BỘ MÔN CÔNG NGHỆ PHẦN MỀM
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Bài 08-3. Nguyên lý thiết kế và mẫu thiết kế (design pattern)

Nội dung

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

Tài liệu tham khảo

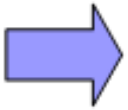
- Bruce Eckel, *Thinking in Patterns*
- Erich Gamma, *Design Patterns – Elements of Reusable Object-Oriented Software*

Thiết kế module

Dựa trên quan điểm "chia để trị"

C: độ phức tạp $C(p1 + p2) > C(p1) + C(p2)$

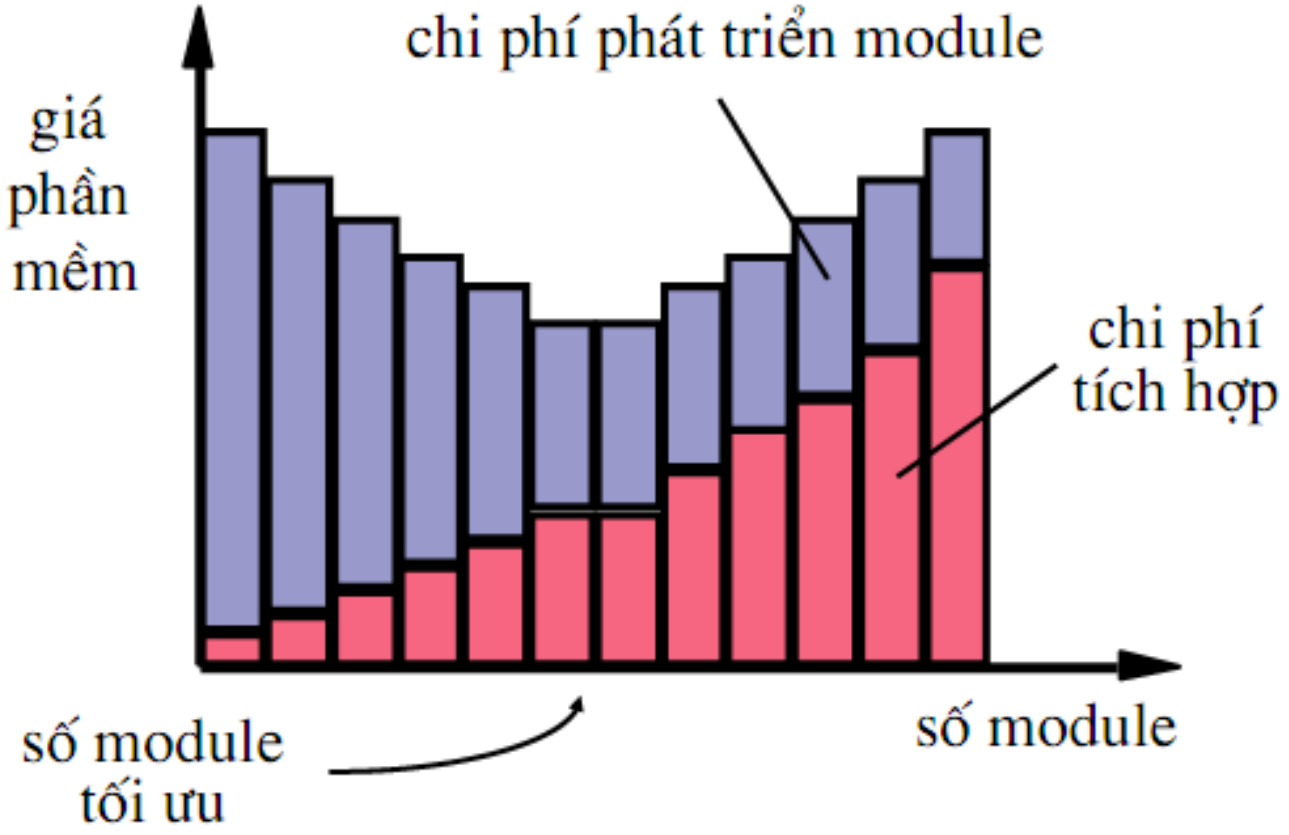
E: nỗ lực thực hiện $E(p1 + p2) > E(p1) + E(p2)$



- 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

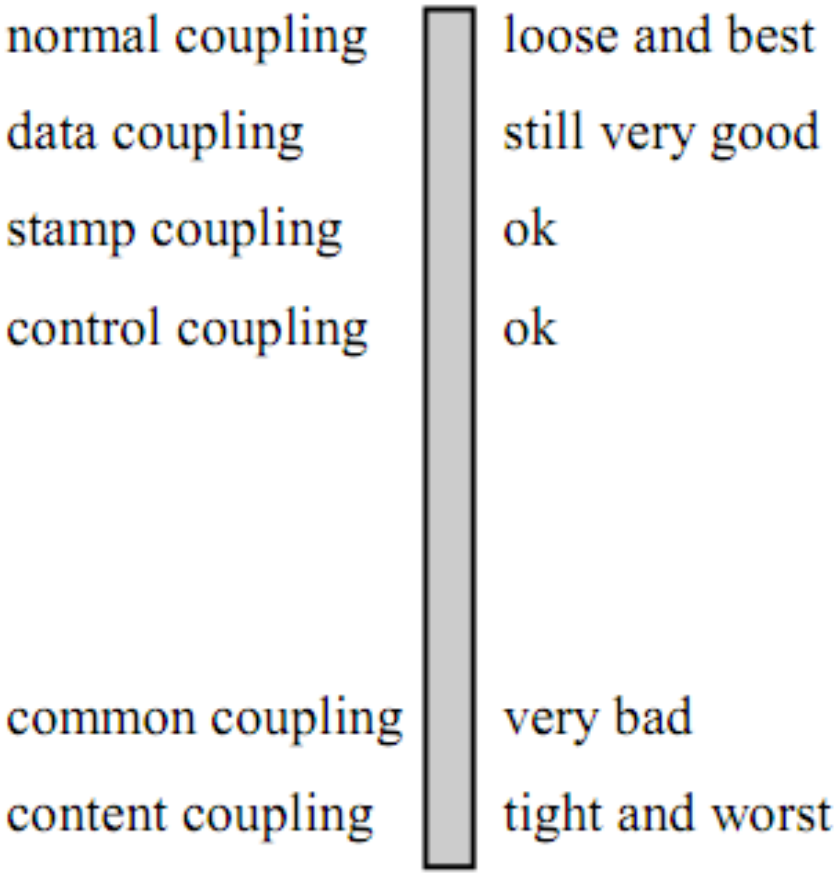
- Phụ thuộc bài toán, không có phương pháp tổng quát
- Một số độ đo
  - Coupling: mức độ ghép nối giữa các module
  - Cohesion: mức độ liên quan lẫn nhau của các thành phần bên trong một module
  - Understandability: tính hiểu được
  - Adaptability: tính thích nghi được

