



Nguyên lý thiết kế và mẫu thiết kế



Nội dung

- Thiết kế module
- Chất lượng thiết kế
- Độ đo thiết kế tốt
- Khái niệm về mẫu thiết kế



Tài liệu tham khảo

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

Thiết kế module

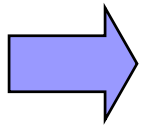
Dựa trên quan điểm "chia để trị"

C: độ phức tạp

$$C(p1 + p2) > C(p1) + C(p2)$$

E: nỗ lực thực hiện

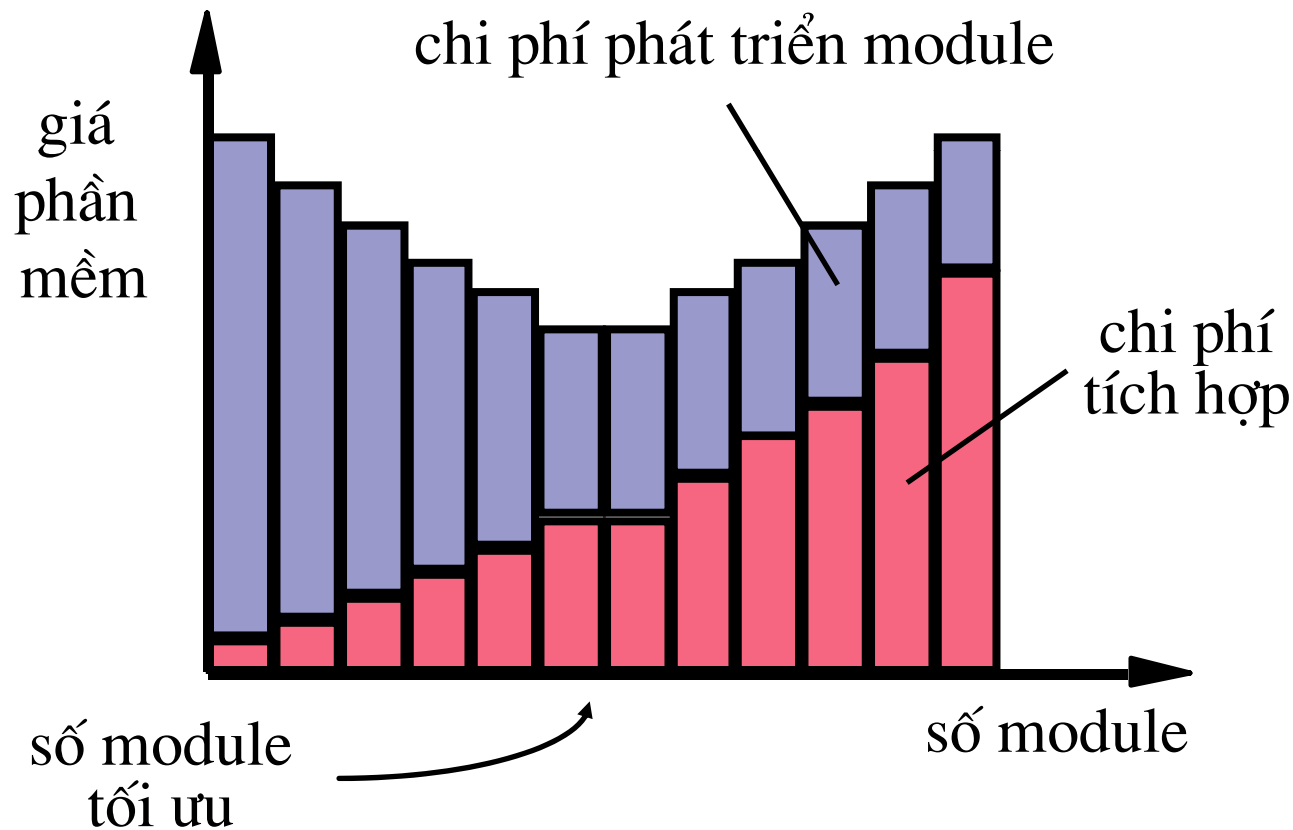
$$E(p1 + p2) > E(p1) + E(p2)$$



- giảm độ phức tạp
- cục bộ, dễ sửa đổi
- có khả năng phát triển song song
- dễ sửa đổi, dễ hiểu nên dễ tái sử dụng

Số lượng module


Cần xác định số môđun tối ưu





Chất lượng = Che giấu thông tin

- Sử dụng module thông qua các *giao diện*
 - tham số và giá trị trả lại
- Không cần biết cách thức cài đặt thực tế
 - thuật toán
 - cấu trúc dữ liệu
 - giao diện ngoại lai (các mô đun thứ cấp, thiết bị vào/ra)
 - tài nguyên hệ thống



