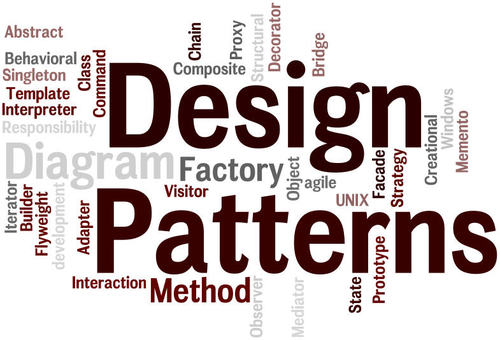
# Giới thiệu Design Patterns

Đăng vào 30/08/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 5505 Lượt xem

Có thể chúng ta đã gặp [Design patterns](https://gpcoder.com/category/design-pattern/) ở đâu đó trong các ứng dụng, cũng có thể chúng ta đã từng sử dụng những mẫu tương tự như Design pattern để giải quyết những tình huống của mình, nhưng chúng ta không rõ hoặc không có một khái niệm gì về nó. Trong bài này, chúng ta sẽ cùng tìm hiểu một số kiến thức tổng quan Design Pattern, sau đó sẽ tìm hiểu chi tiết về từng Design Pattern trong các bài viết tiếp theo.



Nội dung [[Ẩn](https://gpcoder.com/4164-gioi-thieu-design-patterns/)]

* [1 Design Patterns là gì?](https://gpcoder.com/4164-gioi-thieu-design-patterns/" \l "Design_Patterns_la_gi)
* [2 Tại sao phải sử dụng Design Patterns?](https://gpcoder.com/4164-gioi-thieu-design-patterns/" \l "Tai_sao_phai_su_dung_Design_Patterns)
* [3 Khi nào nên sử dụng Design patterns?](https://gpcoder.com/4164-gioi-thieu-design-patterns/" \l "Khi_nao_nen_su_dung_Design_patterns)
* [4 Phân loại Design Patterns](https://gpcoder.com/4164-gioi-thieu-design-patterns/" \l "Phan_loai_Design_Patterns)
  + [4.1 Nhóm Creational (nhóm khởi tạo)](https://gpcoder.com/4164-gioi-thieu-design-patterns/" \l "Nhom_Creational_nhom_khoi_tao)
  + [4.2 Nhóm Structural (nhóm cấu trúc)](https://gpcoder.com/4164-gioi-thieu-design-patterns/" \l "Nhom_Structural_nhom_cau_truc)
  + [4.3 Nhóm Behavioral (nhóm hành vi/ tương tác)](https://gpcoder.com/4164-gioi-thieu-design-patterns/" \l "Nhom_Behavioral_nhom_hanh_vi_tuong_tac)
* [5 Để học Design Patterns cần có gì?](https://gpcoder.com/4164-gioi-thieu-design-patterns/" \l "De_hoc_Design_Patterns_can_co_gi)
* [6 Học ở đâu?](https://gpcoder.com/4164-gioi-thieu-design-patterns/" \l "Hoc_o_dau)
* [7 Học Design Pattern như thế nào?](https://gpcoder.com/4164-gioi-thieu-design-patterns/" \l "Hoc_Design_Pattern_nhu_the_nao)
* [8 Lời kết](https://gpcoder.com/4164-gioi-thieu-design-patterns/" \l "Loi_ket)

## Design Patterns là gì?

[Design Patterns](https://vi.wikipedia.org/wiki/Design_Patterns) (mẫu thiết kế) là một kỹ thuật trong lập trình hướng đối tượng, nó khá quan trọng và mọi lập trình viên muốn giỏi đều phải biết. Design Pattern được sử dụng thường xuyên trong các ngôn ngữ [OOP](https://gpcoder.com/category/java-core/oop/). Nó cung cấp cho bạn các “mẫu thiết kế”, giải pháp để giải quyết các vấn đề chung, thường gặp trong lập trình. 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.

Design Patterns không phải là ngôn ngữ cụ thể nào cả. Nó có thể thực hiện được ở phần lớn các ngôn ngữ lập trình, chẳng hạn như Java, C#, thậm chí là Javascript hay bất kỳ ngôn ngữ lập trình nào khác.

Mỗi pattern mô tả một vấn đề xảy ra lặp đi lặp lại, và trình bày trọng tâm của giải pháp cho vấn đề đó, theo cách mà bạn có thể dùng đi dùng lại hàng triệu lần mà không cần phải suy nghĩ.

— Christopher Alexander —

## Tại sao phải sử dụng Design Patterns?

Design Pattern giúp bạn tái sử dụng mã lệnh và dẽ dàng mở rộng.

Nó là tập hơn những giải pháp đã được tối ưu hóa, đã được kiểm chứng để giải quyết các vấn đề trong software engineering. Vậy khi bạn gặp bất kỳ khó khăn gì, design patterns là kim chỉ nam giúp bạn giải quyết vấn đề thay vì tự tìm kiếm giải pháp cho một vấn đề đã được chứng minh.

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.

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.

Giúp cho các lập trình viên có thể hiểu code của người khác 1 cách nhanh chóng (có thể hiểu là tính communicate). Mọi thành viên trong team có thể dễ dàng trao đổi với nhau để cùng xây dựng dự án mà không mất quá nhiều thời gian.

## Khi nào nên sử dụng Design patterns?

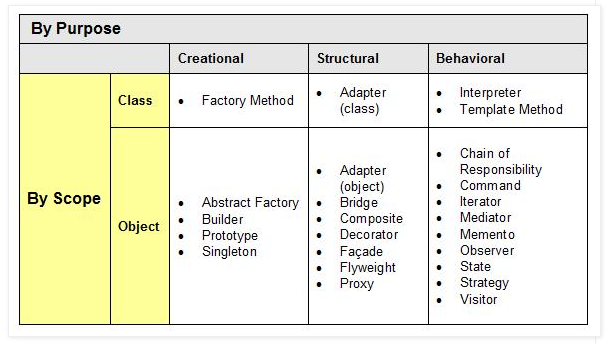
Khi bạn muốn giữ cho chương trình của mình thực sự đơn giản. Việc sử dụng các design pattern sẽ giúp chúng ta giảm được thời gian và công sức suy nghĩ ra các cách giải quyết cho những vấn đề đã có lời giải.