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

Coupling

- mức độ quan hệ của các module
- module nên ghép nối lỏng lẻo
- càng lỏng lẻo càng dễ sửa đổi thiết kế

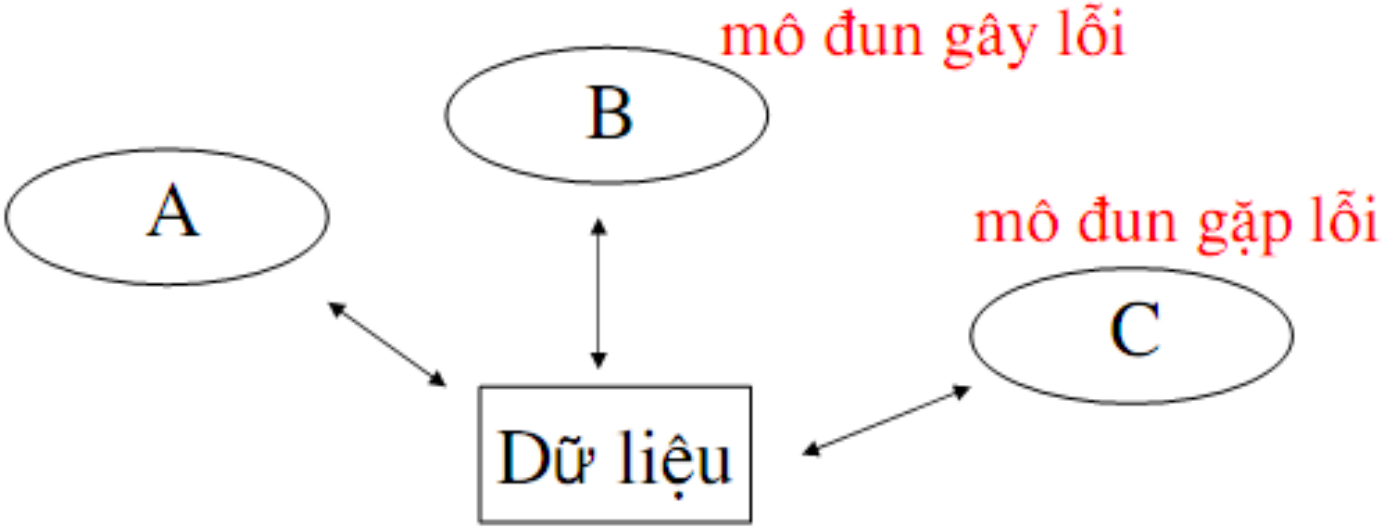


Ghép nối nội dung (content coupling)

- Là trường hợp xấu nhất
- Các module dùng lẫn dữ liệu của nhau
  - các ngôn ngữ bậc thấp không có biến cục bộ
  - lạm dụng lệnh Goto

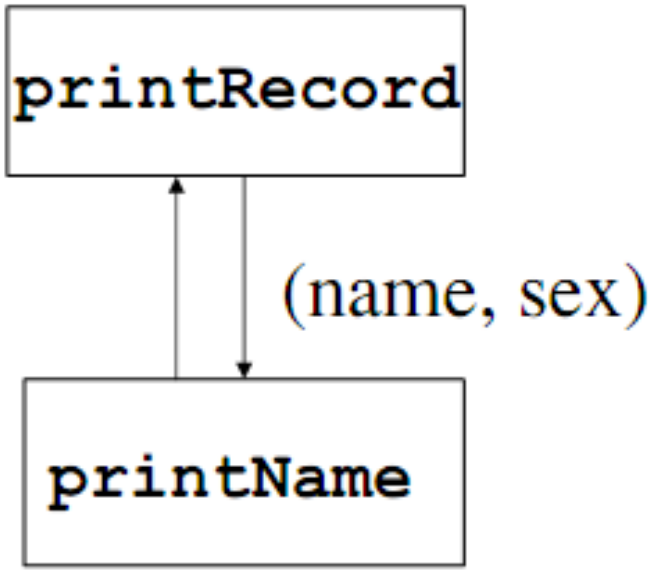
Ghép nối chung (common coupling)

- Các module trao đổi dữ liệu thông qua biến tổng thể
- Lỗi của module này có thể ảnh hưởng đến hoạt động của module khác
- Khó sử dụng lại các module



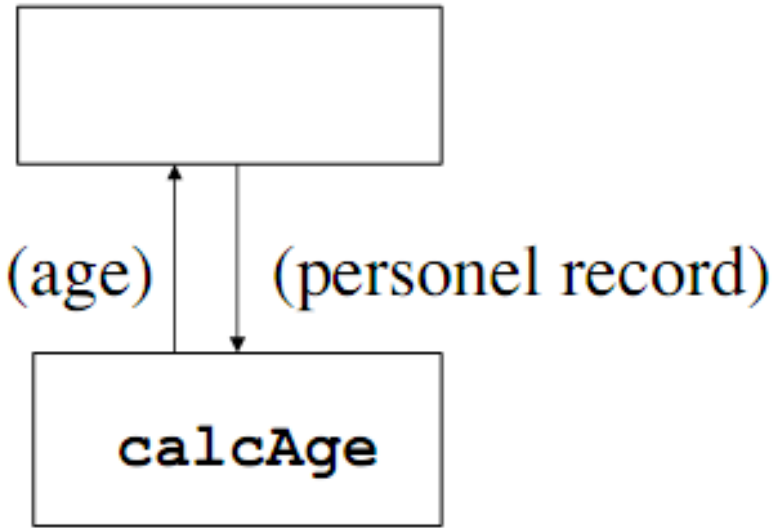
Ghép nối điều khiển (control coupling)

