ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

****KHOA CÔNG NGHỆ PHẦN MỀM

**BÁO CÁO**

**MÔN MẪU THIẾT KẾ**

**Giảng viên hướng dẫn: Th.S Phạm Thi Vương**

**Sinh viên thực hiện: 12520234 – Nguyễn Hữu Lộc**

**12520037 – Lưu Công Chình**

**12520203 – Huỳnh Đăng Khoa**

**TP Hồ Chí Minh, Tháng 12 năm 2016**

**Nhận xét của Giảng viên**

LỜI CẢM ƠN

Trong quá trình thực hiện báo, nhóm xin được gửi lời cảm ơn đến thầy Phạm Thi Vương đã tạo điều kiện và tận tình hướng dẫn cho nhóm các bước tìm hiểu, nghiên cứu về mẫu thiết kế. Bên cạnh đó, nhóm còn được trau dồi thêm nhiều kiến thức, kinh nghiệm có thể áp dụng vào các dự án thực tiễn.

Ngoài những kết quả đạt được, nhóm vẫn không tránh khỏi những thiếu sót. Thông qua môn học này, nhóm mong rằng có thể học hỏi thêm nhiều kinh nghiệm hơn nữa và áp dụng những kinh nghiệm đó cho các dự án sắp tới.

Nhóm xin chân thành cảm ơn!

Sinh viên thực hiện

Nguyễn Hữu Lộc

Lưu Công Chình

Huỳnh Đăng Khoa

[LỜI CẢM ƠN 3](#_Toc472093729)

[Chương 1. TỔNG QUAN VỀ DESIGN PATTERN 8](#_Toc472093730)

[1.1. Tên và phân loại 8](#_Toc472093731)

[1.2. Tại sao phải sử dụng Design Pattern 8](#_Toc472093732)

[1.3. Khi nào nên sử dụng Design Pattern 9](#_Toc472093733)

[1.4. Phân loại Design Pattern 10](#_Toc472093734)

[1.5. Chú ý khi học Design Pattern 10](#_Toc472093735)

[Chương 2. MẪU BRIDGE 11](#_Toc472093736)

[2.1. Tên và phân loại 11](#_Toc472093737)

[2.2. Mục đích, ý nghĩa 11](#_Toc472093738)

[2.3. Bí danh 11](#_Toc472093739)

[2.4. Motivation 11](#_Toc472093740)

[2.5. Khả năng ứng dụng 14](#_Toc472093741)

[2.6. Cấu trúc 14](#_Toc472093742)

[2.7. Sự cộng tác 14](#_Toc472093743)

[2.8. Các hệ quả mang lại 14](#_Toc472093744)

[2.9. Chú ý khi cài đặt 15](#_Toc472093745)

[2.10. Hệ thống thực tế đang sử dụng 15](#_Toc472093746)

[2.11. Mẫu liên quan 15](#_Toc472093747)

[2.12. Mã nguồn minh họa 16](#_Toc472093748)

[2.12.1. Mô tả bài toán 16](#_Toc472093749)

[2.12.2. Tại sao sử dụng mẫu này? 16](#_Toc472093750)

[2.12.3. Sơ đồ UML 17](#_Toc472093751)

[2.12.4. Cấu trúc Package 18](#_Toc472093752)

[2.12.5. Vai trò các class tương ứng với cấu trúc của mẫu 18](#_Toc472093753)

[2.12.6. Source Code minh họa 19](#_Toc472093754)

[Chương 3. MẪU ABSTRACT FACTORY 21](#_Toc472093755)

[3.1. Tên và phân loại 21](#_Toc472093756)

[3.2. Mục đích, ý nghĩa 21](#_Toc472093757)

[3.3. Bí danh 21](#_Toc472093758)

[3.4. Motivation 22](#_Toc472093759)

[3.5. Khả năng ứng dụng 23](#_Toc472093760)

[3.6. Cấu trúc 23](#_Toc472093761)

[3.7. Sự cộng tác 24](#_Toc472093762)

[3.8. Các hệ quả mang lại 24](#_Toc472093763)

[3.9. Chú ý khi cài đặt 24](#_Toc472093764)

[3.10. Hệ thống thực tế đang sử dụng 24](#_Toc472093765)

[3.11. Mẫu liên quan 25](#_Toc472093766)

[3.12. Mã nguồn minh họa 25](#_Toc472093767)

[3.12.1. Mô tả bài toán 25](#_Toc472093768)

[3.12.2. Tại sao sử dụng mẫu này? 26](#_Toc472093769)

[3.12.3. Sơ đồ UML 26](#_Toc472093770)

[3.12.4. Cấu trúc Package 27](#_Toc472093771)

[3.12.5. Vai trò các class tương ứng với cấu trúc của mẫu 28](#_Toc472093772)

[3.12.6. Source Code minh họa 28](#_Toc472093773)

[Chương 4. MẪU OBSERVER 31](#_Toc472093774)

[4.1. Tên và phân loại 31](#_Toc472093775)

[4.2. Mục đích, ý nghĩa 32](#_Toc472093776)

[4.3. Bí danh 32](#_Toc472093777)

[4.4. Motivation 33](#_Toc472093778)

[4.5. Khả năng ứng dụng 34](#_Toc472093779)

[4.6. Cấu trúc 35](#_Toc472093780)

[4.7. Sự cộng tác 36](#_Toc472093781)

[4.8. Các hệ quả mang lại 37](#_Toc472093782)

[4.9. Chú ý khi cài đặt 37](#_Toc472093783)

[4.10. Hệ thống thực tế đang sử dụng 40](#_Toc472093784)

[4.11. Mẫu liên quan 40](#_Toc472093785)

[4.12. Mã nguồn minh họa 41](#_Toc472093786)

[4.12.1. Mô tả bài toán 41](#_Toc472093787)

[4.12.2. Tại sao sử dụng mẫu này? 42](#_Toc472093788)

[4.12.3. Sơ đồ UML 43](#_Toc472093789)

[4.12.4. Cấu trúc Package 44](#_Toc472093790)

[4.12.5. Vai trò các class tương ứng với cấu trúc của mẫu 44](#_Toc472093791)

[4.12.6. Source Code minh họa 44](#_Toc472093792)

[Chương 5. MẪU STRATEGY 47](#_Toc472093793)

[5.1. Tên và phân loại 47](#_Toc472093794)

[5.2. Mục đích, ý nghĩa 47](#_Toc472093795)

[5.3. Bí danh 48](#_Toc472093796)

[5.4. Motivation 48](#_Toc472093797)

[5.5. Khả năng ứng dụng 48](#_Toc472093798)

[5.6. Cấu trúc 49](#_Toc472093799)

[5.7. Sự cộng tác 49](#_Toc472093800)

[5.8. Các hệ quả mang lại 50](#_Toc472093801)

[5.9. Chú ý khi cài đặt 50](#_Toc472093802)

[5.10. Hệ thống thực tế đang sử dụng 50](#_Toc472093803)

[5.11. Mẫu liên quan 50](#_Toc472093804)

[5.12. Mã nguồn minh họa 50](#_Toc472093805)

[5.12.1. Mô tả bài toán 50](#_Toc472093806)

[5.12.2. Tại sao sử dụng mẫu này? 51](#_Toc472093807)

[5.12.3. Sơ đồ UML 51](#_Toc472093808)

[5.12.4. Cấu trúc Package 52](#_Toc472093809)

[5.12.5. Vai trò các class tương ứng với cấu trúc của mẫu 52](#_Toc472093810)

[Chương 6. MẪU FACTORY 53](#_Toc472093811)

[6.1. Tên và phân loại 53](#_Toc472093812)

[6.2. Mục đích, ý nghĩa 53](#_Toc472093813)

[6.3. Bí danh 54](#_Toc472093814)

[6.4. Motivation 54](#_Toc472093815)

[6.5. Khả năng ứng dụng 54](#_Toc472093816)

[6.6. Cấu trúc 55](#_Toc472093817)

[6.7. Sự cộng tác 55](#_Toc472093818)

[6.8. Các hệ quả mang lại 56](#_Toc472093819)

[6.9. Chú ý khi cài đặt 56](#_Toc472093820)

[6.10. Hệ thống thực tế đang sử dụng 56](#_Toc472093821)

[6.11. Mẫu liên quan 56](#_Toc472093822)

[6.12. Mã nguồn minh họa 57](#_Toc472093823)

[6.12.1. Mô tả bài toán 57](#_Toc472093824)

[6.12.2. Tại sao sử dụng mẫu này? 57](#_Toc472093825)

[6.12.3. Sơ đồ UML 58](#_Toc472093826)

[6.12.4. Cấu trúc Package 58](#_Toc472093827)

[6.12.5. Vai trò các class tương ứng với cấu trúc của mẫu 59](#_Toc472093828)

# TỔNG QUAN VỀ DESIGN PATTERN

## Tên và phân loại

Design patterns là các giải pháp đã được tối ưu hóa, được tái sử dụng cho các vấn đề lập trình mà chúng ta gặp phải hàng ngày. Nó là một khuôn mẫu đã được suy nghĩ, giải quyết trong tình huống cụ thể rồi.

Các vấn đề mà bạn gặp phải có thể bạn sẽ tự nghĩ ra cách giải quyết nhưng có thể nó chưa phải là tối ưu. Design Pattern giúp bạn giải quyết vấn đề một cách tối ưu nhất, cung cấp cho bạn các giải pháp trong lập trình OOP.  
Nó không phải là ngôn ngữ cụ thể nào cả. Design patterns có thể thực hiện được ở phần lớn các ngôn ngữ lập trình. Ta thường gặp nó nhất trong lập trình OOP.

## Tại sao phải sử dụng Design Pattern

Là một lập trình viên, thật khó khăn để nhớ chi tiết những việc mình đang thực hiện. Và khi bạn không nắm bắt được tổng quát công việc, bạn có thể dễ dàng bỏ lỡ những việc quan trọng. Khi đó, mã nguồn bạn đang viết có thể vẫn còn làm việc tốt đẹp, nhưng trừ khi bạn bao quát được bức tranh lớn hơn, lúc đó mã nguồn bạn viết mới thực sự hoàn hảo.

Những vấn đề nghiêm trọng thực sự thường xuất hiện sau khi bạn đã chỉnh sửa chúng ít nhất một lần. Những nhà phát triển thường tự mình xử lý bằng cách viết lại mã nguồn và sửa các lỗi. Tuy nhiên trong môi trường công việc, những nhà lập trình thường bỏ phần lớn thời gian để bảo trì, chỉnh sửa những công việc cũ hơn là tập trung vào những sản phẩm mới.

Bạn thấy rằng thật vô lý khi cứ phải làm, rồi sửa, làm lại, sửa tiếp… Giải pháp hợp lý nhất là bạn đưa ra được một quy trình tổng quan cho việc thiết kế và bảo trì, có như vậy, bạn mới tránh được các rắc rối phát sinh khi môi trường ứng dụng thay đổi, hoặc ít nhất bạn cũng giúp cho việc bảo trì, chỉnh sữa dễ dàng hơn khi có phát sinh.

