new WinWidgetFactory();
```

- Khi cần đối tượng, không tạo trực tiếp mà thông qua “factory”:

```
Button b = wf.makeButton(args);
```

- Khi muốn thay đổi họ đối tượng - chỉ sửa **một vị trí** trong mã client!
- Thêm họ - thêm một factory, không ảnh hưởng tới mã đang tồn tại!



Ứng dụng

Dùng cho các phần mềm

- Chạy trên các hệ điều hành khác nhau
- Dùng các chuẩn look-and-feel khác nhau
- Dùng các giao thức truyền thông khác nhau



Composite

- Một chương trình cần thao tác với các đối tượng dù là đơn giản hay phức tạp một cách thống nhất
- Ví dụ, chương trình vẽ hình chứa đồng thời các đối tượng đơn giản (đoạn thẳng, hình tròn, văn bản) và đối tượng hợp thành (bánh xe = hình tròn + 6 đoạn thẳng).



Yêu cầu

- Thao tác với các đối tượng đơn giản/phức tạp một cách thống nhất - move, erase, rotate, set color
- Một vài đối tượng hợp thành được định nghĩa tĩnh (bánh xe) trong khi một vài đối tượng khác được định nghĩa động (do người dùng lựa chọn...)
- Đối tượng hợp thành có thể tạo ra *bằng các đối tượng hợp thành khác*
- Vì vậy cần một cấu trúc dữ liệu *thông minh*



Giải pháp

- Mọi đối tượng đơn giản kế thừa từ một giao diện chung, ví dụ *Graphic*:

```
class Graphic {  
    abstract void move(int x, int y);  
    abstract void setColor(Color c);  
    abstract void rotate(double angle);  
}
```

- Các lớp như *Line*, *Circle...* kế thừa *Graphic* và thêm các chi tiết (bán kính, độ dài,...)



Đối tượng hợp thành

Là một danh sách:

```
class Picture
{
    Graphics list[];
    public void add(Graphic g) {...}
    public void rotate(double angle) {
        for (int i=0; i<list.length; i++)
            list[i].rotate();
    }
}
```

Vậy một Picture có chứa một Picture được không?



Giải pháp (tt)

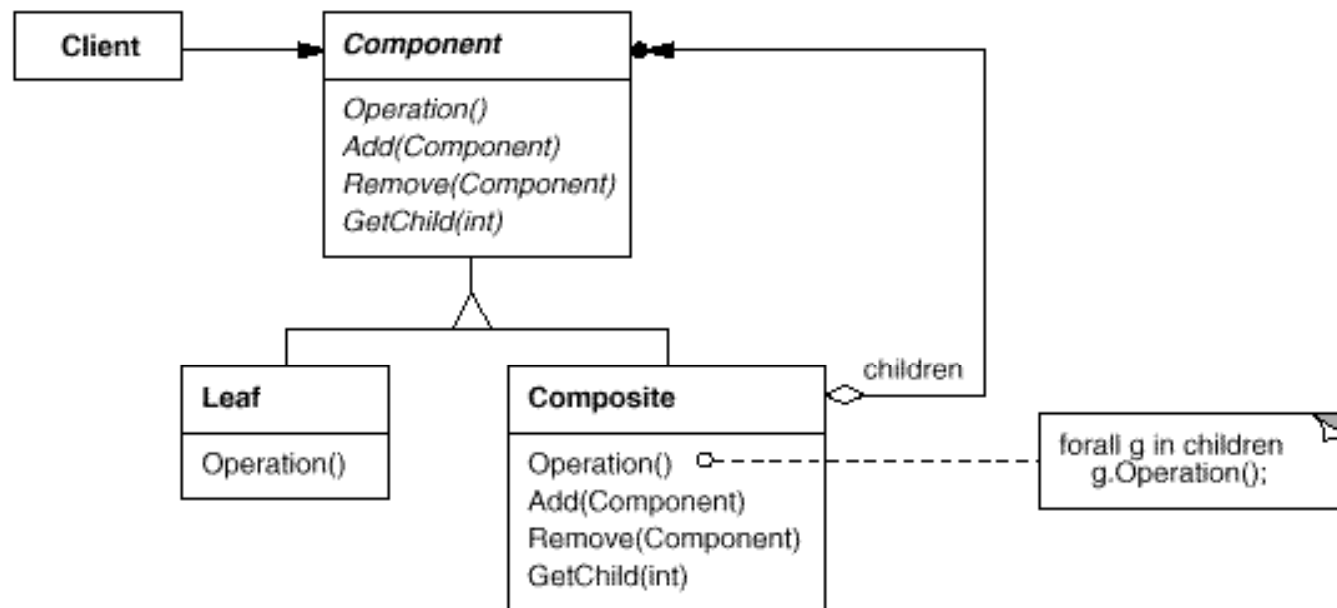
Vậy Picture cũng nên là Graphic