## Phân loại Design Patterns

Năm 1994, bốn tác giả Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides đã cho xuất bản một cuốn sách với tiêu đề Design Patterns – Elements of Reusable Object-Oriented Software, đây là khởi nguồn của khái niệm design pattern trong lập trình phần mềm.

Bốn tác giả trên được biết đến rộng rãi dưới tên Gang of Four (bộ tứ). Theo quan điểm của bốn người, design pattern chủ yếu được dựa theo những quy tắc sau đây về thiết kế hướng đối tượng.

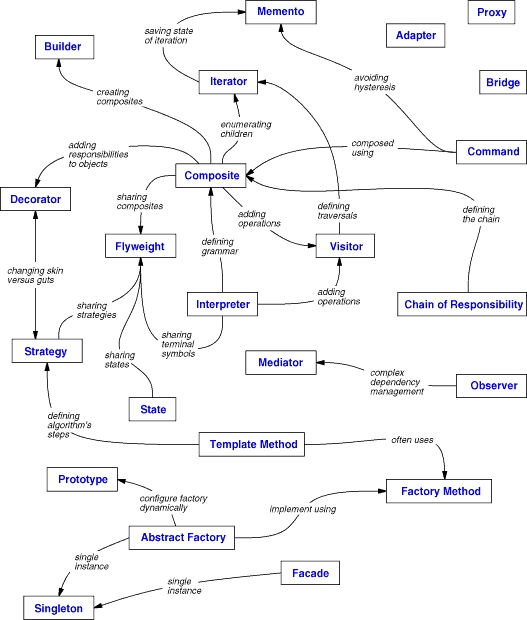
* [Lập trình cho interface chứ không phải để implement interface đó](http://stackoverflow.com/questions/2697783/what-does-program-to-interfaces-not-implementations-mean).
* [Ưu tiên object composition hơn là thừa kế](http://stackoverflow.com/questions/49002/prefer-composition-over-inheritance).



Hệ thống các mẫu Design pattern hiện có 23 mẫu được định nghĩa trong cuốn “Design patterns Elements of Reusable Object Oriented Software” và được chia thành 3 nhóm:

* [Creational Pattern](https://gpcoder.com/category/design-pattern/creational-pattern/)(nhóm khởi tạo – 5 mẫu) gồm: Factory Method, Abstract Factory, Builder, Prototype, Singleton. Những Design pattern loại này cung cấp một giải pháp để tạo ra các object và che giấu được logic của việc tạo ra nó, thay vì tạo ra object một cách trực tiếp bằng cách sử dụng method new. Điều này giúp cho chương trình trở nên mềm dẻo hơn trong việc quyết định object nào cần được tạo ra trong những tình huống được đưa ra.
* [Structural Pattern](https://gpcoder.com/category/design-pattern/structuaral-pattern/) (nhóm cấu trúc – 7 mẫu) gồm: Adapter, Bridge, Composite, Decorator, Facade, Flyweight và Proxy. Những Design pattern loại này liên quan tới class và các thành phần của object. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng.
* [Behavioral Pattern](https://gpcoder.com/category/design-pattern/behavior-pattern/) (nhóm tương tác/ hành vi – 11 mẫu) 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, sự giao tiếp giữa các object với nhau.

Hình dưới là mối quan hệ giữa 23 Design Pattern cơ bản (GoF):

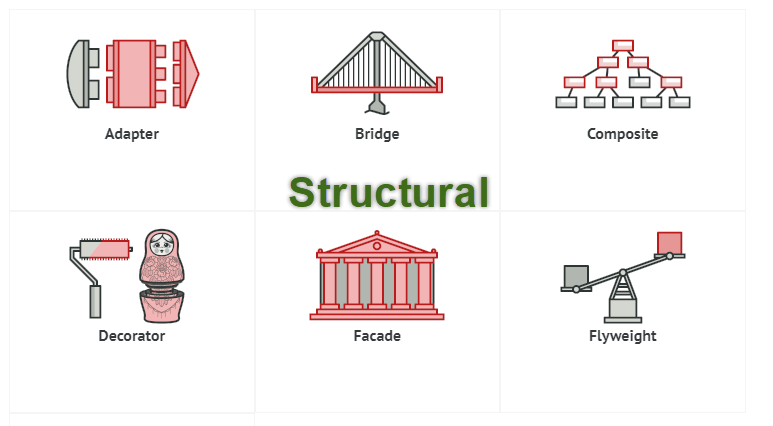


### Nhóm Creational (nhóm khởi tạo)



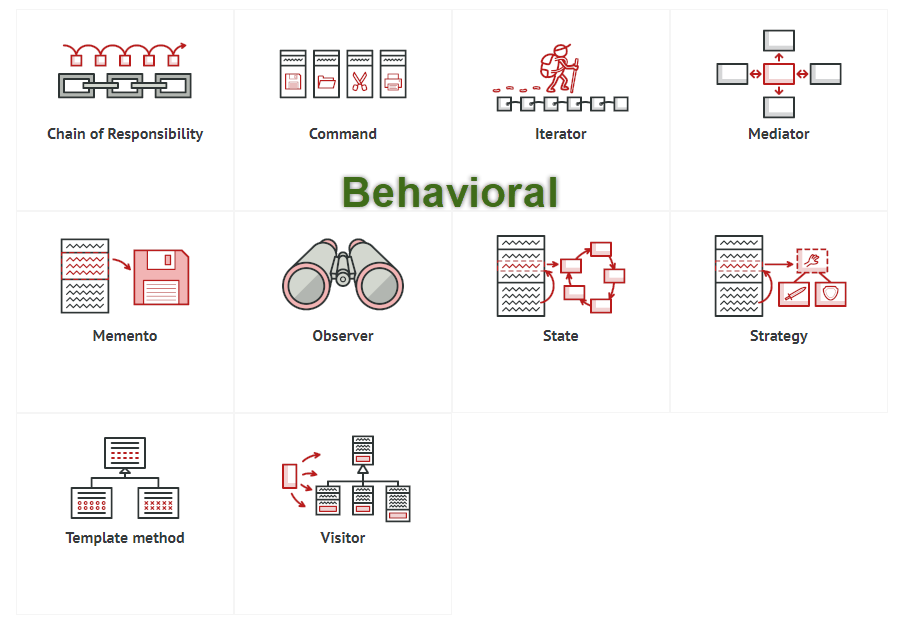
* [Singleton](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/):
  + Đảm bảo 1 class chỉ có 1 instance và cung cấp 1 điểm truy xuất toàn cục đến nó.
  + Tần suất sử dụng: cao trung bình.
* [Abstract Factory](https://gpcoder.com/4365-huong-dan-java-design-pattern-abstract-factory/):
  + 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.
* [Factory Method](https://gpcoder.com/4352-huong-dan-java-design-pattern-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.
* [Builder](https://gpcoder.com/4434-huong-dan-java-design-pattern-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.
* [Prototype](https://gpcoder.com/4413-huong-dan-java-design-pattern-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.