- Các module trao đổi thông tin điều khiển
- Làm cho thiết kế khó hiểu, khó sửa đổi, dễ nhầm



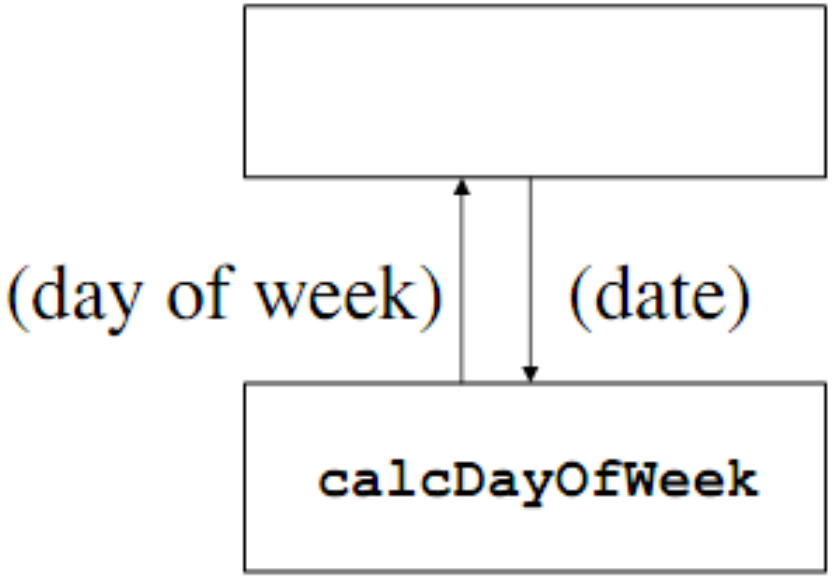
Ghép nối nhãn (stamp coupling)

- Các module trao đổi thừa thông tin
- Module có thể thực hiện chức năng ngoài ý muốn
- Làm giảm tính thích nghi



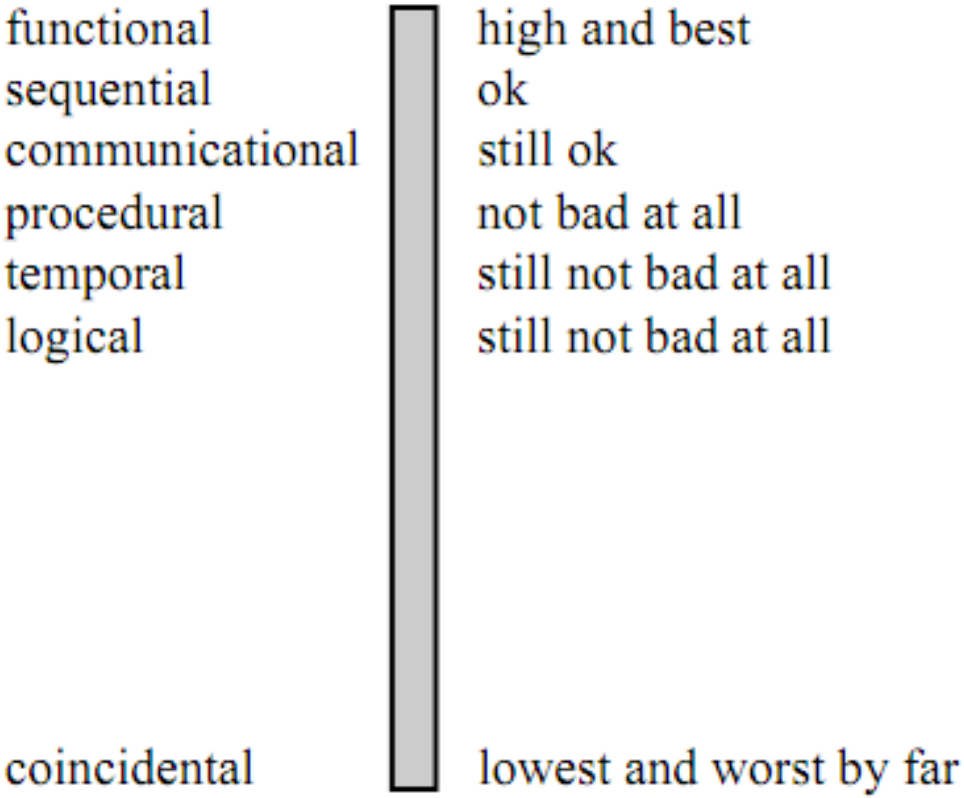
Ghép nối dữ liệu (data coupling)

- Truyền dữ liệu qua tham số
- Nhận kết quả qua tham số và giá trị trả lại



Cohesion

- mỗi module chỉ nên thực hiện một chức năng
- mọi thành phần nên tham gia thực hiện chức năng đó



Các loại kết dính

- Kết dính gom góp (coincidental cohesion)
  - các thành phần không liên quan đến nhau
- Kết dính lô gic (logical cohesion)
  - các thành phần làm chức năng lô gic tương tự
  - vd: hàm xử lý lỗi chung
- Kết dính thời điểm (temporal cohesion)
  - các thành phần hoạt động cùng thời điểm
  - vd: hàm khởi tạo (đọc dữ liệu, cấp phát bộ nhớ...)

Các loại kết dính

- Kết dính thủ tục (procedural cohesion)
  - các thành phần tạo có một thứ tự xác định
  - vd: tính lương cơ bản, tính phụ cấp, tính bảo hiểm

- Kết dính truyền thông (communicational cohesion)
  - các thành phần truy cập cùng dữ liệu
  - vd: thống kê (tính max, min, mean, variation...)