Che giấu thông tin: lý do

- Giảm hiệu ứng phụ khi sửa đổi module
- Giảm sự tác động của thiết kế tổng thể lên thiết kế cục bộ
- Nhấn mạnh việc trao đổi thông tin thông qua giao diện
- Loại bỏ việc sử dụng dữ liệu dùng chung
- Hướng tới sự đóng gói chức năng - thuộc tính của thiết kế tốt

Tạo ra các sản phẩm phần mềm tốt hơn



Chất lượng thiết kế

- Phụ thuộc bài toán, không có phương pháp tổng quát
- Một số độ đo
 - Coupling: mức độ ghép nối giữa các module
 - Cohesion: mức độ liên quan lẫn nhau của các thành phần bên trong một module
 - Understandability: tính hiểu được
 - Adaptability: tính thích nghi được



Coupling and Cohesion

■ Coupling (ghép nối)

- độ đo sự liên kết (trao đổi dữ liệu) giữa các mô đun
- ghép nối chặt chẽ thì khó hiểu, khó sửa đổi (thiết kết tồi)

■ Cohesion (kết dính)

- độ đo sự phụ thuộc lẫn nhau của các thành phần trong một module
- kết dính cao thì tính cục bộ cao (độc lập chức năng); dễ hiểu, dễ sửa đổi



Understandability

Tính hiểu được

- Ghép nối lỏng lẻo
- Kết dính cao
- Được lập tài liệu
- Thuật toán, cấu trúc dễ hiểu



Thiết kế hướng đối tượng

- Thiết kế hướng đối tượng hướng tới chất lượng thiết kế tốt
 - ☐ đóng gói, che giấu thông tin
 - ☐ là các thực thể hoạt động độc lập
 - ☐ trao đổi dữ liệu qua thông điệp
 - ☐ có khả năng kế thừa
 - ☐ cục bộ, dễ hiểu, dễ tái sử dụng



Adaptability

Tính thích nghi được

■ Hiểu được

- ☐ sửa đổi được, tái sử dụng được

■ Tự chứa

- ☐ không sử dụng thư viện ngoài
- ☐ *mâu thuẫn với xu hướng tái sử dụng*



Adaptability (2)

- Các chức năng cần được thiết kế sao cho dễ dàng mở rộng mà không cần sửa các mã đã có (Open closed principle)
- Trừu tượng hóa là chìa khóa để giải quyết vấn đề này
 - các chức năng trừu tượng hóa thường bất biến
 - các lớp dẫn xuất cài đặt các giải pháp cụ thể
 - sử dụng đa hình
- Mẫu thiết kế: là thiết kế chuẩn cho các bài toán thường gặp



Mẫu thiết kế (Design Patterns)

- **Creational** - Thay thế cho khởi tạo tường minh, ngăn ngừa 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

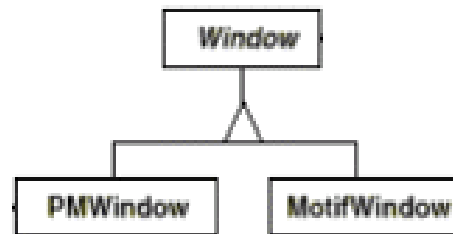


Abstract Factory

- 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 nên 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



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:

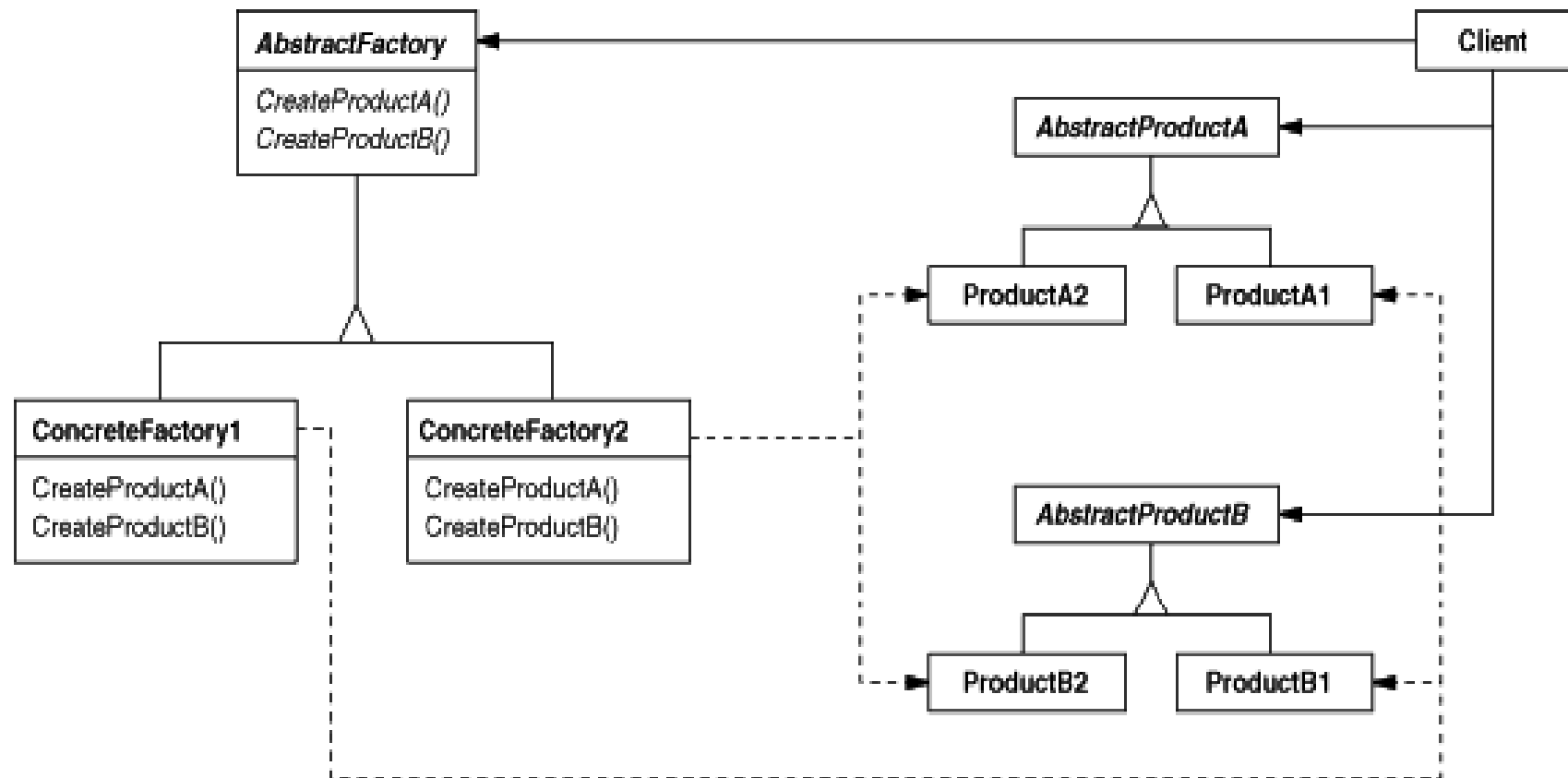
```
WidgetFactory wf = new WinWidgetFactory();
```