### Nhóm Structural (nhóm cấu trúc)



* [Adapter](https://gpcoder.com/4483-huong-dan-java-design-pattern-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.
* [Bridge](https://gpcoder.com/4520-huong-dan-java-design-pattern-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.
* [Composite](https://gpcoder.com/4554-huong-dan-java-design-pattern-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.
* [Decorator](https://gpcoder.com/4574-huong-dan-java-design-pattern-decorator/):
  + Gán thêm trách nhiệm cho đối tượng (mở rộng chức năng) vào lúc chạy (dynamically).
  + Tần suất sử dụng:trung bình.
* [Facade](https://gpcoder.com/4604-huong-dan-java-design-pattern-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.
* [Flyweight](https://gpcoder.com/4626-huong-dan-java-design-pattern-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.
* [Proxy](https://gpcoder.com/4644-huong-dan-java-design-pattern-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 hành vi/ tương tác)



* [Chain of Responsibility](https://gpcoder.com/4665-huong-dan-java-design-pattern-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 gửi request xuyên qua từng đối tượng xử lý đến khi gặp đối tượng xử lý cụ thể.
  + Tần suất sử dụng: trung bình thấp.
* [Command](https://gpcoder.com/4686-huong-dan-java-design-pattern-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.
* [Interpreter](https://gpcoder.com/4702-huong-dan-java-design-pattern-interpreter/):
  + Hỗ trợ việc định nghĩa biểu diễn văn phạm và bộ thông dịch cho một ngôn ngữ.
  + Tần suất sử dụng: thấp.
* [Iterator](https://gpcoder.com/4724-huong-dan-java-design-pattern-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.
* [Mediator](https://gpcoder.com/4740-huong-dan-java-design-pattern-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.
* [Memento](https://gpcoder.com/4763-huong-dan-java-design-pattern-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.
* [Observer](https://gpcoder.com/4747-huong-dan-java-design-pattern-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.
* [State](https://gpcoder.com/4785-huong-dan-java-design-pattern-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.
* [Strategy](https://gpcoder.com/4796-huong-dan-java-design-pattern-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.
* [Template method](https://gpcoder.com/4810-huong-dan-java-design-pattern-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.
* [Visitor](https://gpcoder.com/4813-huong-dan-java-design-pattern-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.

## Để học Design Patterns cần có gì?

Để học tốt Design Pattern bạn nắm vững được [Bốn đặc tính của OOP](https://gpcoder.com/2232-4-tinh-chat-cua-lap-trinh-huong-doi-tuong-trong-java/): [Kế thừa](https://gpcoder.com/2376-tinh-ke-thua-inheritance-trong-java/), [Đa hình](https://gpcoder.com/2386-tinh-da-hinh-polymorphism-trong-java/), [Trừu tượng](https://gpcoder.com/2388-tinh-truu-tuong-abstraction-trong-java/), [Bao đóng](https://gpcoder.com/2365-tinh-dong-goi-encapsulation-trong-java/).

Tiếp theo là hai khái niệm [interface và abstract](https://gpcoder.com/2295-abstract-class-va-interface-trong-java/) : 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](https://gpcoder.com/category/java-core/oop/).

Sau khi bạn đã nắm chắc các thứ kể trên, bạn nên bắt tay vào học Design Pattern càng sớm càng tốt. Design Pattern giúp bạn đặt trình cao mới :).

## Học ở đâu?

Sách:

* Design Patterns: Elements of Reusable Object-Oriented Software
* Head First Design Patterns
* Design Patterns For Dummies
* Pattern Hatching: Design Patterns Applied.
* Refactoring to Patterns.
* Patterns of Enterprise Application Architecture.

Các website viết về Design Pattern:

* <https://sourcemaking.com/design_patterns>
* [https://refactoring.guru](https://refactoring.guru/)
* <https://www.tutorialspoint.com/design_pattern/index.htm>
* <https://www.javatpoint.com/design-patterns-in-java>
* <http://www.oodesign.com/>
* [http://java-design-patterns.com](http://java-design-patterns.com/)
* <http://www.javacamp.org/designPattern/>
* <http://www.thejavageek.com/design-patterns/>
* <https://en.wikipedia.org/wiki/Software_design_pattern>
* dotNet Design Patterns: <https://www.dofactory.com/net/design-patterns>
* Javascript Design Patterns: <https://www.dofactory.com/javascript/design-patterns>

Theo dõi các bài viết tiếp theo trên [GP Coder](https://gpcoder.com/) :).

## Học Design Pattern như thế nào?

Để học tốt một Design Pattern bạn hãy tập trung chú ý vào 3 phần sau:

* Nó được sử dụng khi nào, vấn đề mà design pattern đó giải quyết là gì?
* Sơ đồ UML mô tả design pattern.
* Code minh họa, ứng dụng thực tiễn của nó là gì?

## Lời kết

Design pattern thể hiện tính kinh nghiệm của công việc lập trình, xây dựng và thiết kế phần mềm. Người hiểu và vận dụng được Design Pattern thích hợp trong hệ thống sẽ tiết kiệm được rất nhiều thời gian, công sức, dễ phát triển, mở rộng, bảo trì. Tuy nhiên không nên quá lạm dụng nó.