```
class Picture extends Graphic
{
    Graphics list[];
    public void add(Graphic g) {...}
    public void rotate(double angle) {
        for (int i=0; i<list.length; i++)
            list[i].rotate();
    }
}
```

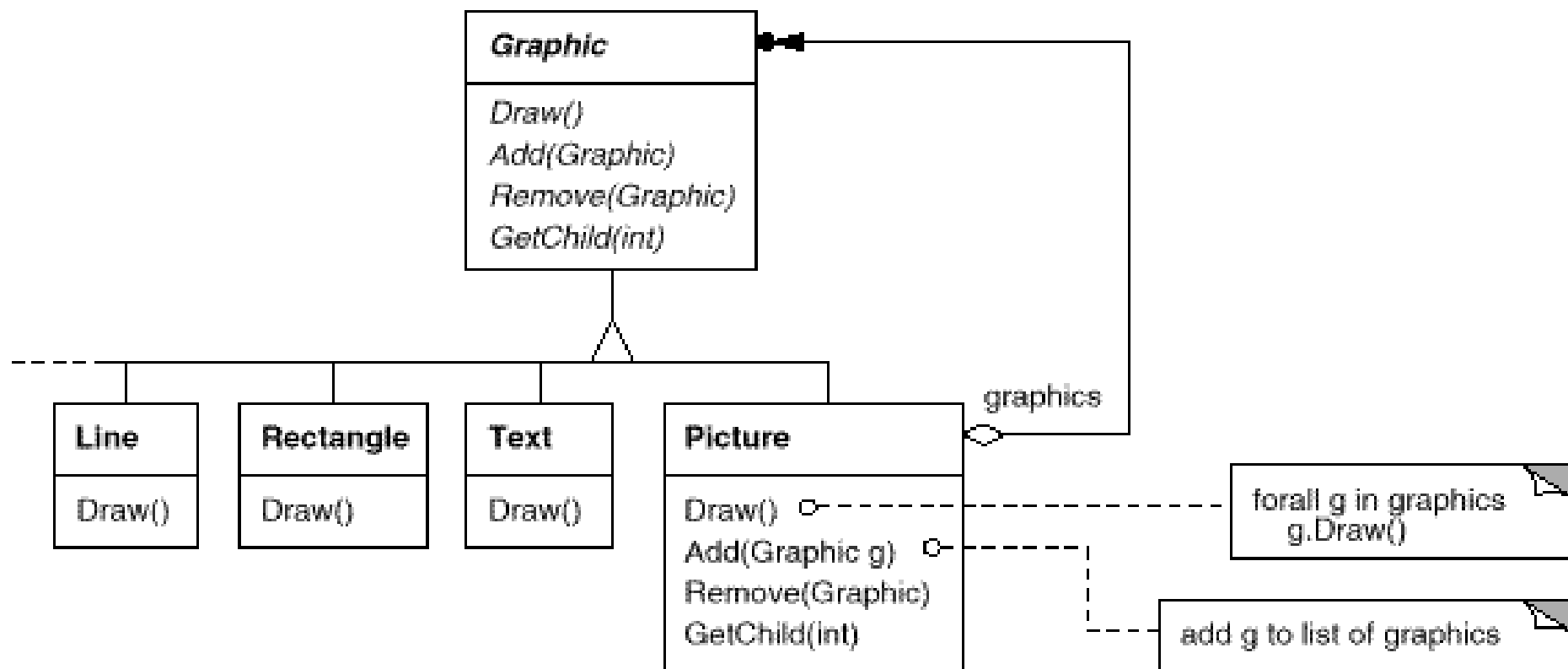
Giải pháp (tt)

- *Picture* là
 - một danh sách nên có *add()*, *remove()* và *count()*
 - *Graphic* nên còn có *rotate()*, *move()* và *setColor()*
- Các thao tác đó đối với một đối tượng hợp thành sử dụng một vòng lặp **‘for all’**
- Thao tác thực hiện ngay cả với trường hợp thành phần của Composite lại là một Composite khác - *cấu trúc dữ liệu dạng cây*
- Có khả năng giữ thứ tự của các thành phần

Sơ đồ lớp



- Kế thừa đơn
- Lớp cơ sở (root) chứa phương thức *add()*, *remove()*





Ví dụ