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

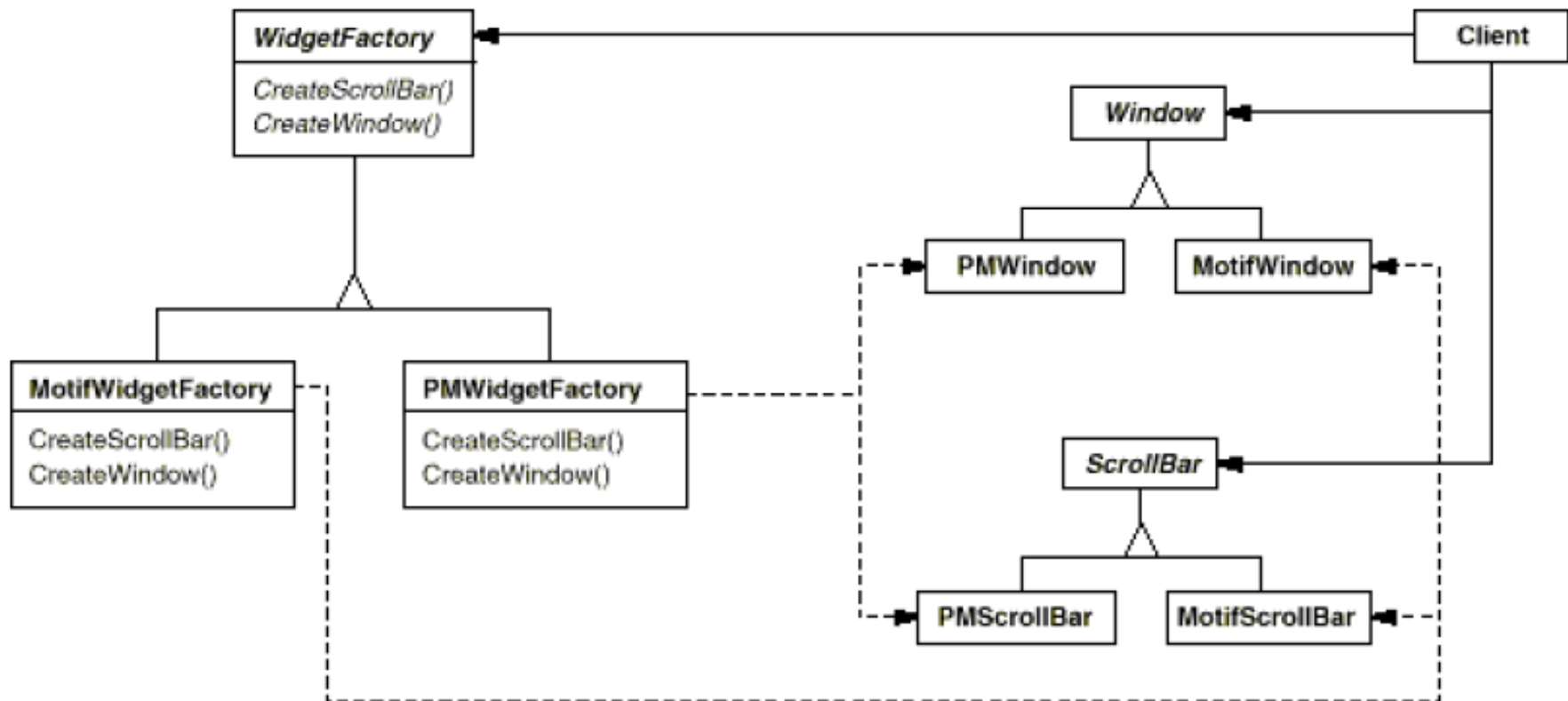
```
Button b = wf.makeButton(args);
```

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

Sơ đồ lớp



Sơ đồ lớp





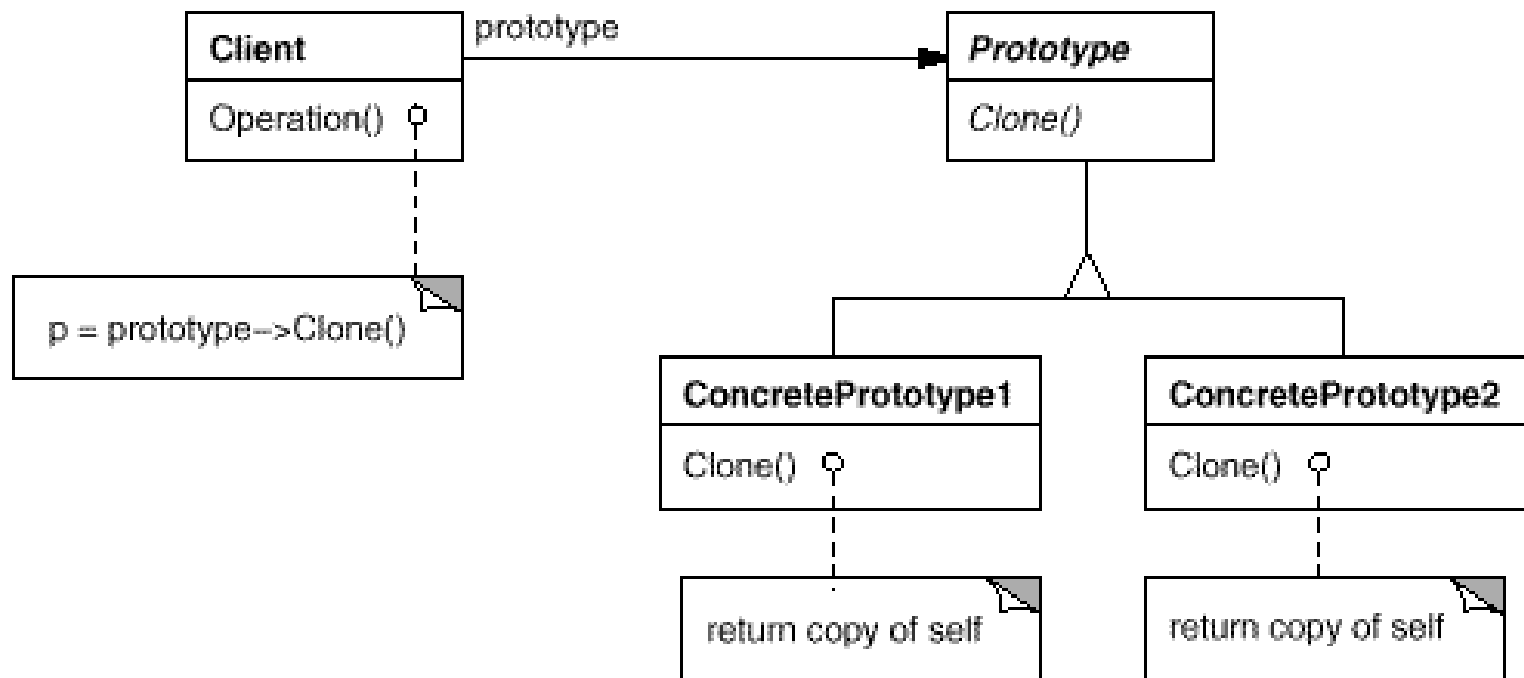