Chúng ta nên nhớ rằng:

* Design patterns là 1 đoạn mô tả, hoặc 1 khuôn mẫu để giải quyết 1 vấn đề nào đó. Nó không phải là thiết kế cuối cùng.
* Design Pattern được tạo ra để giải quyết vấn đề, chứ không phải để phức tạp hóa nó.
* Design Pattern giúp code được tối ưu hóa, dễ tái sử dụng, dễ hiểu, dễ nâng cấp sửa chữa.

Qua bài này mình hi vọng mọi người có thể hiểu hơn về design pattern và ưu điểm của nó. Bài tiếp theo chúng ta sẽ đi vào tìm hiểu từng mẫu Design Patterns. Vào chi tiết, nội dung tôi sẽ sử dụng thông tin chủ yếu từ trang [SourceMaking](https://sourcemaking.com/design_patterns), [TutorialsPoint](https://www.tutorialspoint.com/design_pattern/index.htm) , sách Design Patterns – Elements of Reusable Object-Oriented Software (GOF), Design Pattern for dummies và tìm hiểu thêm những kiến thức từ nhiều trang khác để các bạn dễ hiểu hơn. Vì đây là những kiến thức nâng cao nên tôi cũng phải tìm hiểu từ rất nhiều nguồn để tổng hợp lại, cũng như những kinh nghiệm trong các dự án thực tế vào bài viết để giúp cho các bạn dễ tiếp cận nhất có thể. Mong nhận được sự phản hồi và góp ý của các bạn. *Thanks a lot*!!!

Tài liệu tham khảo:

* <https://toidicodedao.com/2016/03/01/nhap-mon-design-pattern-phong-cach-kiem-hiep/>
* <https://tuanitpro.com/design-pattern-la-gi>
* <https://blog.duyet.net/2015/02/oop-design-patterns-la-gi.html>
* <https://refactoring.guru/design-patterns/catalog>
* <https://www.ibm.com/developerworks/java/tutorials/j-patterns/j-patterns.html>
* <https://www.journaldev.com/1827/java-design-patterns-example-tutorial>

# Hướng dẫn Java Design Pattern – Singleton

Đăng vào 08/09/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 2500 Lượt xem

Đôi khi, trong quá trình phân tích thiết kế một hệ thống, chúng ta mong muốn có những đối tượng cần tồn tại duy nhất và có thể truy xuất mọi lúc mọi nơi. Làm thế nào để hiện thực được một đối tượng như thế khi xây dựng mã nguồn? Chúng ta có thể nghĩ tới việc sử dụng một biến toàn cục (global variable : public static final). Tuy nhiên, việc sử dụng biến toàn cục nó phá vỡ quy tắc của OOP (encapsulation). Để giải bài toán trên, người ta hướng đến một giải pháp là sử dụng Singleton pattern.

Nội dung [[Ẩn](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/)]

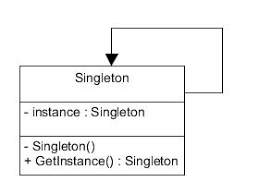
* [1 Singleton Pattern là gì?](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/" \l "Singleton_Pattern_la_gi)
* [2 Implement Singleton Pattern như thế nào?](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/" \l "Implement_Singleton_Pattern_nhu_the_nao)
* [3 Những cách nào để implement Singleton Pattern](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/" \l "Nhung_cach_nao_de_implement_Singleton_Pattern)
  + [3.1 Eager initialization](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/" \l "Eager_initialization)
  + [3.2 Static block initialization](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/" \l "Static_block_initialization)
  + [3.3 Lazy Initialization](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/" \l "Lazy_Initialization)
  + [3.4 Thread Safe Singleton](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/" \l "Thread_Safe_Singleton)
  + [3.5 Double Check Locking Singleton](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/" \l "Double_Check_Locking_Singleton)
  + [3.6 Bill Pugh Singleton Implementation](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/" \l "Bill_Pugh_Singleton_Implementation)
  + [3.7 Phá vỡ cấu trúc Singleton Pattern bằng Reflection](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/" \l "Pha_vo_cau_truc_Singleton_Pattern_bang_Reflection)
  + [3.8 Enum Singleton](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/" \l "Enum_Singleton)
  + [3.9 Serialization and Singleton](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/" \l "Serialization_and_Singleton)
* [4 Sử dụng Singleton Pattern khi nào?](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/" \l "Su_dung_Singleton_Pattern_khi_nao)
* [5 Tổng kết](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/" \l "Tong_ket)

## Singleton Pattern là gì?

[Singleton](https://en.wikipedia.org/wiki/Singleton_pattern) is a creational design pattern that lets you ensure that a class has only one instance and provide a global access point to this instance.

Singleton là 1 trong 5 design pattern của nhóm Creational Design Pattern.

Singleton đảm bảo chỉ duy nhất một thể hiện (instance) được tạo ra và nó sẽ cung cấp cho bạn một method để có thể truy xuất được thể hiện duy nhất đó mọi lúc mọi nơi trong chương trình.



Sử dụng Singleton khi chúng ta muốn:

* Đảm bảo rằng chỉ có một instance của lớp.
* Việc quản lý việc truy cập tốt hơn vì chỉ có một thể hiện duy nhất.
* Có thể quản lý số lượng thể hiện của một lớp trong giớn hạn chỉ định.

## Implement Singleton Pattern như thế nào?

Có rất nhiều cách để implement Singleton Pattern. Nhưng dù cho việc implement bằng cách nào đi nữa cũng dựa vào nguyên tắc dưới đây cơ bản dưới đây:

* private constructor để hạn chế truy cập từ class bên ngoài.
* Đặt private static final variable đảm bảo biến chỉ được khởi tạo trong class.
* Có một method public static để return instance được khởi tạo ở trên.

## Những cách nào để implement Singleton Pattern

Dựa trên những nguyên tắc thiết kế Singleton ở trên, chúng ta có các cách implement singleton như sau:

### Eager initialization

Singleton Class được khởi tạo ngay khi được gọi đến. Đây là cách dễ nhất nhưng nó có một nhược điểm mặc dù instance đã được khởi tạo mà có thể sẽ không dùng tới.