```
Picture pic1 = new Picture();  
pic1.add(new Line(0,0,100,100));  
pic1.add(new Circle(50,50,100));  
Picture pic2 = new Picture();  
pic2.add(new Text("Figure 1"));  
pic2.add(pic1);  
pic2.rotate(90);
```



Ứng dụng

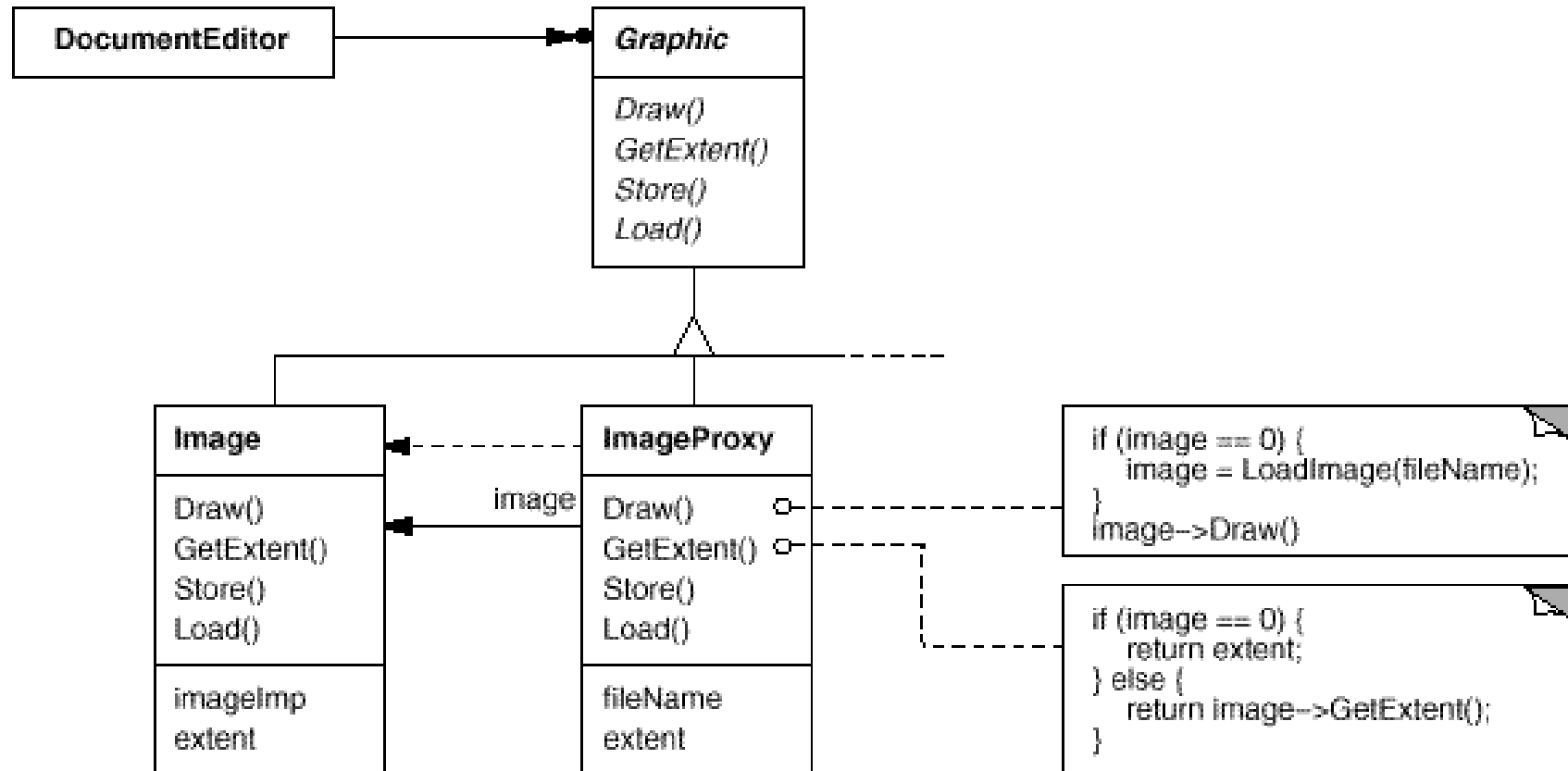
- Được dùng trong hầu hết các hệ thống HĐT
- Chương trình soạn thảo
- Giao diện đồ họa
- Cây phân tích cho biên dịch (một khối là một tập các lệnh/lời gọi hàm/các khối khác)



Proxy

- Các đối tượng có kích thước lớn, chỉ nên nạp vào bộ nhớ khi thực sự cần thiết; hay các đối tượng ở vùng địa chỉ khác (remote objects)
- Ví dụ: Xây dựng một trình soạn thảo văn bản có nhúng các đối tượng Graphic
 - Vấn đề đặt ra: Việc nạp các đối tượng Graphic phức tạp thường rất tốn kém, trong khi văn bản cần được mở nhanh
 - Giải pháp: sử dụng ImageProxy

Sơ đồ lớp





Áp dụng

- Proxy được sử dụng khi nào cần thiết phải có một tham chiếu thông minh đến một đối tượng hơn là chỉ sử dụng một con trỏ đơn giản
 - cung cấp đại diện cho một đối tượng ở một không gian địa chỉ khác (remote proxy).
 - trì hoãn việc tạo ra các đối tượng phức tạp (virtual proxy).
 - quản lý truy cập đến đối tượng có nhiều quyền truy cập khác nhau (protection proxy).
 - smart reference



Strategy

- Chương trình cần chuyển đổi *động* giữa các thuật toán
- Ví dụ, chương trình soạn thảo sử dụng vài thuật toán hiển thị với các hiệu ứng/lợi ích khác nhau



Yêu cầu

- Thuật toán phức tạp và sẽ không có lợi khi cài đặt chúng trực tiếp trong lớp sử dụng chúng
 - ví dụ: việc cài thuật toán hiển thị vào lớp *Document* là không thích hợp
- Cần thay đổi động giữa các thuật toán
- Dễ dàng thêm thuật toán mới



Giải pháp

- Định nghĩa lớp trừu tượng để biểu diễn thuật toán:

```
class Renderer {  
    abstract void render(Document d);  
}
```

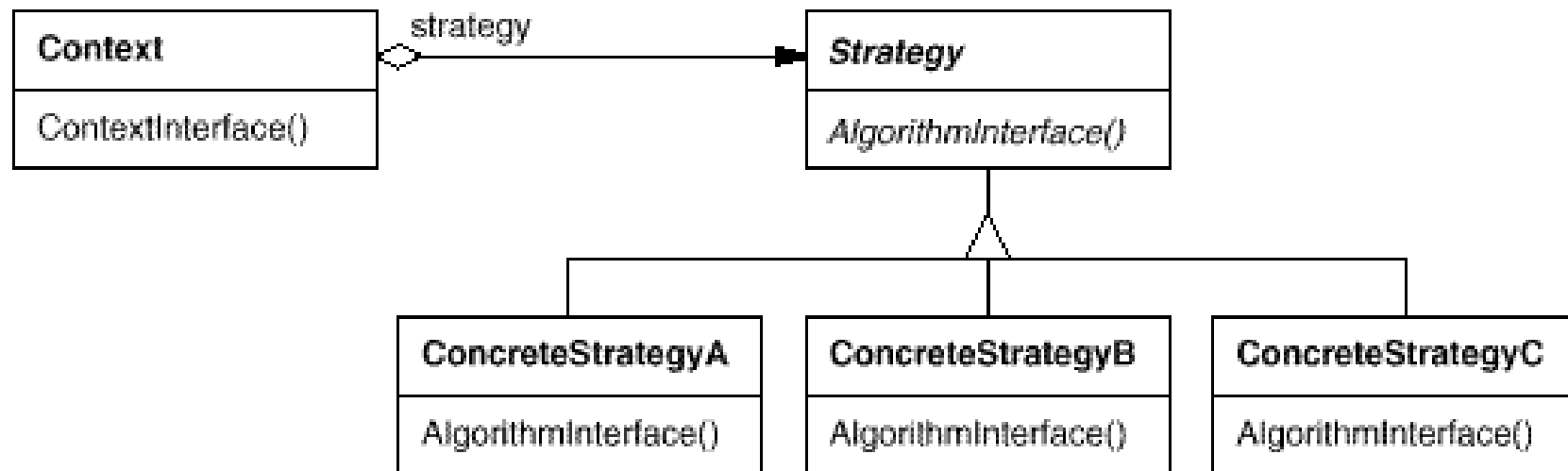
- Mỗi thuật toán là một lớp dẫn xuất
`FastRenderer, TexRenderer, ...`



Thay đổi động thuật toán

```
class Document {  
    render() { renderer.render(this); }  
    setRenderer(Renderer r) { renderer = r(); }  
    private Renderer renderer;  
  
    ...  
}  
  
...  
Document d = new Document();  
  
...  
d.setRenderer(new FastRenderer());  
d.render();  
d.setRenderer(new TextRenderer());  
d.render();
```

Sơ đồ lớp





Ứng dụng

- Chương trình vẽ/soạn thảo
- Tối ưu biên dịch
- Chọn lựa các thuật toán heuristic khác nhau (trò chơi...)
- Lựa chọn các phương thức quản lý bộ nhớ khác nhau



Một số mẫu khác

- Façade
- Template method
- Iterator



Tổng kết

- Mẫu thiết kế dựa trên sự vận dụng ở mức cao về kế thừa và đa hình
- Mẫu thiết kế tăng tính mở, khả năng sử dụng lại
- Cần biết vận dụng mẫu thiết kế
 - Qui bài toán về mẫu thiết kế chuẩn
- Cần biết kết hợp các mẫu thiết kế khác nhau