Ứng dụng

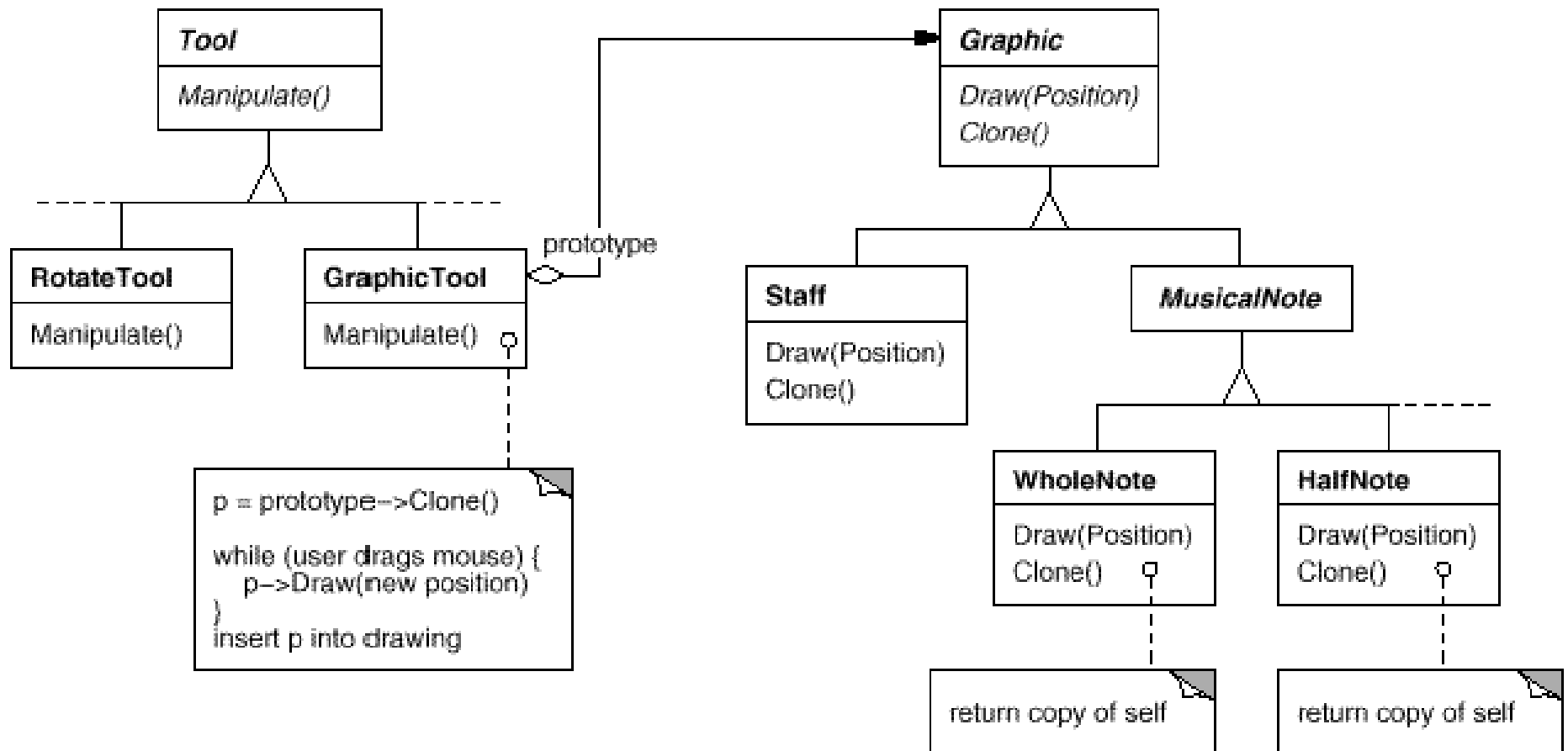
Dùng cho các phần mềm

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

Prototype

Nhân bản đối tượng mà không cần biết lớp của nó







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,...)