Các loại kết dính

- Kết dính tuần tự (sequential cohesion)
  - output của một thành phần là input của thành phần tiếp theo
  - vd: ảnh mầu -> đen trắng -> ảnh nén
- Kết dính chức năng (functional cohesion)
  - các thành phần cùng góp phần thực hiện một chức năng
  - vd: sắp xếp

Understandability

- 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 dấu 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

Design Pattern

- Design pattern cung cấp giải pháp ở dạng tổng quát, giúp tăng tốc độ phát triển phần mềm bằng cách đưa ra các mô hình test, mô hình phát triển đã qua kiểm nghiệm.
- Thiết kế phần mềm hiệu quả đòi hỏi phải cân nhắc các vấn đề sẽ nảy sinh trong quá trình hiện thực hóa (implementation).
- Dùng lại các design pattern giúp tránh được các vấn đề tiềm ẩn có thể gây ra những lỗi lớn, dễ dàng nâng cấp, bảo trì về sau.

Design Pattern

- Hệ thống các mẫu design pattern hiện có **23** mẫu được định nghĩa trong cuốn "*Design patterns Elements of Reusable Object Oriented Software*".
- Các tác giả của cuốn sách là Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides, hay còn được biết đến với các tên "**Gang of Four**" hay đơn giản là "**GoF**".
- Hệ thống các mẫu này có thể nói là đủ và tối ưu cho việc giải quyết hết các vấn đề của bài toán phân tích thiết kế và xây dựng phần mềm trong thời điểm hiện tại.
- Hệ thống các mẫu design pattern được chia thành 3 nhóm: nhóm Creational (5 mẫu), nhóm Structural (7 mẫu) và nhóm Behavioral (11 mẫu).

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 dấu cài đặt, che dấu thuật toán, cho phép thay đổi động cấu hình của đối tượng

Nhóm Creational (nhóm kiến tạo)

1	**Abstract Factory**	Cung cấp một interface cho việc tạo lập các đối tượng (có liên hệ với nhau) mà không cần qui định lớp khi hay xác định lớp cụ thể (concrete) tạo mỗi đối tượng **Tần suất sử dụng: cao**
2	**Builder**	Tách rời việc xây dựng (construction) một đối tượng phức tạp khỏi biểu diễn của nó sao cho cùng một tiến trình xây dựng có thể tạo được các biểu diễn khác nhau. Tần suất sử dụng: trung bình thấp
3	**Factory Method**	Định nghĩa Interface để sinh ra đối tượng nhưng để cho lớp con quyết định lớp nào được dùng để sinh ra đối tượng Factory method cho phép một lớp chuyển quá trình khởi tạo đối tượng cho lớp con. **Tần suất sử dụng: cao**
4	**Prototype**	Qui định loại của các đối tượng cần tạo bằng cách dùng một đối tượng mẫu, tạo mới nhờ vào sao chép đối tượng mẫu này. **Tần suất sử dụng: trung bình**
5	**Singleton**	Đảm bảo 1 class chỉ có 1 instance và cung cấp 1 điểm truy xuất toàn cục đến nó. **Tần suất sử dụng: cao / trung bình**

Nhóm Structural (nhóm cấu trúc)

6	**Adapter**	Do vấn đề tương thích, thay đổi interface của một lớp thành một interface khác phù hợp với yêu cầu người sử dụng lớp. **Tần suất sử dụng: cao trung bình**
7	**Bridge**	Tách rời ngữ nghĩa của một vấn đề khỏi việc cài đặt ; mục đích để cả hai bộ phận (ngữ nghĩa và cài đặt) có thể thay đổi độc lập nhau. Tần suất sử dụng: trung bình
8	**Composite**	Tổ chức các đối tượng theo cấu trúc phân cấp dạng cây; Tất cả các đối tượng trong cấu trúc được thao tác theo một cách thuần nhất như nhau. Tạo quan hệ thứ bậc bao gộp giữa các đối tượng. Client có thể xem đối tượng bao gộp và bị bao gộp như nhau -> khả năng tổng quát hoá trong code của client -> dễ phát triển, nâng cấp, bảo trì. **Tần suất sử dụng: cao trung bình**
9	**Decorator**	Gán thêm trách nhiệm cho đối tượng (mở rộng chức năng) vào lúc chạy (dynamically). Tần suất sử dụng: trung bình

Nhóm Structural (nhóm cấu trúc)

10	**Facade**	Cung cấp một interface thuần nhất cho một tập hợp các interface trong một "hệ thống con" (subsystem). Nó định nghĩa 1 interface cao hơn các interface có sẵn để làm cho hệ thống con dễ sử dụng hơn. **Tần suất sử dụng: cao**
11	**Flyweight**	Sử dụng việc chia sẻ để thao tác hiệu quả trên một số lượng lớn đối tượng "cở nhỏ" (chẳng hạn paragraph, dòng, cột, ký tự…). Tần suất sử dụng: thấp
12	**Proxy**	Cung cấp đối tượng đại diện cho một đối tượng khác để hỗ trợ hoặc kiểm soát quá trình truy xuất đối tượng đó. Đối tượng thay thế gọi là proxy. Tần suất sử dụng: cao trung bình

Nhóm Behavioral (nhóm tương tác)

13	**Chain of Responsibility**	Khắc phục việc ghép cặp giữa bộ gởi và bộ nhận thông điệp; Các đối tượng nhận thông điệp được kết nối thành một chuỗi và thông điệp được chuyển dọc theo chuỗi nầy đến khi gặp được đối tượng xử lý nó.Tránh việc gắn kết cứng giữa phần tử gởi request với phần tử nhận và xử lý request bằng cách cho phép hơn 1 đối tượng có có cơ hội xử lý request . Liên kết các đối tượng nhận request thành 1 dây chuyền rồi "pass" request xuyên qua từng đối tượng xử lý đến khi gặp đối tượng xử lý cụ thể. Tần suất sử dụng: trung bình thấp
14	**Command**	Mỗi yêu cầu (thực hiện một thao tác nào đó) được bao bọc thành một đối tượng. Các yêu cầu sẽ được lưu trữ và gởi đi như các đối tượng.Đóng gói request vào trong một Object , nhờ đó có thể nthông số hoá chương trình nhận request và thực hiện các thao tác trên request: sắp xếp, log, undo… Tần suất sử dụng: cao trung bình
15	**Interpreter**	Hỗ trợ việc định nghĩa biểu diễn văn phạm và bộ thông dịch cho một ngôn ngữ. Tần suất sử dụng: thấp
16	**Iterator**	Truy xuất các phần tử của đối tượng dạng tập hợp tuần tự (list, array, …) mà không phụ thuộc vào biểu diễn bên trong của các phần tử. **Tần suất sử dụng: cao**