Ví dụ:

package com.gpcoder.patterns.creational.singleton;

public class EagerInitializedSingleton {

private static final EagerInitializedSingleton INSTANCE = new EagerInitializedSingleton();

// Private constructor to avoid client applications to use constructor

private EagerInitializedSingleton() {

}

public static EagerInitializedSingleton getInstance() {

return INSTANCE;

}

}

### Static block initialization

Cách làm tương tự như Eager initialization chỉ khác phần static block cung cấp thêm lựa chọn cho việc handle exception hay các xử lý khác.

Ví dụ:

package com.gpcoder.patterns.creational.singleton;

public class StaticBlockSingleton {

private static final StaticBlockSingleton INSTANCE;

private StaticBlockSingleton() {

}

// Static block initialization for exception handling

static {

try {

INSTANCE = new StaticBlockSingleton();

} catch (Exception e) {

throw new RuntimeException("Exception occured in creating singleton instance");

}

}

public static StaticBlockSingleton getInstance() {

return INSTANCE;

}

}

### Lazy Initialization

Là một cách làm mang tính mở rộng hơn so với 2 cách làm trên và hoạt động tốt trong môi trường đơn luồng (single-thread).

Ví dụ:

package com.gpcoder.patterns.creational.singleton;

public class LazyInitializedSingleton {

private static LazyInitializedSingleton instance;

private LazyInitializedSingleton() {

}

public static LazyInitializedSingleton getInstance() {

if (instance == null) {

instance = new LazyInitializedSingleton();

}

return instance;

}

}

Cách này đã khắc phục được nhược điểm của cách Eager initialization, chỉ khi nào getInstance() được gọi thì instance mới được khởi tạo. Tuy nhiên, cách này chỉ sử dụng tốt trong trường hợp đơn luồng (single-thread), trường hợp nếu có nhiều luồng (multi-thread) cùng chạy và cùng gọi hàm getInstance() tại cùng một thời điểm thì có thể có nhiều hơn 1 thể hiện của instance. Để khắc phục nhược điểm này chúng ta sử dụng Thread Safe Singleton.

Một nhược điểm nữa của Lazy Initialization cần quan tâm là: đối với thao tác create instance quá chậm thì người dùng có phải chờ lâu cho lần sử dụng đầu tiên.

### Thread Safe Singleton

Cách đơn giản nhất là chúng ta gọi phương thức synchronized của hàm getInstance() và như vậy hệ thống đảm bảo rằng tại cùng một thời điểm chỉ có thể có 1 luồng có thể truy cập vào hàm getInstance() và đảm bảo rằng chỉ có duy nhất 1 thể hiện của class.

Ví dụ:

package com.gpcoder.patterns.creational.singleton;

public class ThreadSafeLazyInitializedSingleton {

private static volatile ThreadSafeLazyInitializedSingleton instance;

private ThreadSafeLazyInitializedSingleton() {

}

public static synchronized ThreadSafeLazyInitializedSingleton getInstance() {

if (instance == null) {

instance = new ThreadSafeLazyInitializedSingleton();

}

return instance;

}

}

Biến volatile trong Java có tác dụng thông báo sự thay đổi giá trị của biến tới các thread khác nhau nếu biến này đang được sử dụng trong nhiều thread.

Cách này có nhược điểm là một phương thức synchronized sẽ chạy rất chậm và tốn hiệu năng, bất kỳ Thread nào gọi đến đều phải chờ nếu có một Thread khác đang sử dụng. Có những tác vụ xử lý trước và sau khi tạo thể hiện không cần thiết phải block. Vì vậy chúng ta cần cải tiến nó đi 1 chút với Double Check Locking Singleton.

### Double Check Locking Singleton

Để implement theo cách này, chúng ta sẽ kiểm tra sự tồn tại thể hiện của lớp, với sự hổ trợ của đồng bộ hóa, hai lần trước khi khởi tạo. Phải khai báo volatile cho instance để tránh lớp làm việc không chính xác do quá trình tối ưu hóa của trình biên dịch.

package com.gpcoder.patterns.creational.singleton;

public class DoubleCheckLockingSingleton {

private static volatile DoubleCheckLockingSingleton instance;

private DoubleCheckLockingSingleton() {

}

public static DoubleCheckLockingSingleton getInstance() {

// Do something before get instance ...

if (instance == null) {

// Do the task too long before create instance ...

// Block so other threads cannot come into while initialize

synchronized (DoubleCheckLockingSingleton.class) {

// Re-check again. Maybe another thread has initialized before

if (instance == null) {

instance = new DoubleCheckLockingSingleton();

}

}

}

// Do something after get instance ...

return instance;

}

}

### Bill Pugh Singleton Implementation

Với cách làm này bạn sẽ tạo ra static nested class với vai trò 1 Helper khi muốn tách biệt chức năng cho 1 class function rõ ràng hơn. Đây là cách thường hay được sử dụng và có hiệu suất tốt (theo các chuyên gia đánh giá 🙂 ).

package com.gpcoder.patterns.creational.singleton;

public class BillPughSingleton {

private BillPughSingleton() {

}

public static BillPughSingleton getInstance() {

return SingletonHelper.INSTANCE;

}

private static class SingletonHelper {

private static final BillPughSingleton INSTANCE = new BillPughSingleton();

}

}

Khi Singleton được tải vào bộ nhớ thì SingletonHelper chưa được tải vào. Nó chỉ được tải khi và chỉ khi phương thức getInstance() được gọi. Với cách này tránh được lỗi cơ chế khởi tạo instance của Singleton trong Multi-Thread, performance cao do tách biệt được quá trình xử lý. Do đó, cách làm này được đánh giá là cách triển khai Singleton nhanh và hiệu quả nhất.

Bill Pugh Singleton là cách tiếp cận tốt, tuy nhiên, nó dễ dàng bị phá vỡ bởi Reflection.