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,...)



Giải pháp (tt)

- Lớp dưới đây cũng là một lớp dẫn xuất:

```
class Picture extends Graphic
{
    Graphics list[];
    ...
    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



Giải pháp (tt)

- Ví dụ tạo một đối tượng hợp thành:

```
Picture pic;
```

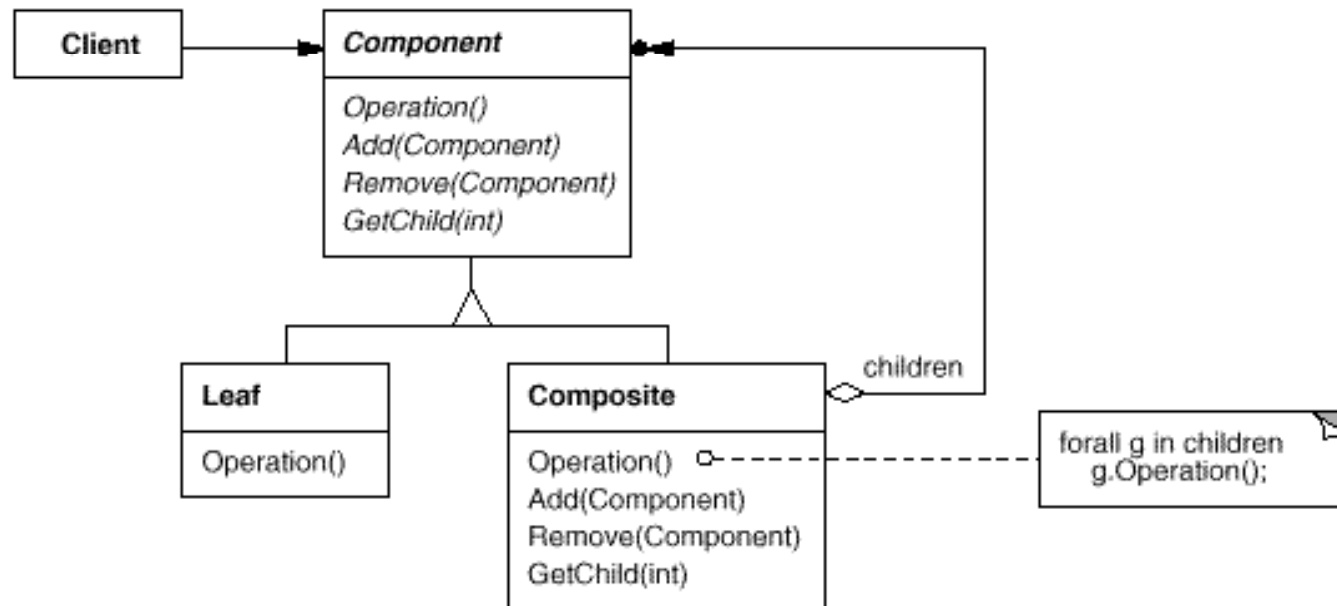
```
pic = new Picture();
```