Để giải quyết các vấn đề trên. Khái niệm về mẫu thiết kế (Design Pattern) ra đời  
làm đơn giản hóa quá trình trên. Kế hoạch này sẽ giúp bạn có một cái nhìn tổng quát. Một mẫu thiết kế Design Pattern là một giải pháp đã được kiểm nghiệm thành công khi đối diện một vấn đề lập trình phát sinh cụ thể. Khi bạn quen thuộc hết các mẫu thiết kế trong sách này, bạn nhìn vào một chương trình. Bam! Một giải pháp đúng đắn sẽ xuất hiện trong tâm trí bạn, thay vì bạn phải đập đầu vào tường trong vô vọng, giờ bạn có thể ung dung nói “Ở đây, tôi sẽ sử dụng mẫu Factory, mẫu Observer, hay mẫu Adapter…”

## Khi nào nên sử dụng Design Pattern

Điểm kỳ diệu của Design Patterns là nó giúp cho công việc của bạn dễ dàng tái sử dụng, dễ mở rộng và bảo trì. Khi bạn thiết kế không tốt, phần mềm của bạn không có khả năng tái sử dụng và bảo trì, khi gặp vấn đề phát sinh, bạn sẽ phải dành nhiều thời gian, có khi là nhiều hơn cả lúc bạn viết ban đầu, chỉ là để sửa chữa chúng.

Ví dụ: Bạn đang muốn tạo một đối tượng Java, nhiệm vụ là đọc và phân tích một tài liệu XML. Bạn cần phải tạo một lớp Parser (chuyên dùng để phân tích XML) sau đó bạn tạo một đối tượng của lớp này. Bạn thầm nghĩ “Tới giờ mọi việc vẫn ổn”. Nhưng thực tế thì đã có hàng tá lớp Parser do người khác viết, và họ luôn muốn sử dụng lại những tính năng đặc biệt trong lớp của họ. Nếu bạn có thể sử dụng mẫu thiết kế Nhà máy Factory, giờ đây bạn có thể sử dụng bất cứ lớp Parser nào, kể cả của những người khác, thay vì cứ khư khư xài lớp Parser do chính bạn viết ra. Và vì vậy, chương trình của bạn đã trở nên dễ mở rộng, tái sử dụng được và bảo trì dễ dàng.

Nói cách khác, Mẫu thiết kế Design Patterns là những giải pháp giúp cho ta có một thiết kế tốt khi đối diện những vấn đề phát sinh trong việc lập trình. Nhiều người đã gặp vấn đề này, và đã giải quyết tốt, việc của bạn là áp dụng chúng. Bạn không cần phải ghi nhớ mọi thứ, chỉ cần nhìn ra đâu là mẫu thiết kế phù hợp và đặt nó vào đúng chỗ.

## Phân loại Design Pattern

**Creational Pattern** (nhóm khởi tạo) gồm: Abstract Factory, Factory Method, Singleton, Builder, Prototype. Nó sẽ giúp bạn trong việc khởi tạo đối tượng, như bạn biết để khởi tạo bạn phải sử dụng từ khóa new, nhóm **Creational Pattern** sẽ sử dụng một số thủ thuật để khởi tạo đối tượng mà bạn sẽ không nhìn thấy từ khóa này.

**Structural Pattern** (nhóm cấu trúc) gồm: Adapter, Bridge, Composite, Decorator, Facade, Proxy và Flyweight. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.

**Behavioral Pattern** gồm: Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy và Visitor. Nhóm này dùng trong thực hiện các hành vi của đối tượng.

## Chú ý khi học Design Pattern

Bốn đặc tính của OOP: Thừa kế, Đa hình, Trừu tượng, Bao đóng.

Khái niệm interface và abstract. Cái này cực kỳ quan trọng, để hiểu và áp dụng 2 khái niệm này có thể sẽ mất một thời gian, nhưng khi bạn nắm chắc nó bạn sẽ thấy nó thực sự cần thiết.

Bỏ tư duy theo lối cấu trúc, nâng tư duy hoàn toàn OOP.

Bạn không cần phải nhớ mọi thứ, bạn chỉ cần biết là có những mẫu thiết kế đó. Và khi bạn đối diện với một vấn đề thực tế, sâu thẳm trong bạn tự nhiên thốt lên “À, có vẻ chỗ này có thể dùng mẫu Iterator…” Sau đó bạn chỉ cần tìm kiếm mẫu thiết kế đó trong cuốn sách này, duyệt qua các ví dụ để biết phải làm gì.

# MẪU BRIDGE

## Tên và phân loại

* Tên: Bridge.
* Phân loại: Structural Pattern.



## Mục đích, ý nghĩa

Là mẫu thiết kế giúp tách biệt thành phần trừu tượng với cài đặt của nó, từ đó cả hai có thể thay đổi một cách độc lập với nhau.

## Bí danh

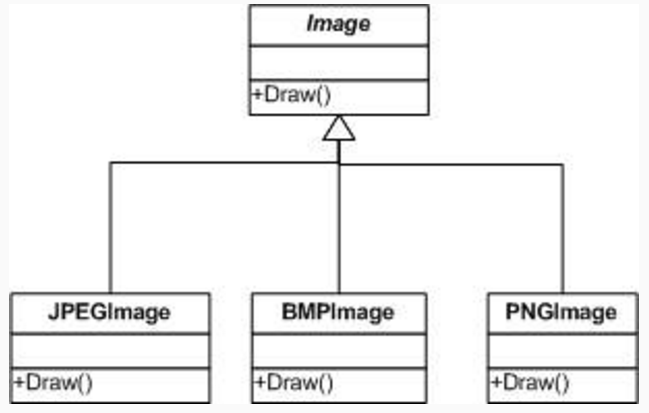
Handle, BodyPatterns.

## Motivation

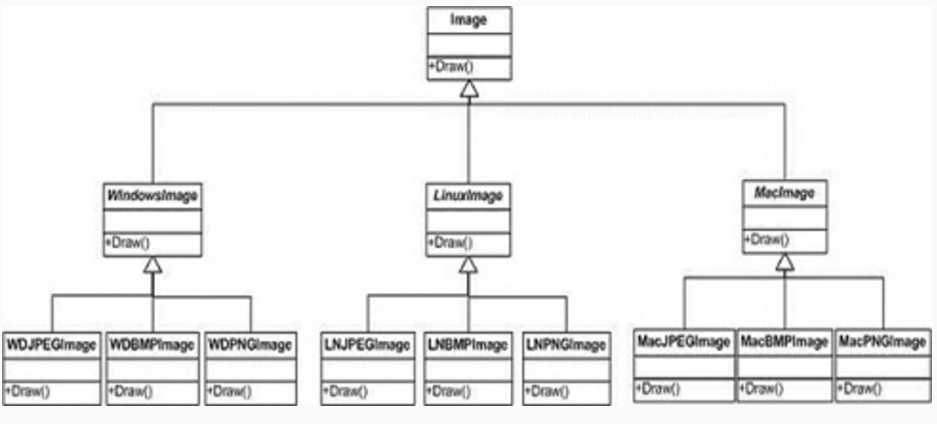
Như chúng ta biết, có rất nhiều chuẩn hình ảnh như JPEG, BMP, PNG… và các hệ điều hành khác nhau như Windows, Linux, MacOS đều có thể hiểu và hiển thị được các định dạng hình ảnh này. Mỗi định dạng ảnh có cấu trúc khác nhau và cách hiển thị cũng khác nhau tùy thuộc môi trường hệ điều hành.



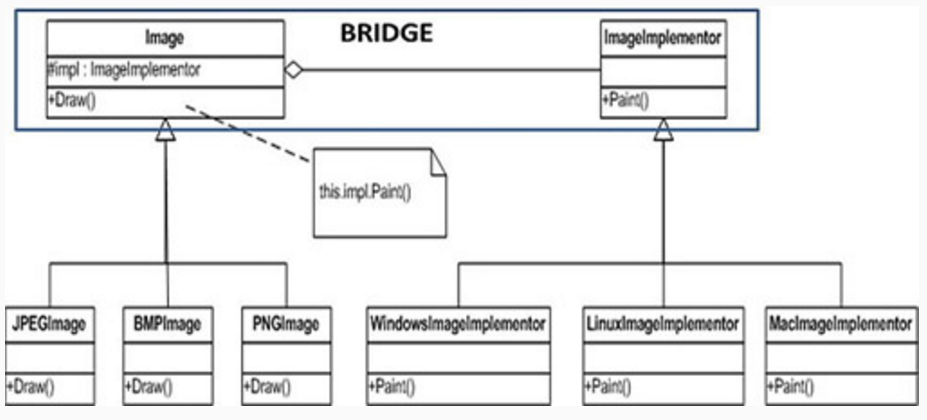
Giả sử chúng ta phải viết một phần mềm xem ảnh JPEG cho hệ điều hành Windows. Sau đó chúng ta phải mở rộng nó để cho phép chương trình này có thể xem được các định dạng ảnh khác như BMP, PNG… cũng trên Windows. Điều này có thể dễ dạng thực hiện khi chúng ta cho các class JPEGImage, BPMImage, PNGImage… kế thừa từ một abstract class (hoặc implement interface) là Image.



Tuy nhiên sau đó chúng ta cần mở rộng tiếp chương trình để nó có thể hiển thị được các định dạng ảnh này trên các hệ điều hành khác như Linux, MacOS… Chúng ta có thể cải tiến lại như sau:



Tuy nhiên, dễ dàng nhận thấy là cách làm này tạo ra sự lặp lại ý tưởng, lặp code, dài dòng và lại khó mở rộng về sau khi chúng ta có thêm các định dạng ảnh mới và các hệ điều hành mới.  
Để khắc phục điều này, ý tưởng của mẫu Bridge là tách phần xử lý sang một lớp khác, nghĩa là tách phần xử lý hiển thị hình ảnh sang một lớp mới.

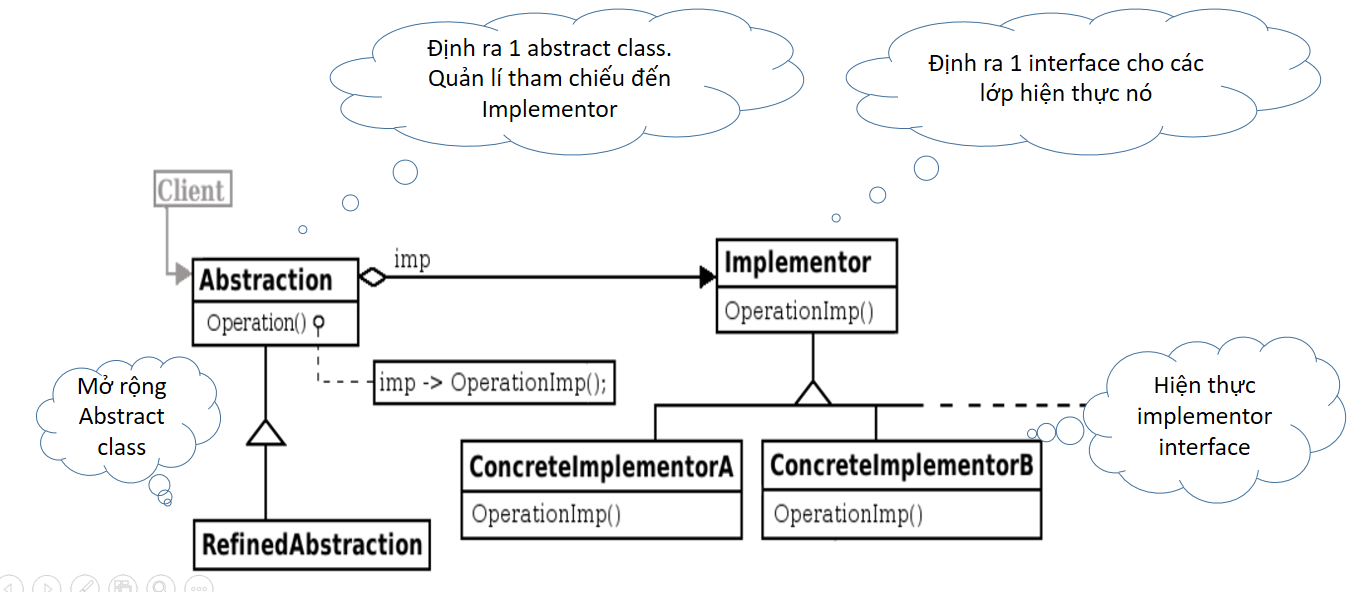


## Khả năng ứng dụng

Dùng khi ta muốn tách biệt phần trừu tượng với phần cài đặt.