### Phá vỡ cấu trúc Singleton Pattern bằng Reflection

Reflection có thể được dùng để phá vỡ Pattern của Eager Initialization ở trên. Ví dụ:

package com.gpcoder.patterns.creational.singleton;

import java.lang.reflect.Constructor;

import java.lang.reflect.InvocationTargetException;

public class ReflectionBreakSingleton {

public static void main(String[] args)

throws InstantiationException, IllegalAccessException, InvocationTargetException {

EagerInitializedSingleton instanceOne = EagerInitializedSingleton.getInstance();

EagerInitializedSingleton instanceTwo = null;

Constructor<?>[] constructors = EagerInitializedSingleton.class.getDeclaredConstructors();

for (Constructor<?> constructor : constructors) {

constructor.setAccessible(true);

instanceTwo = (EagerInitializedSingleton) constructor.newInstance();

}

System.out.println(instanceOne.hashCode());

System.out.println(instanceTwo.hashCode());

}

}

Output của chương trình:

2018699554

1311053135

Do đó, khi cần sử dụng Eager Initialization chúng ta nên implement theo Bill Pugh Singleton để không bị break bởi Reflection và cũng đạt được hiệu suất tốt hơn.

### Enum Singleton

Khi dùng enum thì các params chỉ được khởi tạo 1 lần duy nhất, đây cũng là cách giúp bạn tạo ra Singleton instance.

Ví dụ:

package com.gpcoder.patterns.creational.singleton;

/\*\*

\* Singleton implementation using enum initialization

\*/

public enum EnumSingleton {

INSTANCE;

}

Lưu ý:

* Enum có thể sử dụng như một Singleton, nhưng nó có nhược điểm là không thể extends từ một lớp được, nên khi sử dụng cần xem xét vấn đề này.
* Hàm constructor của enum là lazy, nghĩa là khi được sử dụng mới chạy hàm khởi tạo và nó chỉ chạy duy nhất một lần. Nếu muốn sử dụng như một eager singleton thì cần gọi thực thi trong một static block khi start chương trình.

So sánh giữa 2 cách sử dụng enum initialization và static block initialization method, enum có một điểm rất mạnh khi giải quyết về vấn đề Serialization/ Deserialization.

### Serialization and Singleton

Đôi khi trong các hệ thống phân tán (distributed system), chúng ta cần implement interface Serializable trong lớp Singleton để chúng ta có thể lưu trữ trạng thái của nó trong file hệ thống và truy xuất lại nó sau.

Ví dụ:

package com.gpcoder.patterns.creational.singleton;

import java.io.ObjectStreamException;

import java.io.Serializable;

public class SerializedSingleton implements Serializable {

private static final long serialVersionUID = 1741825395699241705L;

private SerializedSingleton() {

}

private static class SingletonHelper {

private static final SerializedSingleton instance = new SerializedSingleton();

}

public static SerializedSingleton getInstance() {

return SingletonHelper.instance;

}

/\*\*

\* Special hook provided by serialization where developer can control what object needs to sent.

\* However this method is invoked on the new object instance created by de serialization process.

\*

\* @return

\* @throws ObjectStreamException

\*/

// private Object readResolve() throws ObjectStreamException {

// return SingletonHelper.instance;

// }

}

Đoạn code test quá trình Serialize/ Deserialize:

package com.gpcoder.patterns.creational.singleton;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInput;

import java.io.ObjectInputStream;

import java.io.ObjectOutput;

import java.io.ObjectOutputStream;

public class SingletonSerializedTest {

public static void main(String[] args) throws FileNotFoundException, IOException, ClassNotFoundException {

SerializedSingleton serializedSingleton1 = SerializedSingleton.getInstance();

EnumSingleton enumSingleton1 = EnumSingleton.INSTANCE;

ObjectOutput out = new ObjectOutputStream(new FileOutputStream("SingletonSerializedTest.txt"));

out.writeObject(serializedSingleton1);

out.writeObject(enumSingleton1);

out.close();

// De-serialize from file to object

ObjectInput in = new ObjectInputStream(new FileInputStream("SingletonSerializedTest.txt"));

SerializedSingleton serializedSingleton2 = (SerializedSingleton) in.readObject();

EnumSingleton enumSingleton2 = (EnumSingleton) in.readObject();

in.close();

System.out.println("serializedSingleton1 hashCode=" + serializedSingleton1.hashCode());

System.out.println("serializedSingleton2 hashCode=" + serializedSingleton2.hashCode());

System.out.println("enumSingleton1 hashCode=" + enumSingleton1.hashCode());

System.out.println("enumSingleton2 hashCode=" + enumSingleton2.hashCode());

}

}

Output của chương trình:

serializedSingleton1 hashCode=1028566121

serializedSingleton2 hashCode=1747585824

enumSingleton1 hashCode=1118140819

enumSingleton2 hashCode=1118140819

Như trong ví dụ trên, Deserialize đối tượng của SerializedSingleton khác với đối tượng gốc. Tuy nhiên vấn đề này không xảy ra khi sử dụng enum.

Thực tế thì vẫn có cách khắc phục khi sử dụng class SerializedSingleton là implement một phương thức readResolve(). Nhưng khi chúng ta thật sự gặp vấn đề và cần sử dụng Serialize/ Deserialize, thì nên sử dụng enum sẽ đơn giản hơn.

## Sử dụng Singleton Pattern khi nào?

Dưới đây là một số trường hợp sử dụng của Singleton Pattern thường gặp:

* Vì class dùng Singleton chỉ tồn tại 1 Instance (thể hiện) nên nó thường được dùng cho các trường hợp giải quyết các bài toán cần truy cập vào các ứng dụng như: Shared resource, Logger, Configuration, Caching, Thread pool, …
* Một số design pattern khác cũng sử dụng Singleton để triển khai: Abstract Factory, Builder, Prototype, Facade,…
* Đã được sử dụng trong một số class của core java như: java.lang.Runtime, java.awt.Desktop.

## Tổng kết

Có rất nhiều cách implement cho Singleton, mình thường sử dụng BillPughSingleton vì có hiệu suất cao, sử dụng LazyInitializedSingleton cho những ứng dụng chỉ làm việc với ứng dụng single-thread và sử dụng DoubleCheckLockingSingleton khi làm việc với ứng dụng multi-thread. Tùy theo trường hợp cụ thể, bạn hãy chọn cho mình cách implement phù hợp.