Nhóm Behavioral (nhóm tương tác)

17	**Mediator**	Định nghĩa một đối tượng để bao bọc việc giao tiếp giữa một số đối tượng với nhau. Tần suất sử dụng: trung bình thấp
18	**Memento**	Hiệu chỉnh và trả lại như cũ trạng thái bên trong của đối tượng mà vẫn không vi phạm việc bao bọc dữ liệu. Tần suất sử dụng: thấp
19	**Observer**	Định nghĩa sự phụ thuộc một-nhiều giữa các đối tượng sao cho khi một đối tượng thay đổi trạng thái thì tất cả các đối tượng phụ thuộc nó cũng thay đổi theo. **Tần suất sử dụng: cao**
20	**State**	Cho phép một đối tượng thay đổi hành vi khi trạng thái bên trong của nó thay đổi , ta có cảm giác như class của đối tượng bị thay đổi. Tần suất sử dụng: trung bình

Nhóm Behavioral (nhóm tương tác)

21	**Strategy**	Bao bọc một họ các thuật toán bằng các lớp đối tượng để thuật toán có thể thay đổi độc lập đối với chương trình sử dụng thuật toán.Cung cấp một họ giải thuật cho phép client chọn lựa linh động một giải thuật cụ thể khi sử dụng. **Tần suất sử dụng: cao trung bình**
22	**Template method**	Định nghĩa phần khung của một thuật toán, tức là một thuật toán tổng quát gọi đến một số phương thức chưa được cài đặt trong lớp cơ sở; việc cài đặt các phương thức được ủy nhiệm cho các lớp kế thừa. Tần suất sử dụng: trung bình
23	**Visitor**	Cho phép định nghĩa thêm phép toán mới tác động lên các phần tử của một cấu trúc đối tượng mà không cần thay đổi các lớp định nghĩa cấu trúc đó. Tần suất sử dụng: thấp

Abstract Factory

- Một chương trình cần có khả năng chọn một trong một vài họ các lớp đối tượng
- Ví dụ, giao diện đồ họa nên chạy được trên một vài môi trường
- Mỗi môi trường (platform) cung cấp một tập các lớp đồ họa riêng:

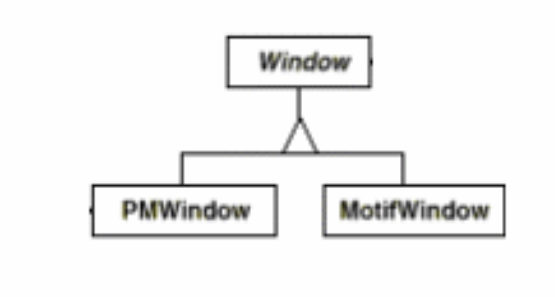
WinButton, WinScrollBar, WinWindow

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 mới

Giải pháp

- Định nghĩa Factory - lớp để tạo đối tượng:

```
class WidgetFactory {
   Button makeButton(args) = 0;
   Window makeWindow(args) = 0;
   // 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:

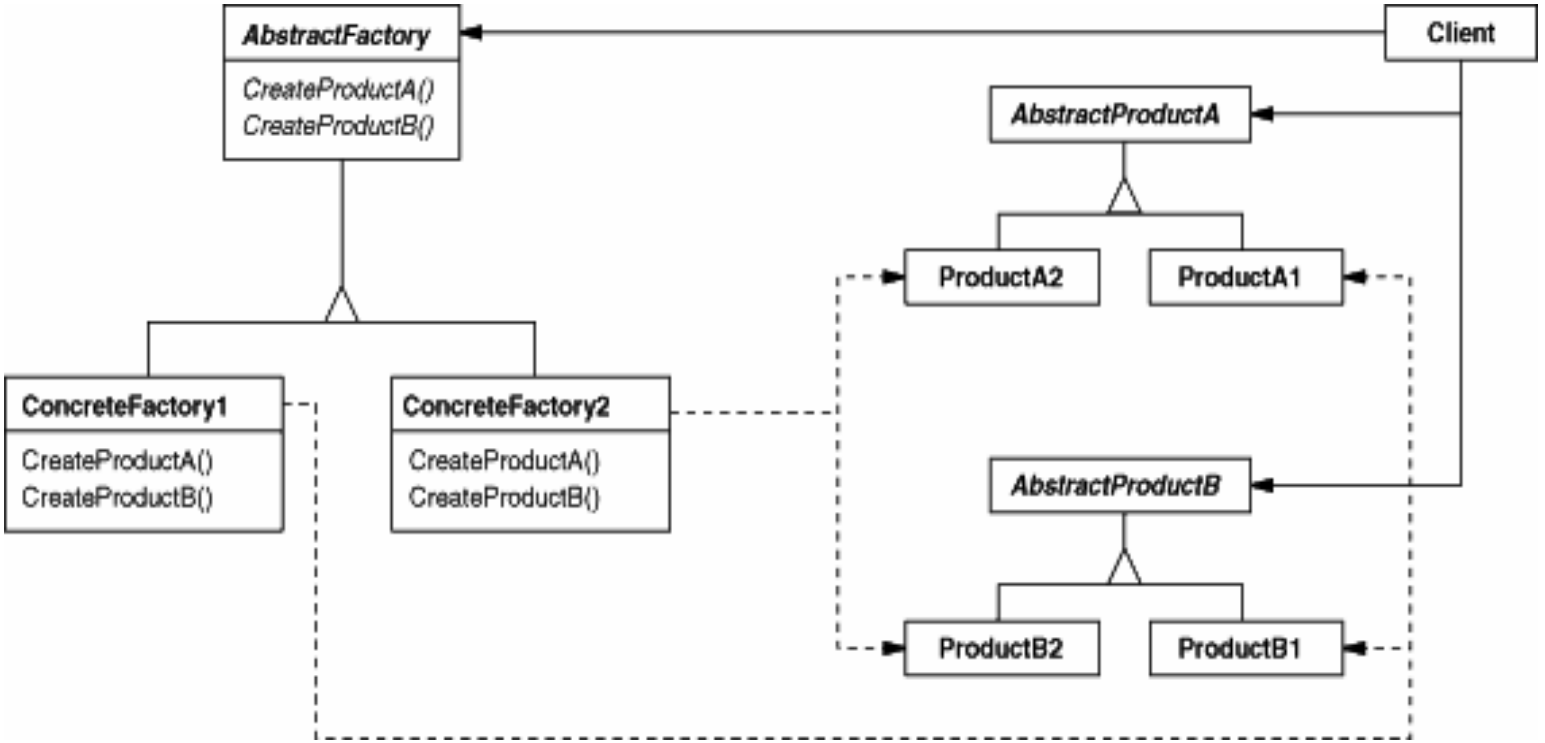
```
WidgetFactory wf =
new WinWidgetFactory();
```

- Khi khởi tạo đối tượng, không dùng "*new*" mà gọi:

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

- Thay đổi họ đối tượng - chỉ một lần trong mã cài đặt!
- Thêm họ - thêm một factory, không ảnh hưởng tới mã đang tồn tại!

Sơ đồ lớp


AbstractFactory
CreateProductA()
CreateProductB()
Client
AbstractProductA
ProductA2
ProductA1
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
AbstractProductB
ProductB2
ProductB1

Ứng dụng

- Các hệ điều hành khác nhau
- Các chuẩn look-and-feel khác nhau
- 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
- Chúng ta 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 CompositeGraphic extends Graphic
{
Graphics list[];
...
public void rotate(double angle) {  for
  (int i=0; i<list.length; i++)
list[i].rotate();
}
}
```