Khi ta muốn ẩn phần cài đặt chi tiết đối với client. Các thay đổi ở phần cài đặt không ảnh hưởng đến client.

## Cấu trúc



## Sự cộng tác

Abstraction và Implementor khai báo các giao tiếp.

RefinedAbstraction và các ConcreteImplementor sẽ sử dụng các giao tiếp đó để mở rộng hoặc cài đặt cho cụ thể.

Trong Abstraction sẽ có biến kiểu là Implemetor và có constructor với tham số đầu vào là Implemetor. Do cùng kiểu nên có thể gán giá trị biến bằng tham số đó. Từ đó phần Abstraction sẽ có được Implementor tương ứng.

## Các hệ quả mang lại

Có thể thay đổi code ở phần cài đặt mà không ảnh hưởng đến code ở client.

Ẩn phần cài đặt đối với client.

## Chú ý khi cài đặt

Khi cài đặt, Imlementor và Abstraction là interface hoặc abstract class đều được.

## Hệ thống thực tế đang sử dụng

JDBC (Java Database Connectivity) là một API dùng để tương tác với các loại cơ sở dữ liệu quan hệ. Các class mà implement interface này là JDBC drivers. Các ứng dụng dựa trên các driver đó chính là phần abstraction.

Nó có thể làm việc với tất cả các database mà có JDBC driver hỗ trợ. Từ đó tách biệt được phần Abstraction và phần Imlementor.



## Mẫu liên quan

Thường nhầm lẫn với adapter pattern.

Khi phía client không tương thích với mã nguồn cũ. Ta sử dụng mẫu adapter để giải quyết vấn đề không tương thích đó.

Khi ta muốn tách biệt phần trừu tượng với phần cài đặt, ta sử dụng mẫu bridge.



## Mã nguồn minh họa

### Mô tả bài toán

Xét 2 loại định dạng ảnh là JPEG và PNG. Xét 2 hệ điều hành là Mac và Windows. Mỗi hệ điều hành sẽ có cách hiển thị ảnh tương ứng là MacAPI và WindowsAPI.

Áp dụng kiến thức về Design Pattern, hãy viết chương trình đơn giản mô phỏng việc vẽ các loại định dạng ảnh đó ở các hệ điều hành sao việc sửa chữa code ở phần định dạng ảnh và code các hệ điều hành API có thể thay đổi độc lập với nhau.

Chương trình xuất in ra kết quả ví dụ như :

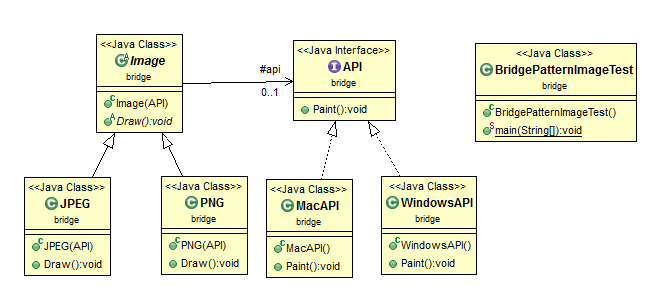
“PNG by MAC API”

“JPEG by Windows API”

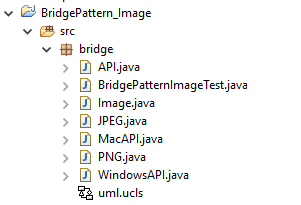
### Tại sao sử dụng mẫu này?

* Ta thấy có nhiều loại định dạng ảnh là JPEG, PNG.
* Có nhiều cách để hiện thực hóa việc vẽ các loại định dạng ảnh đó là WindowsAPI và MacAPI.
* Phần định dạng ảnh (Abstraction) thay đổi độc lập với phần API (Implementor)

### Sơ đồ UML



### Cấu trúc Package



### Vai trò các class tương ứng với cấu trúc của mẫu

Image: Abstraction.

JPEG, PNG: RefinedAbstraction.

API: Implementor.

MacAPI, WindowsAPI: ConcreteImplementor

### Source Code minh họa

API đóng vai trò là Implementor.

**public** **interface** API {

**public** **void** Paint();

}

MacAPI, WindowsAPI đóng vai trò là các ConcreteImplementor.

**public** **class** MacAPI **implements** API{

@Override

**public** **void** Paint() {

System.***out***.println("MAC API ");

}

}

**public** **class** WindowsAPI **implements** API {

@Override

**public** **void** Paint() {

System.***out***.println("Windows 10 API ");

}

}

Image đóng vai trò là Abstraction.

**public** **abstract** **class** Image {

**protected** API api;

**public** Image(API a)

{

**this**.api = a;

}

**abstract** **public** **void** Draw();

}

JPEG, PNG đóng vai trò là RefinedAbstraction.

**public** **class** JPEG **extends** Image{

**public** JPEG(API a) {

**super**(a);

}

@Override

**public** **void** Draw() {

System.***out***.print("JPEG by ");

api.Paint();

}

}

**public** **class** PNG **extends** Image {

**public** PNG(API a) {

**super**(a);

}

@Override

**public** **void** Draw() {

System.***out***.print("PNG by ");

api.Paint();

}

}

BridgePatternImageTest đóng vai trò là client.

**public** **class** BridgePatternImageTest {

**public** **static** **void** main(String[] args) {

PNG png = **new** PNG(**new** MacAPI());

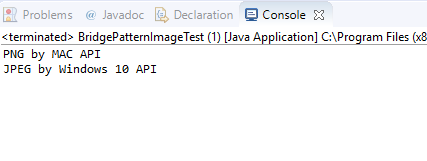
png.Draw();

JPEG jpeg = **new** JPEG(**new** WindowsAPI());

jpeg.Draw();

}

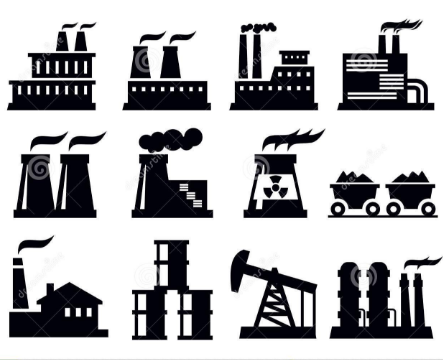
}



# MẪU ABSTRACT FACTORY

## Tên và phân loại

* Tên: Abstract Factory.
* Phân loại: Creational Pattern



## Mục đích, ý nghĩa

Cung cấp một interface để phục vụ cho việc tạo ra các dòng họ các đối tượng liên quan hoặc phụ thuộc lẫn nhau mà không chỉ ra lớp cụ thể của chúng.

## Bí danh

Factory of factories, Super factory.

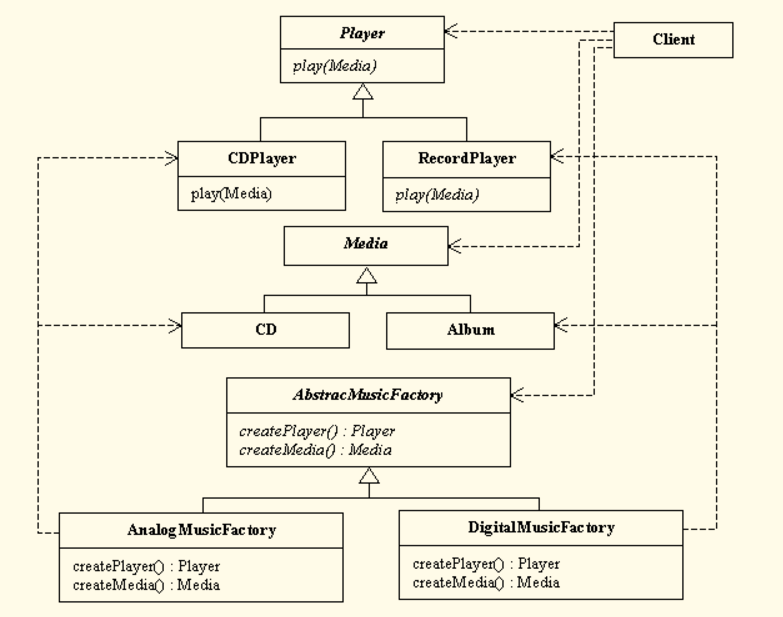
## Motivation

Giả sử ta cần 1 nhà máy để sản xuất ra cả phần cứng và phần mềm để phục vụ cho việc chơi nhạc.

Client không quan tâm loại phần cứng, hay loại phần mềm nào được dùng.

Cái client quan tâm đơn giản chỉ là chơi nhạc.

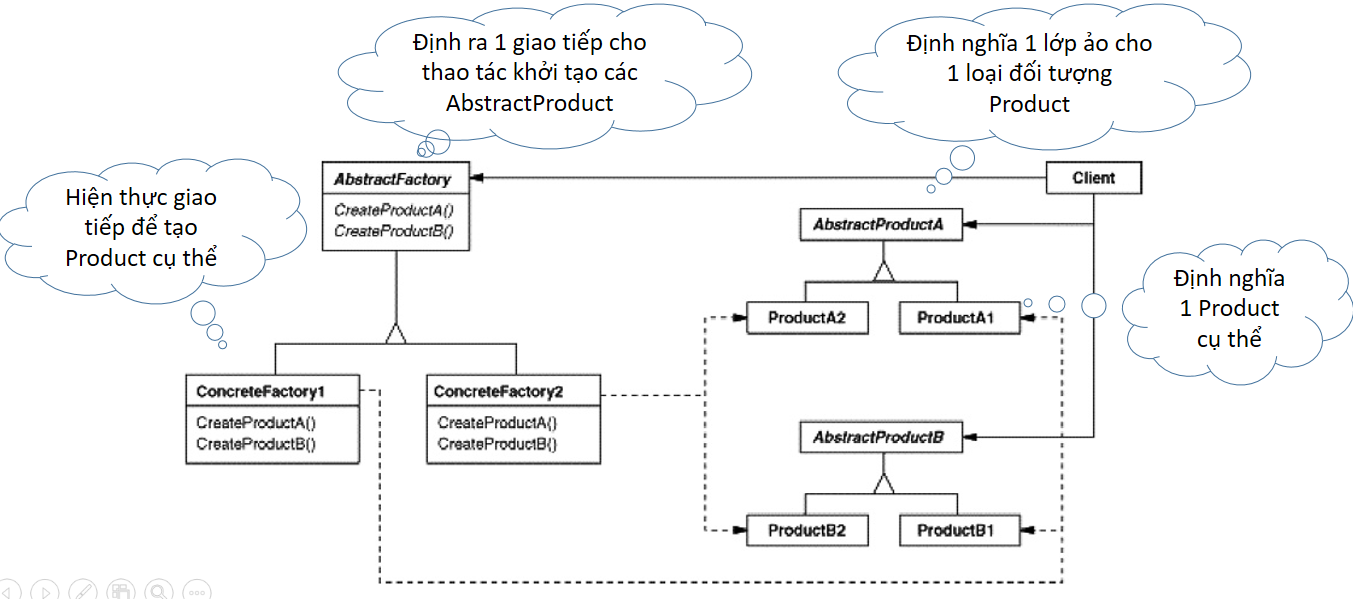
Các abstract classes “Player”, “Media” và “AbstractMusicFactory” tách biệt client với phần cứng và phần mềm chi tiết được dùng.