Tài liệu tham khảo:

* <https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples>
* <https://community.oracle.com/docs/DOC-918906>
* <https://www.geeksforgeeks.org/prevent-singleton-pattern-reflection-serialization-cloning/>
* <http://www.java67.com/2015/09/thread-safe-singleton-in-java-using-double-checked-locking-pattern.html>
* <http://www.javacreed.com/the-broken-singleton/>

# Hướng dẫn Java Design Pattern – Abstract Factory

Đăng vào 15/09/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 4628 Lượt xem

Trong bài trước chúng ta đã tìm hiểu về [Factory Method Pattern](https://gpcoder.com/4352-huong-dan-java-design-pattern-factory-method/). Trong bài này chúng ta tiếp tục tìm hiểu một Pattern khác trong nhóm Creational Design Pattern là Abstract Factory Design Pattern.

Nội dung [[Ẩn](https://gpcoder.com/4365-huong-dan-java-design-pattern-abstract-factory/)]

* [1 Abstract Factory Pattern là gì?](https://gpcoder.com/4365-huong-dan-java-design-pattern-abstract-factory/" \l "Abstract_Factory_Pattern_la_gi)
* [2 Cài đặt Abstract Factory Pattern như thế nào?](https://gpcoder.com/4365-huong-dan-java-design-pattern-abstract-factory/" \l "Cai_dat_Abstract_Factory_Pattern_nhu_the_nao)
* [3 Lợi ích của Abstract Factory Pattern là gì?](https://gpcoder.com/4365-huong-dan-java-design-pattern-abstract-factory/" \l "Loi_ich_cua_Abstract_Factory_Pattern_la_gi)

## Abstract Factory Pattern là gì?

[Abstract Factory](https://en.wikipedia.org/wiki/Abstract_factory_pattern) is a creational design pattern that provide an interface for creating families of  related or dependent objects without specifying their concrete classes.

Abstract Factory pattern là một trong những Creational pattern. Nó là phương pháp tạo ra một Super-factory dùng để tạo ra các Factory khác. Hay còn được gọi là Factory của các Factory. Abstract Factory Pattern là một Pattern cấp cao hơn so với Factory Method Pattern.

Trong Abstract Factory pattern, một interface có nhiệm vụ tạo ra một Factory của các object có liên quan tới nhau mà không cần phải chỉ ra trực tiếp các class của object. Mỗi Factory được tạo ra có thể tạo ra các object bằng phương pháp giống như Factory pattern.

Hãy tưởng tượng, Abstract factory như là một nhà máy lớn chứa nhiều nhà máy nhỏ, trong các nhà máy đó có những xưởng sản xuất, các xưởng đó tạo ra những sản phẩm khác nhau.

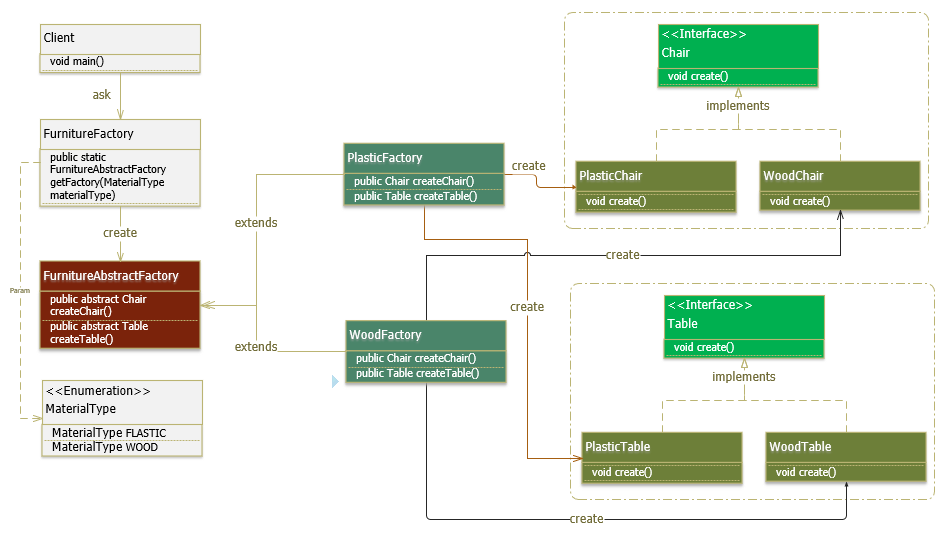
## Cài đặt Abstract Factory Pattern như thế nào?

Một Abstract Factory Pattern bao gồm các thành phần cơ bản sau:

* AbstractFactory: Khai báo dạng interface hoặc abstract class chứa các phương thức để tạo ra các đối tượng abstract.
* ConcreteFactory: Xây dựng, cài đặt các phương thức tạo các đối tượng cụ thể.
* AbstractProduct: Khai báo dạng interface hoặc abstract class để định nghĩa đối tượng abstract.
* Product: Cài đặt của các đối tượng cụ thể, cài đặt các phương thức được quy định tại AbstractProduct.
* Client: là đối tượng sử dụng AbstractFactory và các AbstractProduct.

Ví dụ: Một công ty đồ nội thất chuyên sản xuất ghế (Chair): ghế nhựa (PlasticChair) và ghế gỗ (WoodChair). Với tình hình kinh doanh ngày càng thuận thợi nên công ty quyết định mở rộng thêm sản xuất bàn (Table). Với lợi thế là đã có kinh nghiệm từ sản xuất ghế nên công ty vẫn giữ chất liệu là nhựa (PlasticTable) và gỗ (WoodTable) cho sản xuất bàn. Tuy nhiên, quy trình sản xuất ghế/ bàn theo từng chất liệu (MaterialType) là khác nhau. Nên công ty tách ra là nhà máy (Factory): 1 cho sản xuất vật liệu bằng nhựa (PlasticFactory), 1 cho sản xuất vật liệu bằng gỗ (WoodFactory), nhưng cả 2 đều có thể sản xuất ghế và bàn (FunitureAbstractFactory). Khi khách hàng cần mua một món đồ nào, khách hàng (Client) chỉ cần đến cửa hàng để mua (FunitureFactory). Khi đó ứng với từng hàng hóa và vật liệu sẽ được chuyển về phân xưởng tương ứng để sản xuất (createXXX) ra bàn (Table) và ghế (Chair).