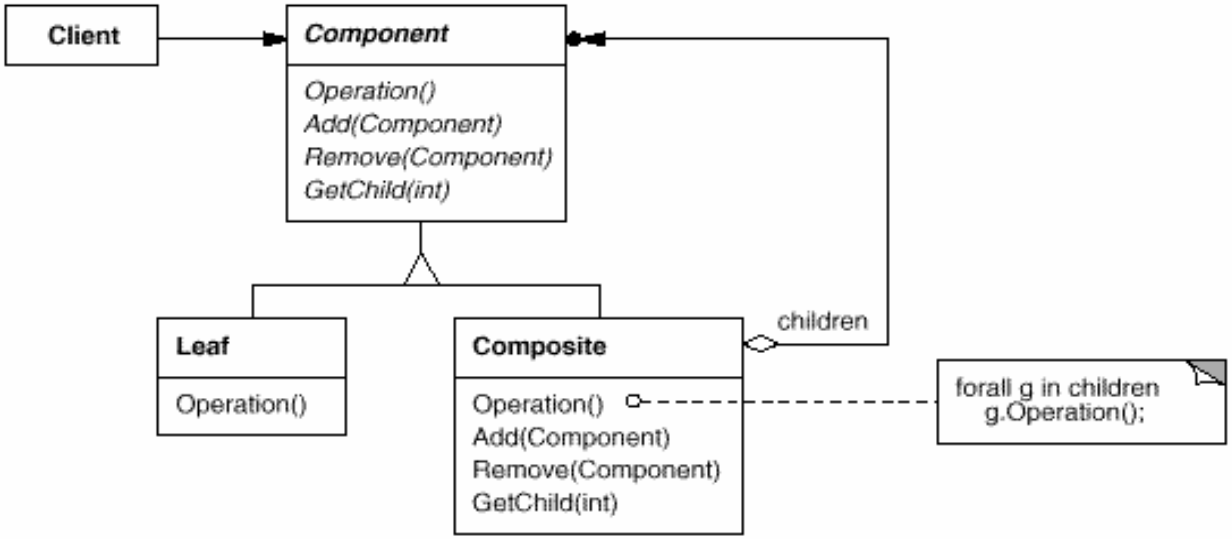
Giải pháp (tt)