## Khả năng ứng dụng

* Mẫu này giải quyết được nhu cầu cần tạo ra các objects phù hợp với abstract interface.
* Dời trách nhiệm tạo đối tượng ra khỏi phía client.
* Client quyết định objects nào được tạo, nhưng 1 class khác (bên ngoài client) quyết định classes nào được tạo.
* Mẫu này được thiết kế để tạo ra các dòng các objects có liên quan với nhau.

## Cấu trúc



## Sự cộng tác

* AbstractFactory và AbstractProduct khai báo các giao tiếp.
* ConcreteFactory và ConcreteProduct cài đặt hoặc mở rộng các giao tiếp đó.
* Các ConcreteFactory đóng vai trò sản xuất ra các ConcreteProduct.

## Các hệ quả mang lại

* Giúp tách biệt các concrete classes với client và có thể thay đổi các concrete classes mà không cần phải chỉnh sửa client code.
* Tuy nhiên sẽ gặp khó khăn trong việc thêm sản phẩm mới.

## Chú ý khi cài đặt

AbstractFactory và AbstractProduct có thể là imterface hoặc abstract class đều được

## Hệ thống thực tế đang sử dụng

* XML API implements abstract factory. Có 1 class tên là [SchemaFactory](http://java.sun.com/javase/6/docs/api/javax/xml/validation/SchemaFactory.html) hoạt động như một factory và hỗ trợ hiện thực của nhiều schemas sử dụng abstract factory design pattern.
* <http://docs.oracle.com/javase/6/docs/api/javax/xml/validation/SchemaFactory.html>



## Mẫu liên quan

AbstractFactory thường sử dụng Factory Method Design Pattern để tạo ra các đối tượng sản phẩm.

## Mã nguồn minh họa

### Mô tả bài toán

Để phục vụ cho việc chơi nhạc. Ta sử dụng các phần cứng gồm album và CD, đồng thời ta sử dụng phần mềm chơi nhạc gồm CDPlayer và RecordPlayer.

Nhạc Digital được chơi bằng cách sử dụng CD và CDPlayer. Nhạc Analog được chơi bằng cách sử dụng Album và RecordPlayer

Áp dụng kiến thức về Design Pattnern, hãy viết chương trình đơn giản mô phỏng việc chơi nhạc cho client. Trong đó, client sẽ ra yêu cầu là chơi nhạc Digital hoặc Analog, chương trình phải biết kết hợp đúng phần cứng và phần mềm để chơi nhạc cho client. Còn cụ thể là phần mềm nào kết hợp với phần mềm nào, phía client không quan tậm.

Chương trình xuất ra kết quả đơn giản ví dụ như :

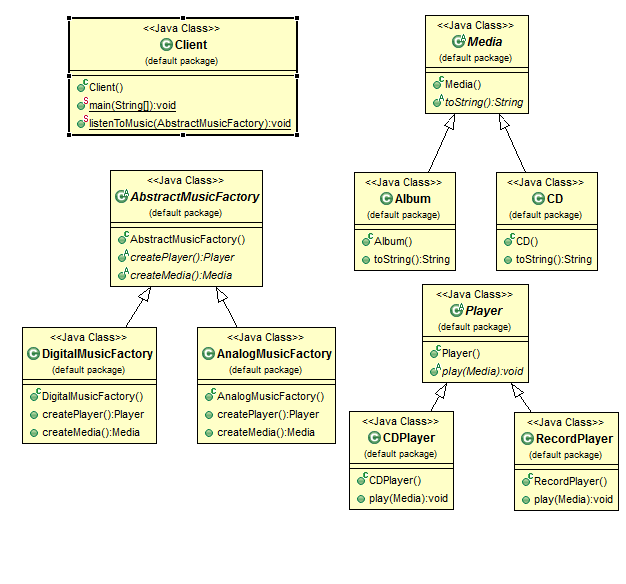
* “Playing a CD on a CD player”
* “Playing an album on a record player”



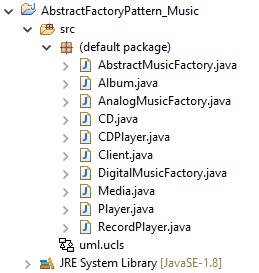
### Tại sao sử dụng mẫu này?

* Ta thấy có nhiều dòng các sản phẩm. Dòng thứ nhất là các Media (bao gồm Album và CD), dòng thứ hai là các Player (bao gồm CDPlayer và RecordPlayer).
* Ta cần khởi tạo ra các đối tượng đó và kết hợp với nhau sao cho đúng cách để có thể chơi được nhạc
* Việc kết hợp như thế nào, phía client không quan tâm.

### Sơ đồ UML



### Cấu trúc Package



### Vai trò các class tương ứng với cấu trúc của mẫu

* Media: AbstractProductA.
* Player: AbstractProductB.
* Album, CD: ProductA1, ProductA2.
* CDPlayer, RecordPlayer: ProductB1, ProductB2.
* AbstractMusicFatory: AbstractFactory.
* DigitalMusicFactory, AnalogMusicFactory: ConcreteFactory1, ConcreteFactory2.

### Source Code minh họa

AbstractMusicFatory đóng vai trò là AbstractFactory.

**public** **abstract** **class** AbstractMusicFactory {

**public** **abstract** Player createPlayer();

**public** **abstract** Media createMedia();

}

DigitalMusicFactory đóng vai trò là ConcreteFactory1, ConcreteFactory2.

**public** **class** DigitalMusicFactory **extends** AbstractMusicFactory {

@Override

**public** Player createPlayer() {

// **TODO** Auto-generated method stub

**return** **new** CDPlayer();

}

@Override

**public** Media createMedia() {

// **TODO** Auto-generated method stub

**return** **new** CD();

}

}

**public** **class** AnalogMusicFactory **extends** AbstractMusicFactory {

@Override

**public** Player createPlayer() {

// **TODO** Auto-generated method stub

**return** **new** RecordPlayer();

}

@Override

**public** Media createMedia() {

// **TODO** Auto-generated method stub

**return** **new** Album();

}

}

Media đóng vai trò là AbstractProductA.

**public** **abstract** **class** Media {

**public** **abstract** String toString();

}

Album, CD đóng vai trò là ProductA1, ProductA2.

**public** **class** Album **extends** Media {

@Override

**public** String toString() {

// **TODO** Auto-generated method stub

**return** "an album";

}

}

**public** **class** CD **extends** Media {

@Override

**public** String toString() {

// **TODO** Auto-generated method stub

**return** "a CD";

}

}

Player đóng vai trò là AbstractProductB.

**public** **abstract** **class** Player {

**public** **abstract** **void** play(Media m);

}

**public** **class** CDPlayer **extends** Player {

@Override

**public** **void** play(Media m) {

// **TODO** Auto-generated method stub

System.***out***.println("Playing " + m + " on a CD player");

}

}

CDPlayer, RecordPlayer: đóng vai trò là ProductB1, ProductB2.

**public** **class** RecordPlayer **extends** Player {

@Override

**public** **void** play(Media m) {

// **TODO** Auto-generated method stub

System.***out***.println("Playing " + m + " on a record player");

}

}

Client đóng vai trò là Client.

**public** **class** Client {

**public** **static** **void** main(String[] args) {

*listenToMusic*(**new** DigitalMusicFactory());

*listenToMusic*(**new** AnalogMusicFactory());

}

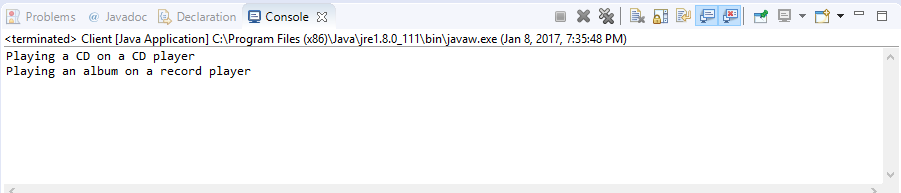
**public** **static** **void** listenToMusic(AbstractMusicFactory amf) {

Player p = amf.createPlayer();

Media m = amf.createMedia();

p.play(m);

}

} 

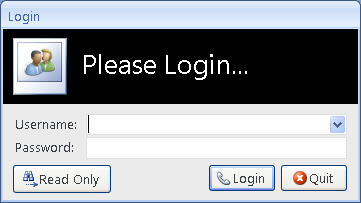
# MẪU OBSERVER

## Tên và phân loại

* Tên: Observer.
* Phân loại: Behavioral Pattern.

Là một trong những mẫu thiết dành cho việc một đối tượng khi *thay đổi trạng thái* của bản thân nó thì các đối tượng đính kèm theo cũng sẽ được thông báo.   
Ví dụ minh họa trong đời sống:   
“Fans hâm mộ và người nghệ sĩ nổi tiếng được hâm mộ”.  
Tất cả các thành viên của nhóm hâm mộ nghệ sĩ đó sẽ “theo dõi” và cập nhật những thông tin mới nhất về người nghệ sĩ đó đến khi nào họ không còn thích nữa. Nếu không thích nữa đơn giản họ ngừng mọi hành động “theo dõi” và cập nhật thông tin về người nghệ sĩ mà họ hâm mộ.

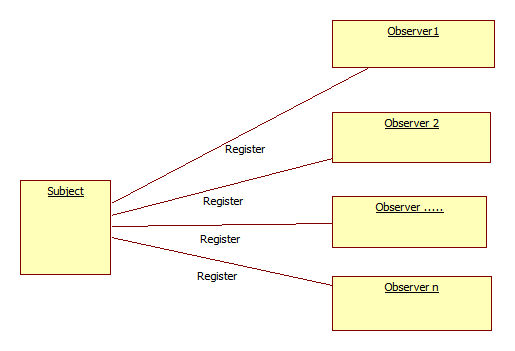
Ví dụ trong lĩnh vực công nghệ thông tin:  
Trong thế giới khoa học máy tính, Giao diện người dùng( UI) là một ví dụ điển hình. Khi UI đã được kết nối với database hoặc Business logic. Người dùng có thể truy vấn dữ liệu thông qua UI, sau khi tìm kiếm trong database hoàn thành, kết quả trả về UI. Trong hều hết các trường hợp, chúng ta thường tách riêng UI và Database. Nếu như có một sự thay đổi xảy ra ở database, UI nên được thông báo để mà có thể cập nhật và hiển thị cho người dùng.



## Mục đích, ý nghĩa

Xác định sự phụ thuộc một-nhiều giữa các đối tượng để khi một đối tượng chủ thể(Subject) thay đổi trạng thái, tất cả những đối tượng phụ thuộc(Observer) của nó sẽ được được thông báo và cập nhật tự động.

Phía “một” thường là dữ liệu, phía “nhiều” thường là giao diện người dùng.



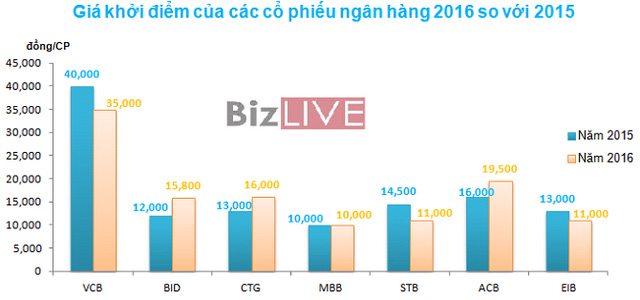
## Bí danh

* Dependents-Sự phụ thuộc.
* Publish-Subscribe.
* Source/Listener-Lắng nghe.