```
pic.add(new Line(0,0,100,100));
```

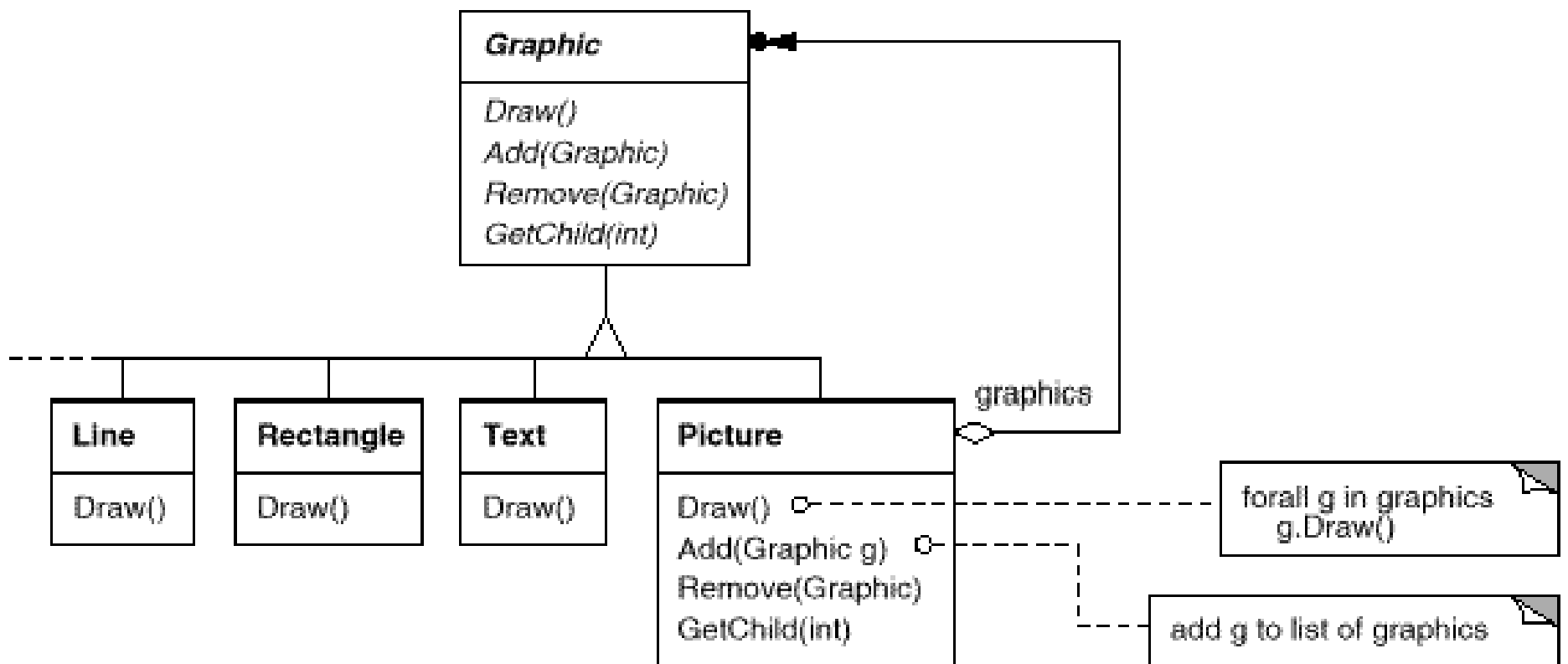
```
pic.add(new Circle(50,50,100));
```