- *CompositeGraphic* là
  - một danh sách nên nó có *add(), remove()* và *count()*
  - *Graphic* nên nó 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:
  - `CompositeGraphic cg;`
  - ```
    cg = new CompositeGraphic();
    cg.add(new Line(0,0,100,100));
    cg.add(new Circle(50,50,100));
    cg.rotate(90);
    ```
    ```

Sơ đồ lớp



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

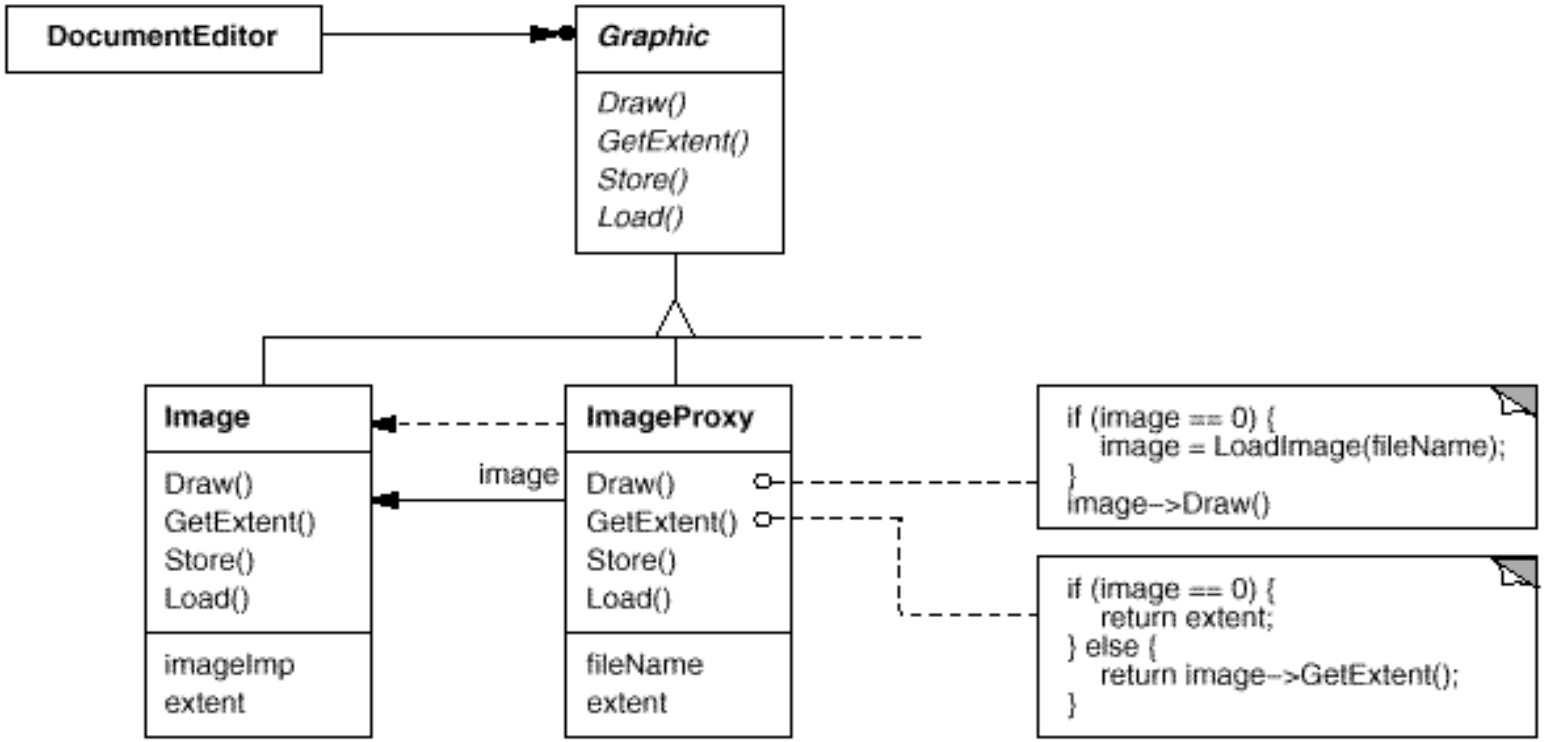
Ứng dụng

- 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


DocumentEditor
Graphic
Draw()
GetExtent()
Store()
Load()
Image
Draw()
GetExtent()
Store()
Load()
imageImp
extent
image
ImageProxy
Draw()
GetExtent()
Store()
Load()
fileName
extent
if (image == 0) {
image = LoadImage(fileName);
}
image->Draw()
if (image == 0) {
return extent;
} else {
return image->GetExtent();
}

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 D*ocument* 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, …

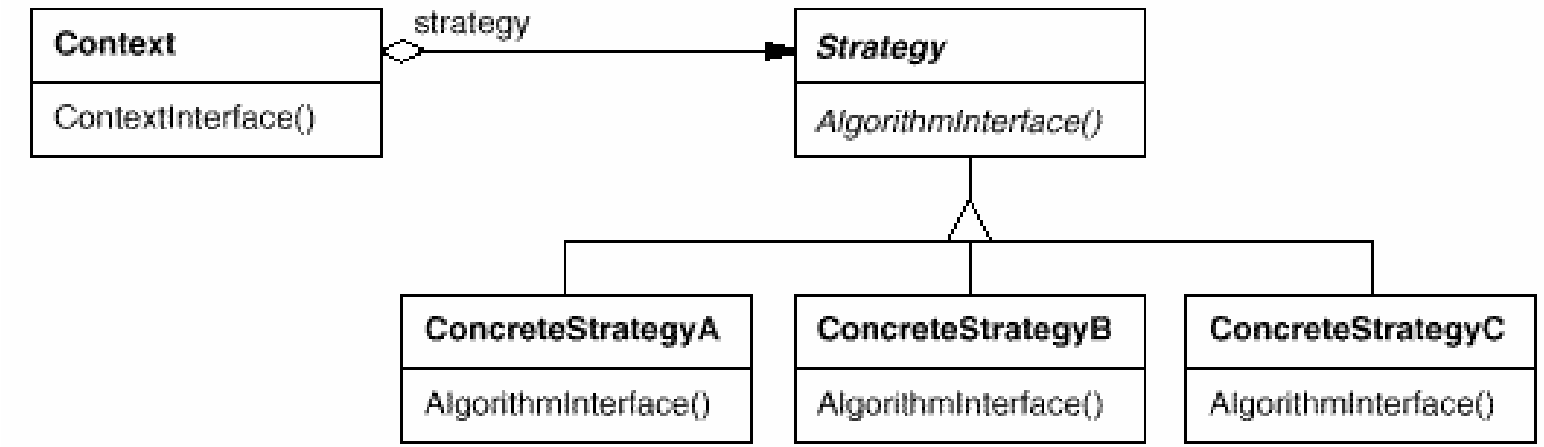
Giải pháp (tt)

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

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

    }
    private Renderer renderer;
  }
```

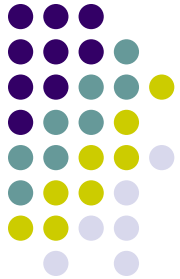
Sơ đồ lớp


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

Ứng dụng

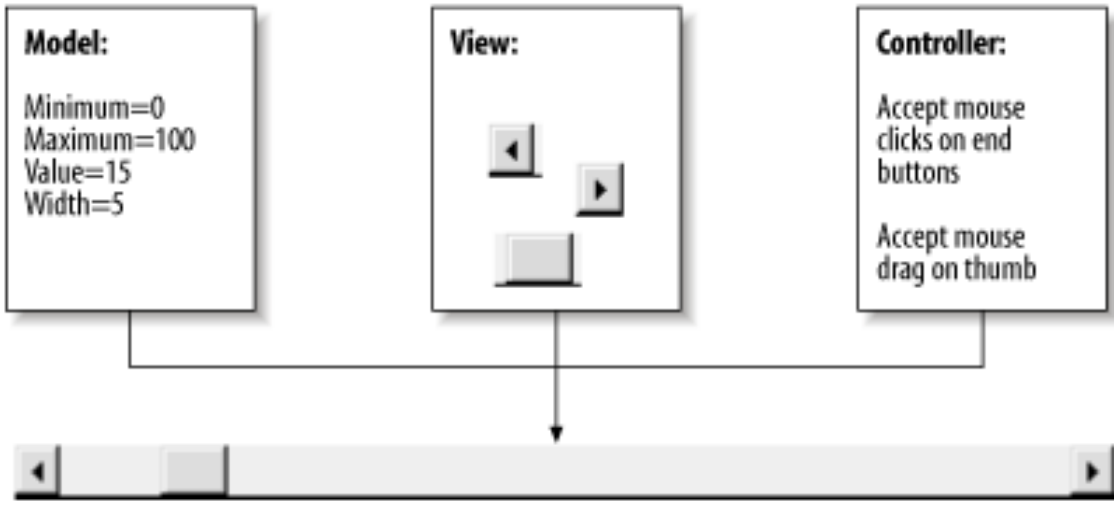
- Chương trình vẽ/soạn thảo
- Tối ưu biên dịch
- Chọn lựa các thuật toán heuristic khác nhau (trò chơi...)
- Lựa chọn các phương thức quản lý bộ nhớ khác nhau