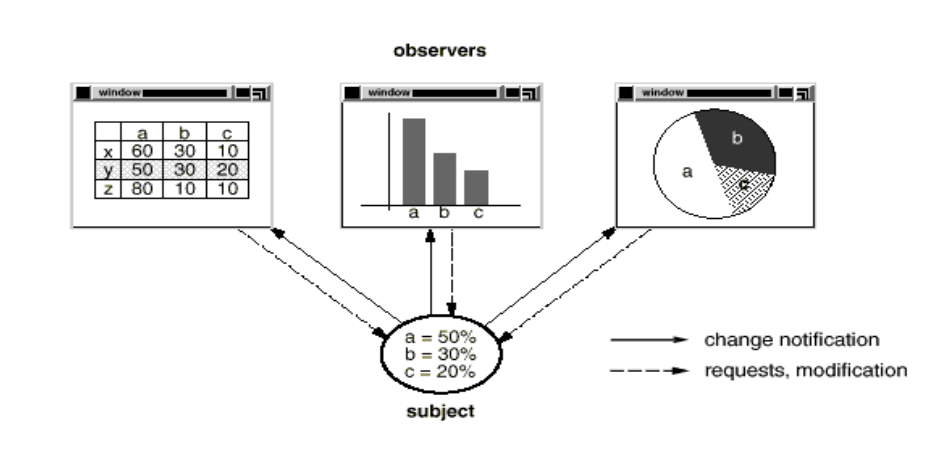
## Motivation

Trong lĩnh vực công nghệ thông tin, chúng ta cũng có thể đã từng thấy hoặc đã làm những phần mềm dựa trên ý nghĩa của mấu observer này.

Ta viết phần mềm hiển thị dữ liệu (giá vàng, cổ phiếu, biểu đồ…) trên các thiết bị, môi trường (window app, web, mobile,tablet…).



Khi có sự thay đổi số liệu ở cơ sở dữ liệu (CSDL), các thiết bị sẽ lần lượt cập nhật và hiển thị lại dữ liệu ngay lập tức.

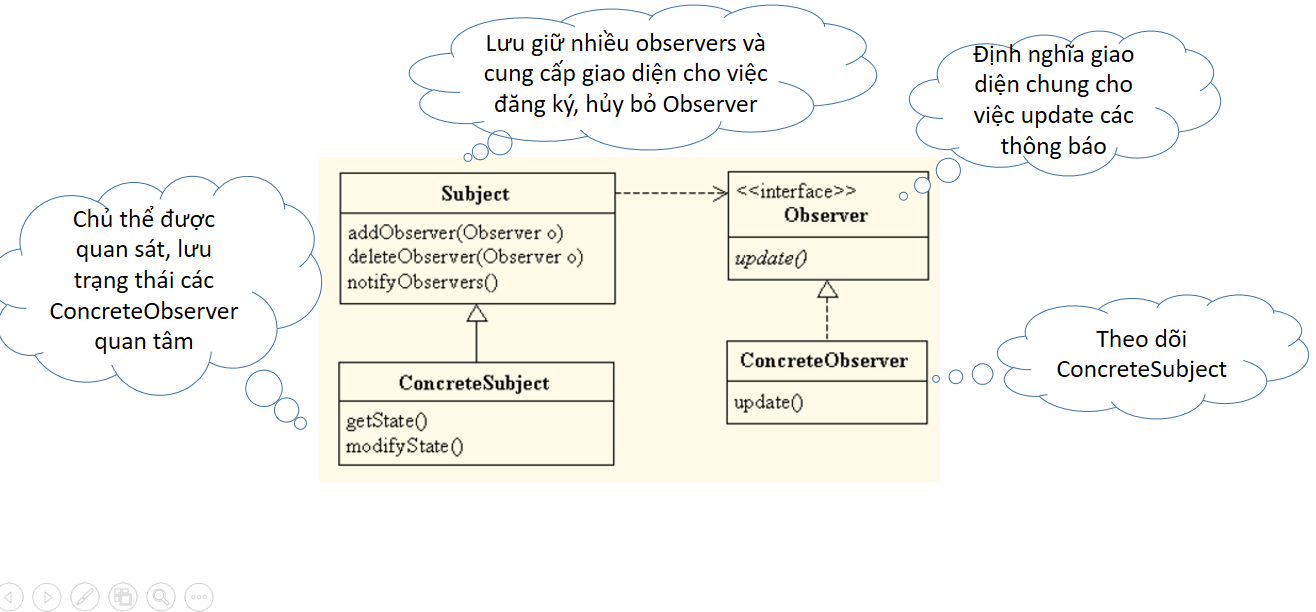
Nếu chúng ta xây dựng phần mềm có chức năng hiển thị dữ liệu giống như biểu đồ phía trên thì ý tưởng đầu tiên phải nghĩ đến có lẽ phải là mẫu design pattern này.   
Nó hiển thị dữ liệu động, và các thông số cột cũng như giá trị hiển thị trên mỗi cột đều dựa trên dữ liệu có được ở database chứ hoàn toàn không thiết lập cứng.Tùy theo giá trị có được của các biến a, b, c thì sẽ cho ra giao diện hiển thị của biều đồ khác nhau tương ứng với các giá trị đó.

## Khả năng ứng dụng

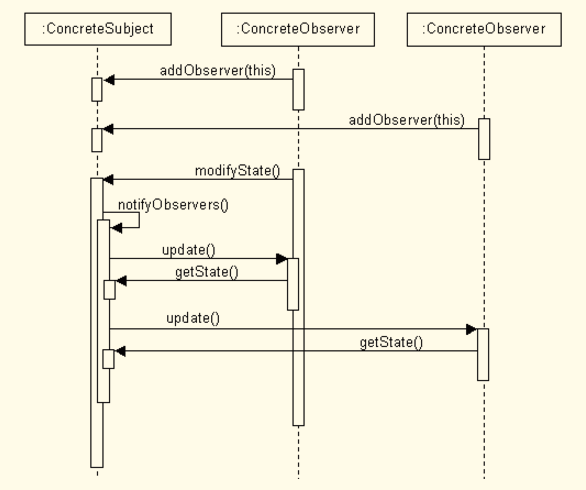
Chúng ta sẽ sử dụng mẫu này trong các tình huống như sau:

* Khi có một sự phụ thuộc của một hoặc nhiều đối tượng vào 1 đối tượng khác.
* Khi sự thay đổi của một đối tượng này có thể gây ra sự thay đổi của các đối tượng khác.
* Khi một đối tượng có khả năng thông báo cho các đối tượng khác mà không làm thay đổi cấu trúc bên trong của các đối tượng đó.

## Cấu trúc



## Sự cộng tác



## Các hệ quả mang lại

Subject không cần biết các Observer được tạo ra như thế nào chỉ cần các Observer hiện thực được interface hoặc class có phương thức cập nhật(Update) là được.

Subject và Observer có thể thuộc về các layer khác nhau( Model-View). Cụ thể Subject là Model còn Observer là View. Phân ra 2 khái niệm tương đối rõ ràng.

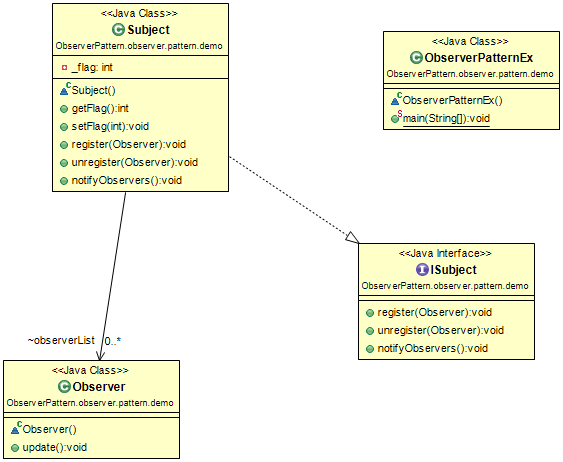
Ví dụ: Một website với giao diện trực quan, người dùng có thể tương tác với cơ sở dữ liệu( Subject) thông qua giao diện đó và những thay đổi ở cơ sở dữ liệu đều được update lên giao diện mỗi khi người dùng thực hiện sự kiện liên quan.

Subject gửi thông báo đến tất cả các đối tượng observer đã được đăng kí. Các observer có thể được thêm và xóa bất cứ lúc nào thông qua phương thức register(Observer o) hoặc unregister(Observer o) do lập trình viên tự định nghĩa.

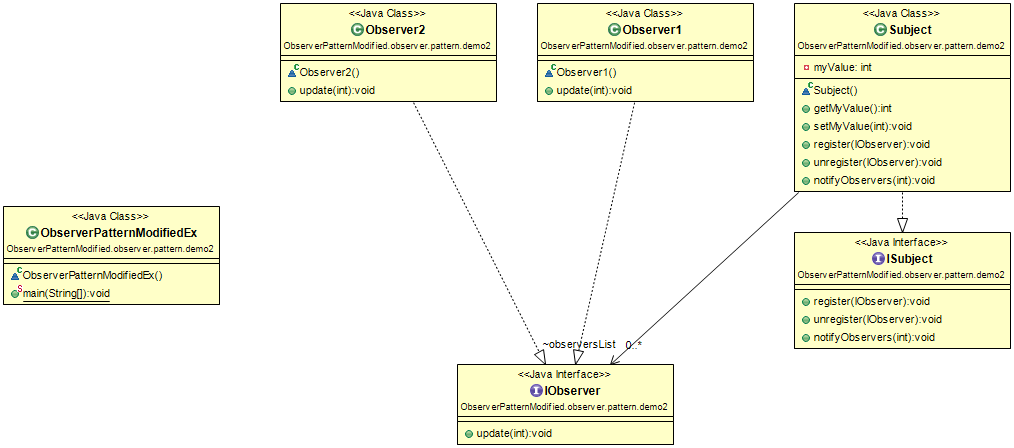
## Chú ý khi cài đặt

Tùy theo mục đích sử dụng mà có thể phân ra 3 trường hợp sử dụng mẫu Observer này:

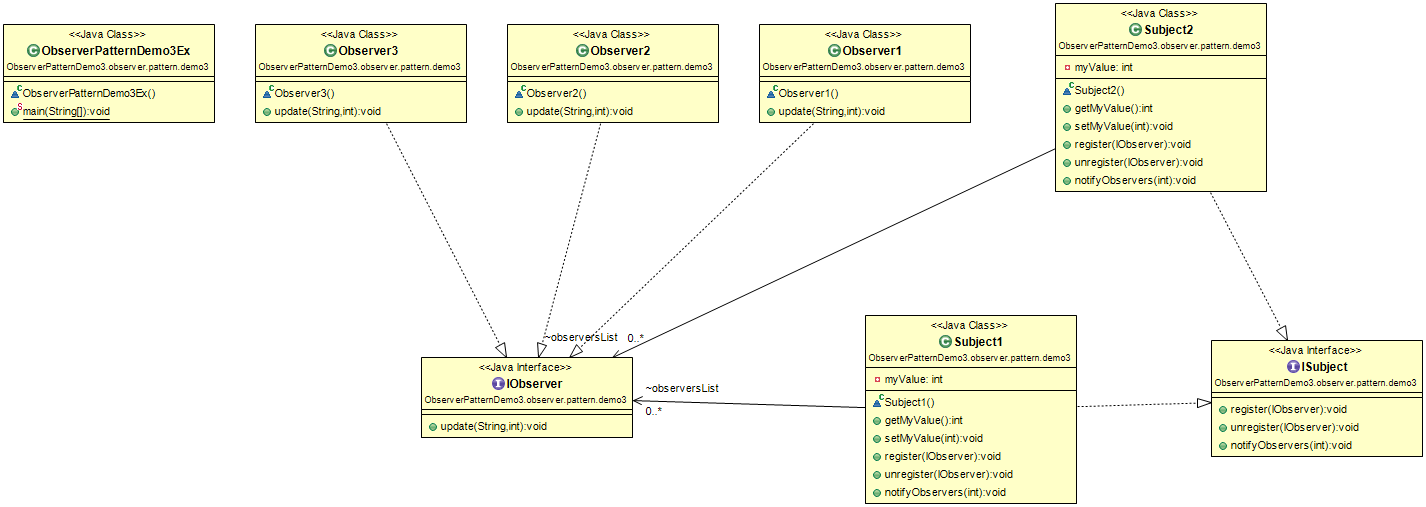
* Một chủ thể và một observer hoặc danh sách observer giống nhau. Có nghĩa là trong trường hợp này bất kể khi nào subject có thay đổi và thực hiện notify đến các observer thì các observer này đều thực hiện cùng 1 phương thức update().Ta sẽ không sử dụng interface cho các observer trong trường hợp này. Mà thay vào đó là 1 class Observer như hình minh họa phía dưới.