```
pic.rotate(90);
```

Sơ đồ lớp



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





Ứng dụng

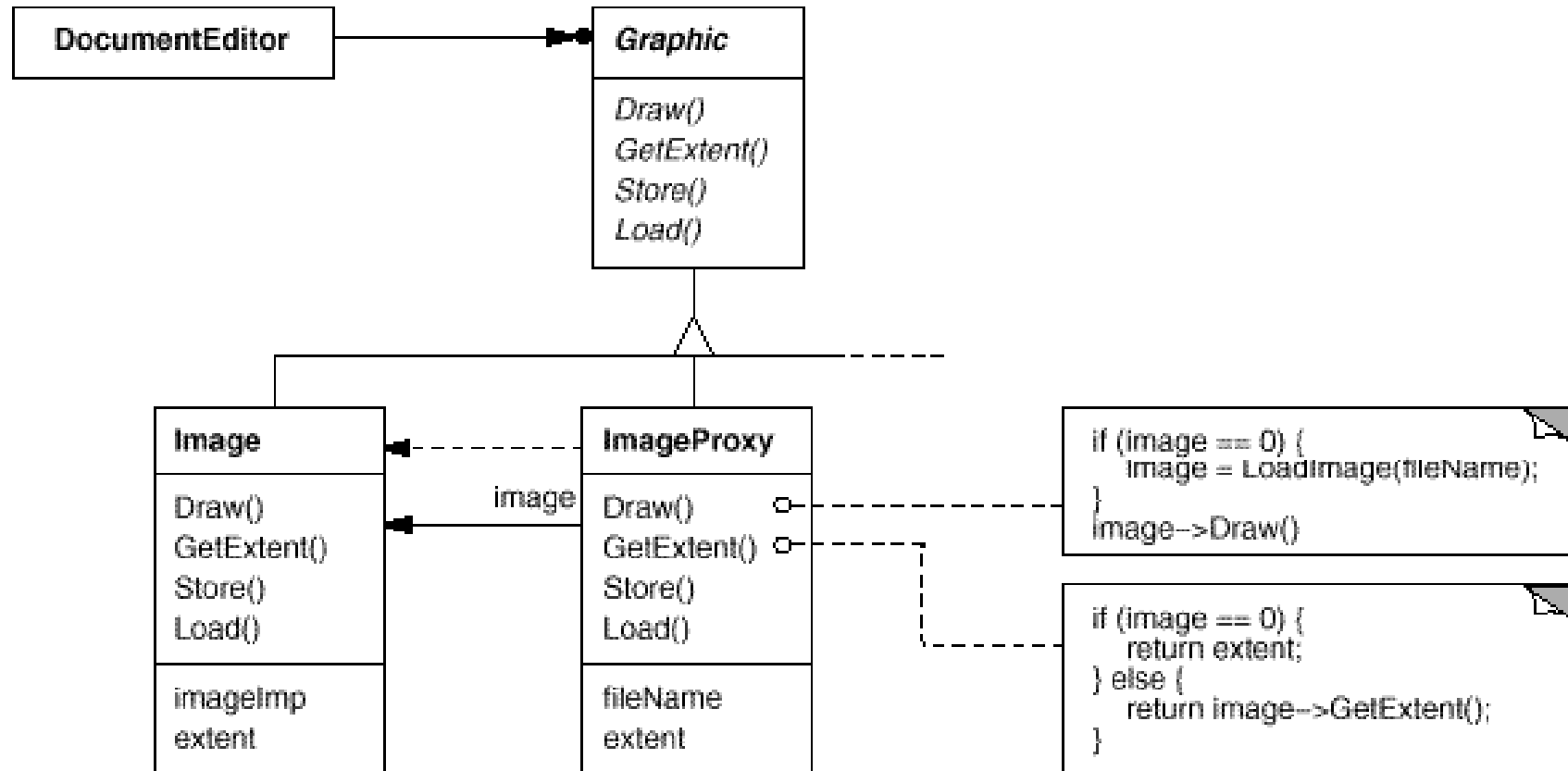
- Được dùng trong hầu hết các hệ thống HĐT
- Chương trình soạn thảo
- Giao diện đồ họa
- Cây phân tích cho biên dịch (một khối là một tập các lệnh/lời gọi hàm/các khối khác)



Proxy Pattern

- 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, ...

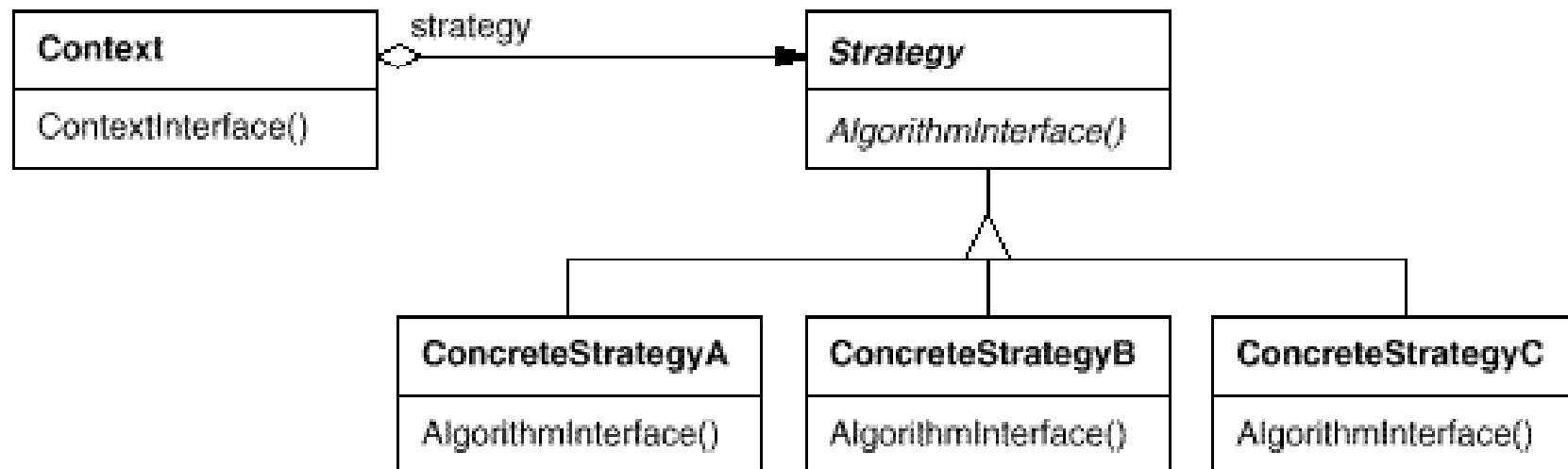


Giải pháp (tt)

- Đối tượng "document" tự chọn thuật toán vẽ:

```
class Document {  
    render() {  
        renderer.render(this);  
    }  
    setFastRendering() {  
        renderer = new FastRenderer();  
    }  
    private Renderer renderer;  
}
```

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