Kiến trúc MVC của Swing

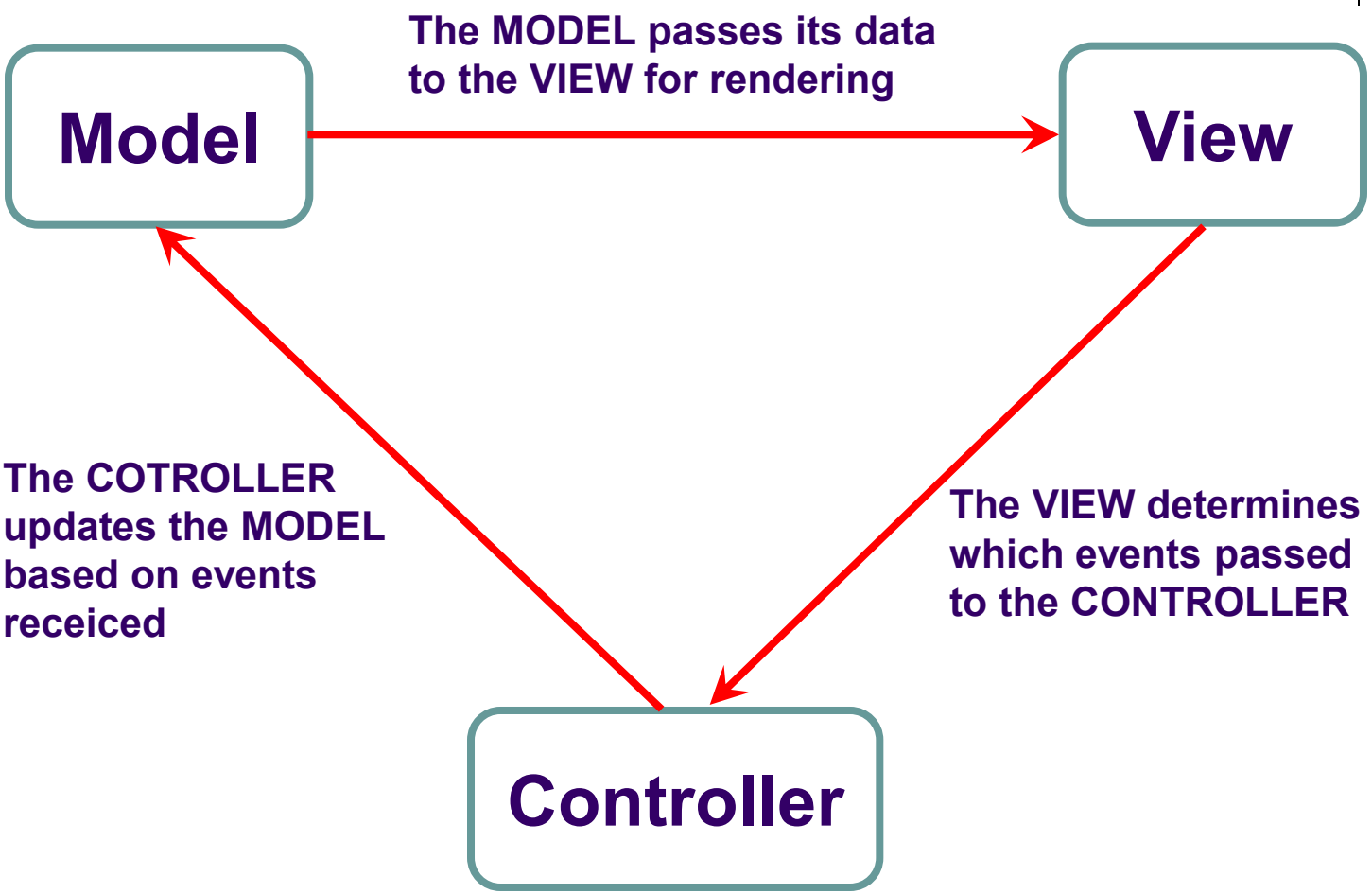


- Swing sử dụng kiến trúc MVC để xây dựng các thành phần, chương trình của mình, MVC chia mỗi thành phần giao diện thành 3 phần;
  - Model
    - Phần chứa nội dung trạng thái của các thành phần GUI
    - Mỗi kiểu thành phần GUI có 1 model khác nhau
  - View
    - Thể hiện trực quan thành phần GUI
  - Controller
    - Quản lý cách thức tương tác giữa các thành phần GUI với các sự kiện người dùng: click chuột, nhập phím…

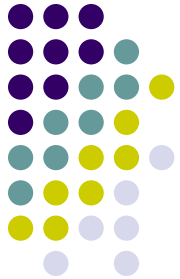
Kiến trúc MVC của Swing



Kiến trúc MVC thông thường



MVC trong Swing



- Swing hiện thực 1 mô hình `MVC` khá đơn giản còn được gọi là `model-delegate`
- Mô hình này nối `View` và `Controller` thành 1 đối tượng duy nhất gọi là `UI-delegate`
- UI-delegate làm 2 nhiệm vụ:
  - Hiển thị thành phần Swing lên màn hình
  - Quản lý các sự kiện

MVC trong Swing

- Mỗi thành phần GUI gồm 2 phần:
  - Model
  - UI-delegate