* Một chủ thể và nhiều observer có phương thức update khác nhau( tương tự như tính đa hình). Trong trường hợp này chúng ta sử dụng interface và có phương thức trừu tượng update(tham số). Khi có nhiều class observer khác nhau sẽ implement interface đó. Và hiện thực phương thức update(tham số) theo cách của riêng mình. Min họa trong diagram sau:



* Trường hợp thứ 3 là có nhiều subject và có nhiều observer. Các observer khác nhau có thể lắng nghe từ các subject khác nhau và hiện thực phương thức update khác nhau tùy theo mục đích sử dụng. Các subject khác nhau cùng implement một interface có các phương thức trừu tượng như register(Observer o), unregister(Observer o) và notifyObservers(tham số). Minh họa như hình sau:



Chắn chắn các Subject update trạng thái trước khi gửi thông báo. Đây là một chú ý khi làm với mẫu này. Phương thức notifyObserver phải gọi sau khi đã có sự thay đổi dữ liệu ở subject. Nếu không thì khi tiến hành cập nhật. Observer vẫn lấy dữ liệu cũ ban đầu.

Các observer cũng có thể trở thành một Subject. Ngoài chức năng là lắng nghe những thay đổi từ subject thì các observer hoàn toàn có thể trở thành các subject nếu nó có các phương thức tương tự như một subject.

## Hệ thống thực tế đang sử dụng

Java API cung cấp 1 class và 1 interface để implement observer pattern đó là : java.util.Observable (class) và java.util.Observer (interface).

Java cung cấp class Observable. Bất cứ class nào muốn được theo dõi hay lắng nghe đều phải extend class này.

Class này có gì? Class này cung cấp các phương thức để add/delete observers.

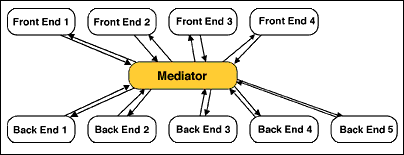
Cung cấp các phương thức để notify đến tất các các observers.

Ngoài class trên. Java còn cung cấp 1 interface Observer. Nó phải được implement bởi bất kì class observer nào. Trong interface này có một phương thức trừu tượng là update(Observable o, Object arg).

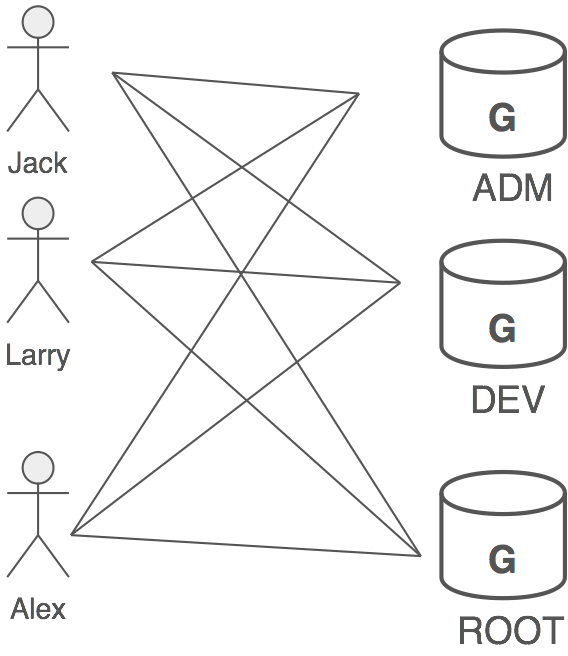
## Mẫu liên quan

Cả Observer Pattern và Mediator Pattern đều phục vụ cho việc giao tiếp.   
Điểm khác biệt là Observer Pattern cung cấp đối tượng “Observer” và “Subject” để phục vụ cho việc giao tiếp. Còn Mediator Pattern dùng đối tượng “Mediator” để đóng gói giao tiếp giữa các đối tượng.

Mediator: Đơn giản hơn mẫu observer bởi vì các sự kiện thực hiện được gửi đến một trạm chung được gọi là mediator. Và từ đó sẽ điều phối thực hiện đến observer hay subject tương ứng.



Observer : Với mẫu observer thì chúng ta phải tạo liên kết từng đối tượng subject với từng observer . Tạo nên một sự rối rắm nếu như số lượng subject và observer nhiều.



## Mã nguồn minh họa

### Mô tả bài toán

Giả sử, ta có một chủ thể(Subject) ví dụ như ca sĩ và một danh sách người theo dõi(observer) như là fan ham mộ của ca sĩ đó. Họ có nhiệm vụ lắng nghe, theo dõi những thay đổi từ chủ thể. Khi có bất cứ thông tin nào thay đổi từ phía ca sĩ, thì đều phải được thông báo đến các fan ham mộ. Chẳng hạn như ca sĩ đó mới có bài hit và muốn cho fan ham mộ của mình biết thông tin về nó. Dựa trên ý nghĩa của mẫu observer như sau:

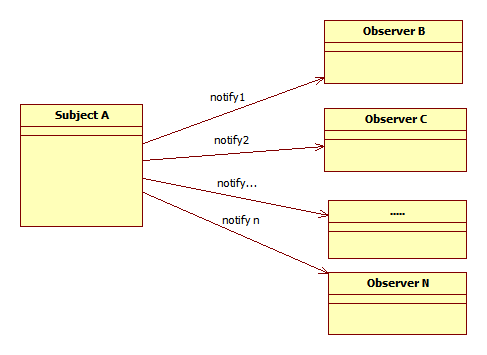
Một chủ thể(Subject) cần thông báo cho các đối tượng khác (Observer) về sự thay đổi trạng thái hay dữ liệu bên trong. Các đối tượng nhận thông báo từ chủ thể và tiếp hành update lại dữ liệu hoặc trạng thái cho phù hợp.

Hãy viết một chương trình đơn giản, hỗ trợ các chức năng cơ bản như thông báo(notify) từ Subject đến Observer và chức năng cập nhật(update) từ Observer khi nhận được thông báo có sự thay đổi từ Subject.

### Tại sao sử dụng mẫu này?

Observer là một mẫu hớp lý trong tình huống này. Giả sử ta không biết sự tồn tại của mẫu này. Ta sẽ tiến hành làm theo cách thông thường. Subject A sau khi thay đổi dữ liệu sẽ tiếng hành thông báo cho đối tượng B. Mọi chuyện vẫn diễn ra bình thường. Tiếp tục them một đối tượng C cần biết sự thay đổi của đối tượng A.

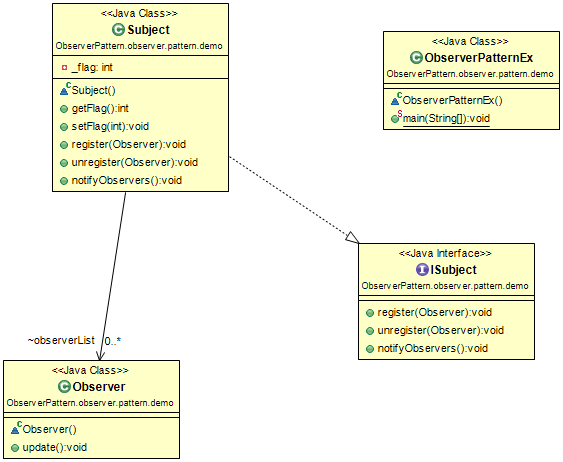
Vậy lúc này đối tượng A lại viết một phương thức để thông báo sự thay đổi dữ liệu cho đối tượng C biết. và cứ thế nếu có 100 đối tượng cần biết đến sự thay đổi bên trong của đối tượng A thì ta phải viết một 100 phương thức để gửi thông báo đến từng đối tượng. Vấn đề trở nên nan giải vì có sự trùng lặp không cần thiết.



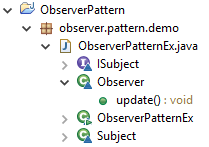
Thêm một rắc rối xảy ra khi giải quyết bài toán trên theo cách thông thường đó là mỗi lần không muốn thông báo đến một đối tượng nào từ A cho đến N thì ta phải đi rà soát lại từ đầu đến cuối để tìm ra phương thứ notify cho đối tượng đó. Tốn rất nhiều thời gian và rất không khả thi.

Sự xuất hiện của mẫu Observer đã giúp chúng ta giải quyết tất cả vấn đề trên.  
Mẫu observer lưu trữ các đối tượng cần quan tấm trong một danh sách. Và quan trọng hơn đó là có thể giúp chúng ta thêm hoặc loại bỏ đi đối tượng không quan tâm với các phương thức do chính chúng ta định nghĩa.

### Sơ đồ UML



### Cấu trúc Package



### Vai trò các class tương ứng với cấu trúc của mẫu

Subject là một class có các phương thức cần thiết để thêm hoặc loại bỏ các observer trong danh sách. Ngoài ra còn có một nhiệm vụ quan trọng nữa là thông báo sự thay đổi đến các observer và gọi phương thức cập nhật bên trong của các observer.

Observer là một class đại diện cho các đối tượng đang theo dõi hoặc lắng nghe đối tượng chủ thể.

Isubject là một interface có chứa các phương thức trừu tượng. Để khi bất kể đối tượng làm chủ thể đc theo dõi thì phải implement interface này.

### Source Code minh họa

**class** Observer

{

//@Override

**public** **void** update()

{

System.***out***.println("flag value changed in Subject");

}

}

Class Observer chứa phương thức update dùng để update lại trạng thái khi có một notify từ Subject.

**interface** ISubject

{

**void** register(Observer o);

**void** unregister( Observer o);

**void** notifyObservers();

}

**class** Subject **implements** ISubject

{

//List<IObserver> observerList = new ArrayList<IObserver>();

List<Observer> observerList = **new** ArrayList<Observer>();

**private** **int** \_flag;

**public** **int** getFlag()

{

**return** \_flag;

}

**public** **void** setFlag(**int** \_flag)

{

**this**.\_flag=\_flag;

//flag value changed .So notify observer(s)

notifyObservers();

}

@Override

**public** **void** register(Observer o)

{

observerList.add(o);

}

@Override

**public** **void** unregister(Observer o)

{

observerList.remove(o);

}

@Override

**public** **void** notifyObservers()

{

**for**(**int** i=0;i<observerList.size();i++)

{

observerList.get(i).update();

}

}

}

ISubject là một Interface có các phương thức chung cho tất cả các Subject implement interface này.

* Phương thức register(Observer o) dùng để thêm một observer vào danh sách observer chứa trong Subject.
* Phương thức unregister(Observer o) dùng để loại một observer trong danh sách các observer hiện có.
* Phương thức notifyObservers dùng để thông báo đến tất cả các observer mỗi khi có một thay đổi nào đó. Trong phương thức này nó sẽ duyệt qua hết các phần từ có trong list observer và gọi phương thức update() của từng Observer.

Subject là 1 class implement các interface ISubject. Và nó sẽ implement các phương thức ở trên và thêm phương thức setFlag(int +flag) để thay đổi dữ liệu và gọi phương thức notifyObservers();

**class** ObserverPatternEx

{

**public** **static** **void** main(String[] args)

{

System.***out***.println("\*\*\*Observer Pattern Demo\*\*\*\n");

Observer o1 = **new** Observer();

Subject sub1 = **new** Subject();

sub1.register(o1);

System.***out***.println("Setting Flag = 5 ");

sub1.setFlag(5);

System.***out***.println("Setting Flag = 25 ");

sub1.setFlag(25);

sub1.unregister(o1);

//No notification this time to o1 .Since it is unregistered.

System.***out***.println("Setting Flag = 50 ");

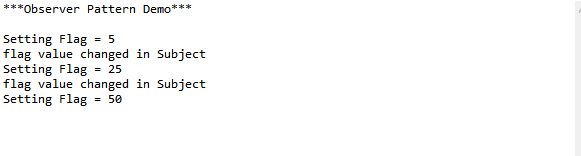
sub1.setFlag(50);

}

}

Hàm main ta tiến hành khởi tạo danh sách các observer và tiến hành thay đổi dữ liệu để gửi thông báo đến các Observers đã thêm vào.

Kết quả:



# MẪU STRATEGY

## Tên và phân loại

* Tên: Strategy.
* Phân loại: Behavioral Pattern.



## Mục đích, ý nghĩa

Đây là pattern cho phép các giải thuật khác nhau có thể được lựa chọn trong thời-gian-chạy (run-time). Hay nói cách khác, Strategy định nghĩa một họ các giải thuật khác nhau, mỗi giải thuật được triển khai bởi một lớp (class) cụ thể và chúng có thể hoán đổi cho nhau tùy vào ngữ cảnh. Strategy giúp các giải thuật khác nhau độc lập với client sử dụng nó.

## Bí danh

Mẫu strategy còn có tên gọi khác là Policy.

## Motivation

Một lớp thực hiện nhiệm vụ so sánh dữ liệu đầu vào có thể sử dụng mẫu thiết kế Strategy để tự động lựa chọn giải thuật cho việc này dựa trên loại dữ liệu, nguồn gốc của chúng, lựa chọn của người dùng hay các yếu tố khác. Những yếu tố này không được biết cho tới thời-gian-chạy (runtime) và khi đó tùy vào loại dữ liệu mà hệ thống lựa chọn cách thức so sánh khác nhau.

## Khả năng ứng dụng

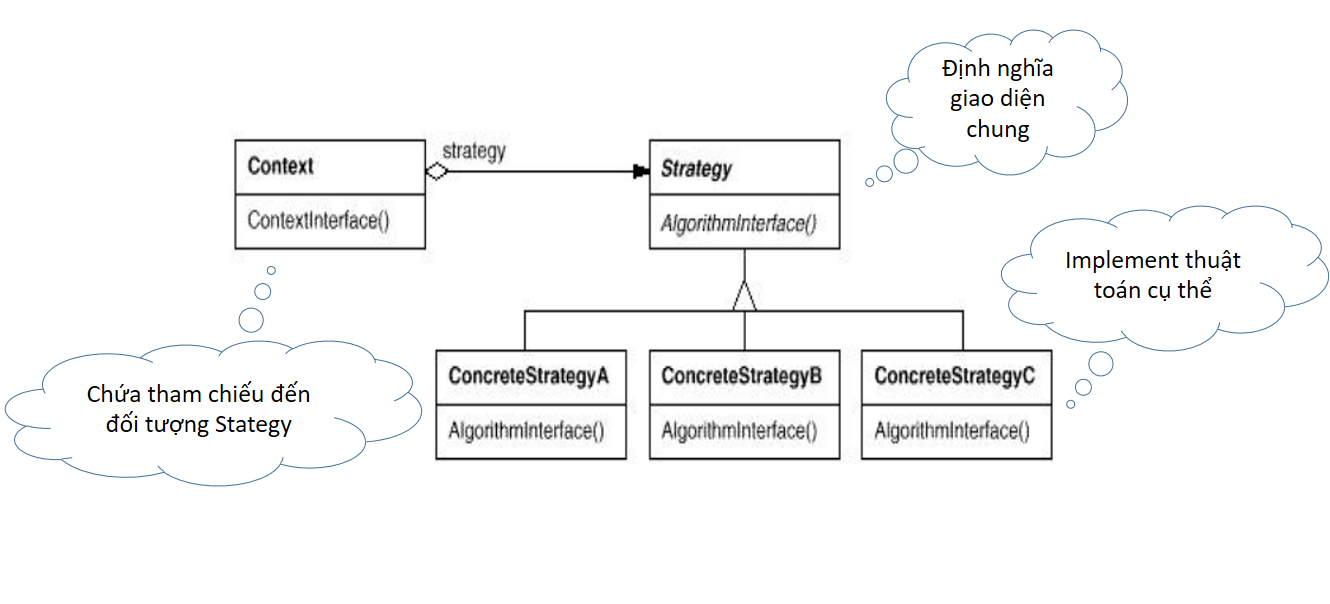
Nhiều lớp liên quan chỉ khác nhau ở cách xử lý yêu cầu. Với 1 lựa chọn trong những cách xử lý Strategy giúp ta thực hiện trách nhiệm của 1 lớp.

Có nhiều cách thực hiện cùng một thuật toán. Phải cho khách hàng khả năng lựa chọn cách ưu việt nhất trong sử dụng tài nguyên như vị trí và thời gian. Nên dùng Strategy khi các thuật toán này được thể hiện như một cơ cấu lớp của các thuật toán.

Thuật toán dùng dữ liệu mà khách hàng không biết tới. Dùng Strategy để thay thế việc công khai hoá những cấu trúc dữ liệu phức tạp, đặc thù cho thuật toán.

Khách hàng định nghĩa nhiều cách xử lý khác nhau và những cách xử lý này có thể coi nhưcâu lệnh chia nhánh (if- then- elsif, switch) trong phương thức. Thay vì dùng cấu trúc điều kiện ta dùng các lớp Strategy cài đặt riêng từng nhánh.

## Cấu trúc



## Sự cộng tác

Strategy (Compositor): định nghĩa giao diện chung cho các thuật toán được cài đặt.

Context dùng giao diện này để gọi những thuật toán được thực hiện trong những ConcreteStrategy.

ConcreteStrategy: implement giao diện do Strategy định nghĩa để cài đặt các thuật toán cụ thể.

## Các hệ quả mang lại

* Ưu điểm: Tập hợp được các thuật toán liên quan với nhau. Dễ dàng chuyển đổi và mở rộng. Lược bỏ được các câu lệnh điều kiện. Giúp cho client có thể lựa chọn giữa nhiều thuật toán
* Hạn chế: Client cần biết rõ về những Strategy để có sự lựa chọn phù hợp.

## Chú ý khi cài đặt

Có thể sử dụng phương thức compareTo() để sử dụng mẫu này.

## Hệ thống thực tế đang sử dụng

Sử dụng Interface có sẵn trong Java đó là java.lang.Comparable, interface này có phương thức compareTo() cho phép các lớp triển khai nó thực hiện việc so sánh hai đối tượng với nhau.

Có thể tìm hiểu thêm Interface này ở đây: http://docs.oracle.com/javase/6/docs/api/java/lang/Comparable.html

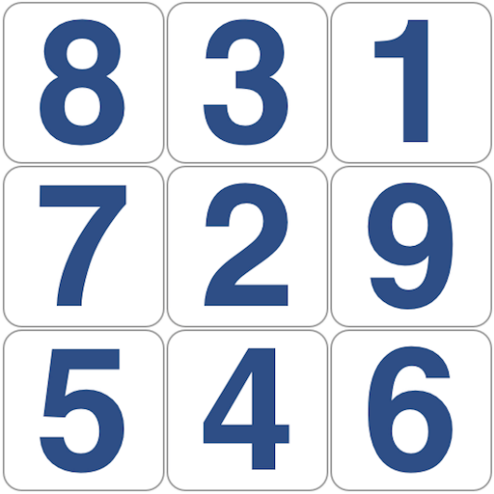
## Mẫu liên quan

Tương đối giống với mẫu State, nhưng mẫu State được sử dụng để điều chỉnh hành vi của đối tượng khi trạng thái của đối tượng đó thay đổi.

## Mã nguồn minh họa

### Mô tả bài toán

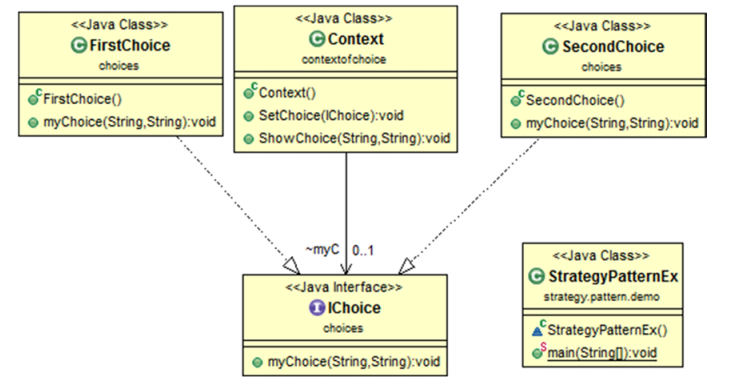
Áp dụng kiến thức về Design Pattern. Hãy viết chương trình yêu cầu nhập vào 2 số sau đó yêu cầu chọn hành động thực hiện nối chuỗi hay cộng 2 số.



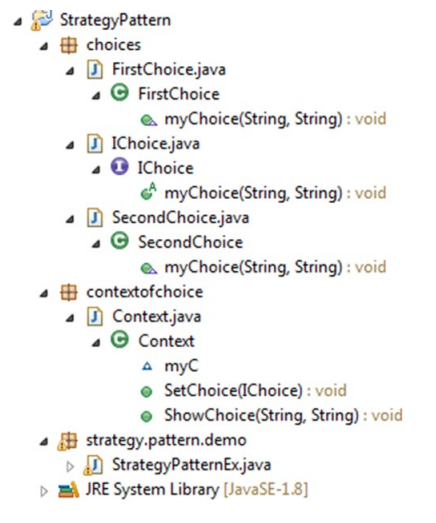
### Tại sao sử dụng mẫu này?

Tùy vào yêu cầu của client lúc runtime mà hành vi khác nhau là thực hiện nối chuỗi hay thực hiện cộng 2 số.

### Sơ đồ UML



### Cấu trúc Package



### Vai trò các class tương ứng với cấu trúc của mẫu

ConcreteStrategy là class First Choise, Second Choise.

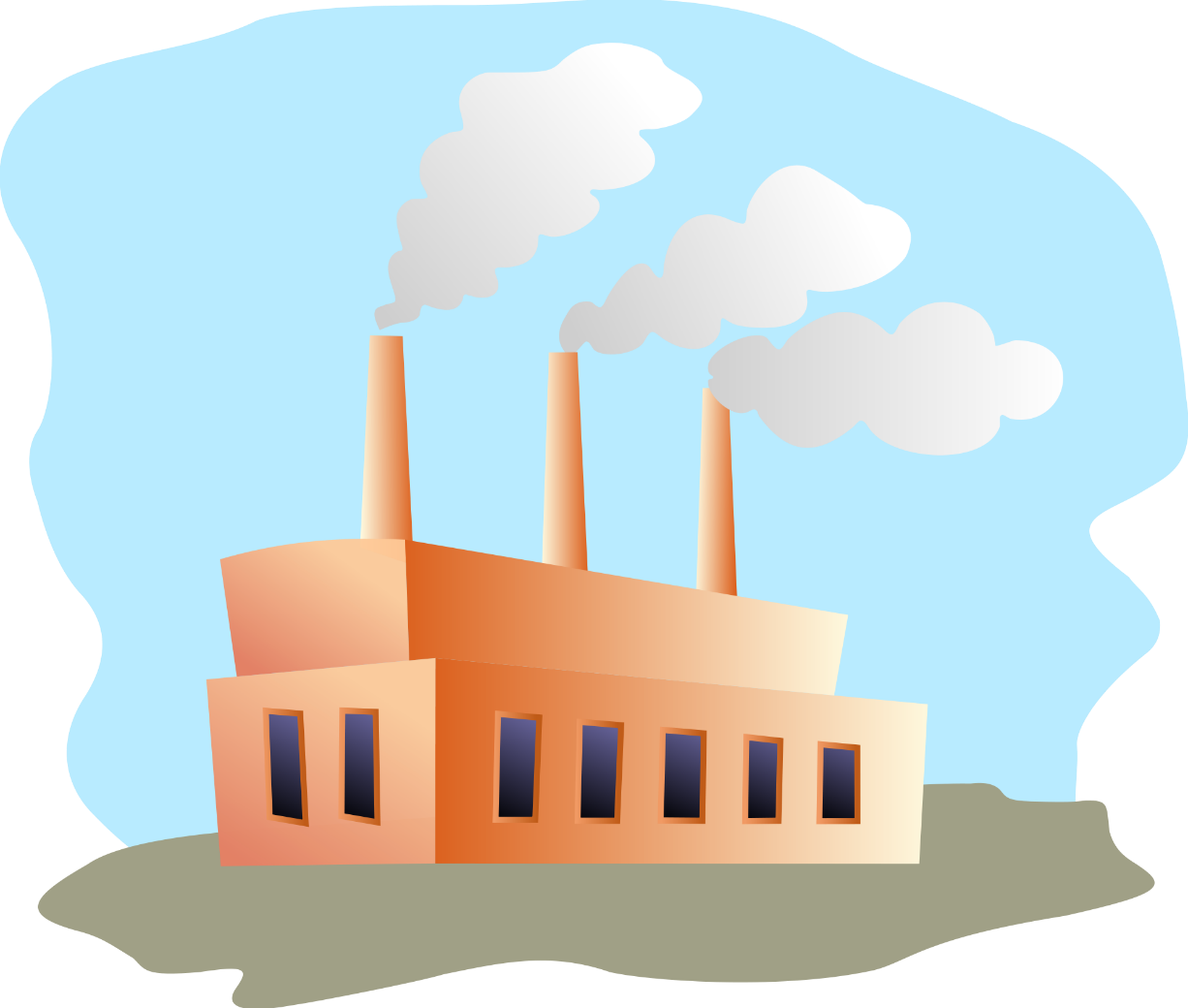
Context là class Context.

Ichoise là class strategy.

# MẪU FACTORY

## Tên và phân loại

* Tên: Factory.
* Phân loại: Creational Pattern.



## Mục đích, ý nghĩa

Factory Method định nghĩa một interface cho phép tạo ra đối tượng nhưng nó để cho lớp con quyết định khởi tạo thể hiện của lớp nào.

Factory Pattern định nghĩa một lớp (interface, abstract class) đóng vai trò như một "nhà xưởng" có nhiệm vụ khởi tạo đối tượng “cụ thể” khi ứng dụng chạy. Tại thời điểm thiết kế đối tượng này được định nghĩa trừu tượng.

Hay nói cách khác: Mẫu này tạo Giao diện (Interface) có vai trò như một "nhà xưởng". Và chỉ tạo đối tượng (object/instance) cụ thể khi chạy chương trình.

Mức độ sử dụng của method factory cũng tương đối thường gặp giống như abstract factory.

## Bí danh

Không có.

## Motivation

Giả sử có 1 công ty chuyên về sản xuất bánh ngọt nhờ bạn viết 1 phần mềm với yêu cầu như sau:

* + Công ty chỉ sản xuất xuất bánh tét vào tháng 12 và tháng 1.
  + Công ty chỉ sản xuất bánh trung thu vào tháng 8 và tháng 9.
  + Những tháng còn lại công ty sản xuất bánh bông lan.

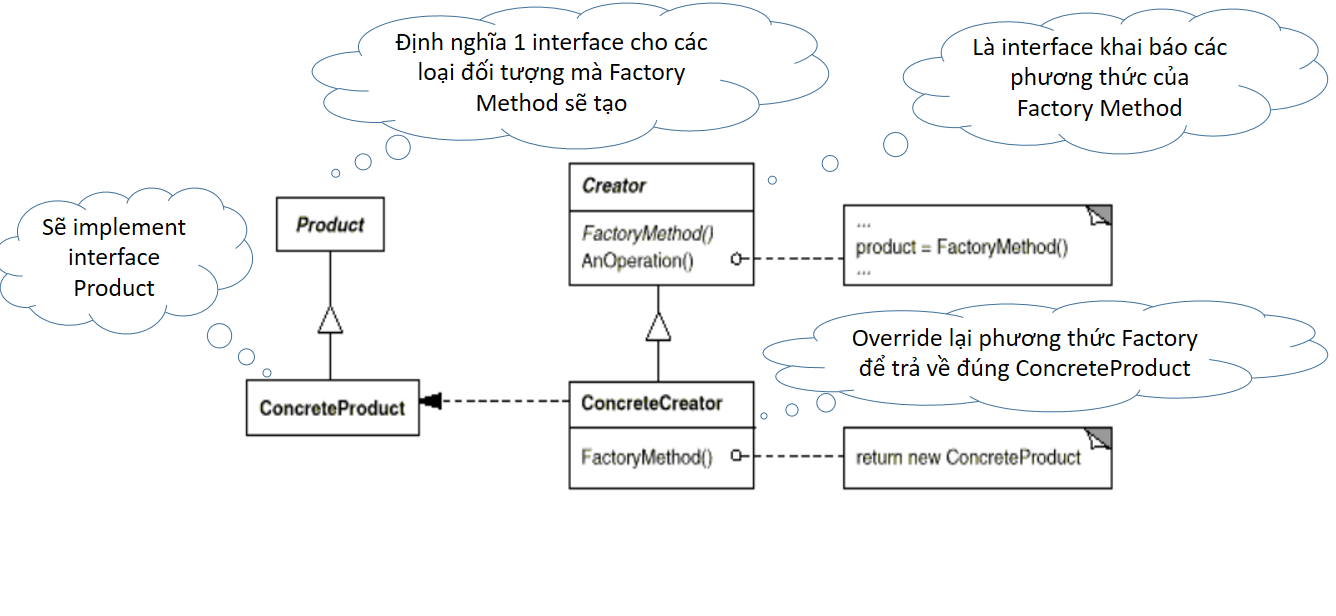
## Khả năng ứng dụng

Chúng ta ứng dụng Factory method pattern trong những trường hợp sau:

* Khi bạn muốn tạo một đối tượng và loại bớt những đoạn code bị trùng lặp (trong trường hợp trên là việc khai báo hàm GetCakeName(), chỉ có thân hàm là khác biệt).
* Tạo tính linh động cho đoạn việc tạo đối tượng.
* Tách biệt việc tạo đối tượng với logic chính của chương trình.

Với những tính năng trên, việc sử dụng Factory Method Pattern một cách hợp lý sẽ giúp code của bạn trông sáng sủa hơn, tính linh động cao hơn.

## Cấu trúc



## Sự cộng tác

Product: định nghĩa interface của đối tượng mà phương thức factory sẽ sinh ra.

ConcreteProduct: Cài đặt phương thức quy định trong Product.

Creator: Khai báo phương thức factory (trả về đôi tượng kiểu Product), Creator cũng có thể định nghĩa phần cài đặt cho phương thức factory mặc định trả về ConcreteProduct mặc định.

ConcreteCreator: override phương thức factory của Creator và trả về ConcreteProduct tương ứng.

## Các hệ quả mang lại

Ưu điểm:

* Tạo sự linh hoạt trong việc sử dụng lại code bằng cách loại bỏ việc tạo ra các lớp ứng dụng cụ thể, chỉ thao tác với interface Product và có thể làm việc với mọi ConcreteProduct hỗ trợ interface này.
* Client thao tác dựa trên interface mà không quan tâm sản phẩm gì được trả về, quá trình tạo sản phẩm được che dấu hoàn toàn.
* Dễ dàng cập nhật phần mềm.

Khuyết điểm:

* Hạn chế có thể thấy ngay là một khi cần tạo một loại đối tượng mới, ta phải thừa kế lại lớp Creator và cài đặt phương thức khởi tạo.
* Khi thêm một sản phẩm mới, ta phải sửa đổi và bổ sung mã chương trình khá nhiều điều này làm gia tăng độ phức tạp của hệ thống.

## Chú ý khi cài đặt

Khi cài đặt, ngoài trường hợp có Creator có thể là abstract class và có ConcreteCreator kế thừa từ nó thì cũng có thể cài đặt chỉ có ConcreteCreator.

## Hệ thống thực tế đang sử dụng

Trong package javax.xml.parsers, ví dụ như: javax.xml.parsers.DocumentBuilderFactory hay javax.xml.parsers.SAXParserFactory.

## Mẫu liên quan

Mẫu Abstract Factory tương tự như mẫu Factory Method, nhưng mẫu Abstract Factory cho phép tạo ra được các họ các sản phẩm có liên quan với nhau.

## Mã nguồn minh họa

### Mô tả bài toán

Áp dụng kiến thức về Design Pattern. Hãy viết chương trình sẽ thực hiện hiện ra tiếng kêu của từng con vật đã được cài đặt tiếng kêu.

Client chỉ ra lệnh là cho con vật nào kêu chứ không quan tâm và không cần biết đến việc khởi tạo các đối tượng con vật đó.



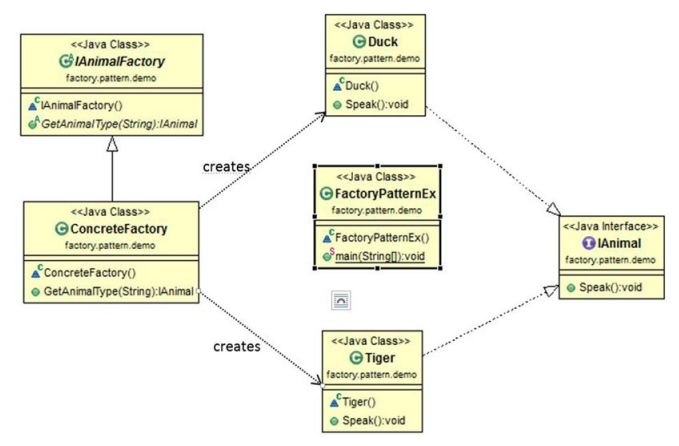
### Tại sao sử dụng mẫu này?

Cần khởi tạo nhiều đối tượng.

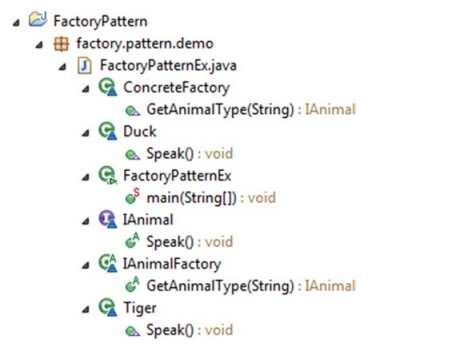
Việc đối tượng nào được tạo ra chỉ khi đến thời điểm runtime mới được biết.

Client không đảm nhận trách nhiệm khởi tạo đối tượng và cũng không quan tâm là class nào thực hiện điều đó.

### Sơ đồ UML



### Cấu trúc Package



### Vai trò các class tương ứng với cấu trúc của mẫu

* Product là Interface Ianimal.
* ConcreteProduct là class Duck và Tiger.
* ConcreteCreator là class ConcreteFactory.
* Creator là interface IanimalFactory.