Hệ thống được minh họa như sau:



Chương trình được cài đặt theo Abstract Factory Pattern như sau:

Super Factory Class:

package com.gpcoder.patterns.creational.abstractfactory.factory;

import com.gpcoder.patterns.creational.abstractfactory.MaterialType;

import com.gpcoder.patterns.creational.abstractfactory.factory.impl.FlasticFactory;

import com.gpcoder.patterns.creational.abstractfactory.factory.impl.WoodFactory;

public class FurnitureFactory {

private FurnitureFactory() {

}

// Returns a concrete factory object that is an instance of the

// concrete factory class appropriate for the given architecture.

public static FurnitureAbstractFactory getFactory(MaterialType materialType) {

switch (materialType) {

case FLASTIC:

return new FlasticFactory();

case WOOD:

return new WoodFactory();

default:

throw new UnsupportedOperationException("This furniture is unsupported ");

}

}

}

AbstractFactory:

package com.gpcoder.patterns.creational.abstractfactory.factory;

import com.gpcoder.patterns.creational.abstractfactory.chair.Chair;

import com.gpcoder.patterns.creational.abstractfactory.table.Table;

public abstract class FurnitureAbstractFactory {

public abstract Chair createChair();

public abstract Table createTable();

}

ConcreteFactory:

FlasticFactory:

package com.gpcoder.patterns.creational.abstractfactory.factory.impl;

import com.gpcoder.patterns.creational.abstractfactory.chair.Chair;

import com.gpcoder.patterns.creational.abstractfactory.chair.PlasticChair;

import com.gpcoder.patterns.creational.abstractfactory.factory.FurnitureAbstractFactory;

import com.gpcoder.patterns.creational.abstractfactory.table.PlasticTable;

import com.gpcoder.patterns.creational.abstractfactory.table.Table;

public class FlasticFactory extends FurnitureAbstractFactory {

@Override

public Chair createChair() {

return new PlasticChair();

}

@Override

public Table createTable() {

return new PlasticTable();

}

}

WoodFactory:

package com.gpcoder.patterns.creational.abstractfactory.factory.impl;

import com.gpcoder.patterns.creational.abstractfactory.chair.Chair;

import com.gpcoder.patterns.creational.abstractfactory.chair.WoodChair;

import com.gpcoder.patterns.creational.abstractfactory.factory.FurnitureAbstractFactory;

import com.gpcoder.patterns.creational.abstractfactory.table.Table;

import com.gpcoder.patterns.creational.abstractfactory.table.WoodTable;

public class WoodFactory extends FurnitureAbstractFactory {

@Override

public Chair createChair() {

return new WoodChair();

}

@Override

public Table createTable() {

return new WoodTable();

}

}

AbstractProduct và Product:

Chair:

package com.gpcoder.patterns.creational.abstractfactory.chair;

public interface Chair {

void create();

}

package com.gpcoder.patterns.creational.abstractfactory.chair;

public class PlasticChair implements Chair {

@Override

public void create() {

System.out.println("Create plastic chair");

}

}

package com.gpcoder.patterns.creational.abstractfactory.chair;

public class WoodChair implements Chair {

@Override

public void create() {

System.out.println("Create wood chair");

}

}

Table:

package com.gpcoder.patterns.creational.abstractfactory.table;

public interface Table {

void create();

}

package com.gpcoder.patterns.creational.abstractfactory.table;

public class PlasticTable implements Table {

@Override

public void create() {

System.out.println("Create plastic table");

}

}

package com.gpcoder.patterns.creational.abstractfactory.table;

public class WoodTable implements Table {

@Override

public void create() {

System.out.println("Create wood table");

}

}

Material type:

public enum MaterialType {

FLASTIC, WOOD

}

Client:

package com.gpcoder.patterns.creational.abstractfactory;

import com.gpcoder.patterns.creational.abstractfactory.chair.Chair;

import com.gpcoder.patterns.creational.abstractfactory.factory.FurnitureAbstractFactory;

import com.gpcoder.patterns.creational.abstractfactory.factory.FurnitureFactory;

import com.gpcoder.patterns.creational.abstractfactory.table.Table;

public class Client {

public static void main(String[] args) {

FurnitureAbstractFactory factory = FurnitureFactory.getFactory(MaterialType.FLASTIC);

Chair chair = factory.createChair();

chair.create(); // Create plastic chair

Table table = factory.createTable();

table.create(); // Create plastic table

}

}

Như bạn thấy, phía Client chỉ cần gọi phương thức FurnitureFactory.getFactory() là có thể sử dụng được các dịch vụ của một Factory bất kỳ.

Khi hệ thống phát triển cần mở rộng thêm 1 nhà máy khác, chẳng hạn sản xuất hàng hóa bằng inox, thì đơn giản cần tạo thêm một class mới implement từ FurnitureAbstractFactory, và thêm vào logic khởi tạo Funiture trong FurnitureFactory. Nó không làm ảnh hưởng đến code ở phía Client.

## Lợi ích của Abstract Factory Pattern là gì?

* Các lợi ích của Factory Pattern cũng tương tự như Factory Method Pattern như: cung cấp hướng tiếp cận với Interface thay thì các implement, che giấu sự phức tạp của việc khởi tạo các đối tượng với người dùng (client), độc lập giữa việc khởi tạo đối tượng và hệ thống sử dụng, …
* Giúp tránh được việc sử dụng điều kiện logic bên trong Factory Pattern. Khi một Factory Method lớn (có quá nhiều sử lý if-else hay switch-case), chúng ta nên sử dụng theo mô hình Abstract Factory để dễ quản lý hơn (cách phân chia có thể là gom nhóm các sub-class cùng loại vào một Factory).
* Abstract Factory Pattern là factory của các factory, có thể dễ dạng mở rộng để chứa thêm các factory và các sub-class khác.
* Dễ dàng xây dựng một hệ thống đóng gói (encapsulate): sử dụng được với nhiều nhóm đối tượng (factory) và tạo nhiều product khác nhau.

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/abstract_factory>
* <https://refactoring.guru/design-patterns/abstract-factory>
* <https://www.journaldev.com/1418/abstract-factory-design-pattern-in-java>

# Hướng dẫn Java Design Pattern – Factory Method

Đăng vào 12/09/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 5966 Lượt xem

Trong bài trước chúng ta đã tìm hiểu về [Singleton Design Pattern](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/). Trong bài này chúng ta tiếp tục tìm hiểu một Pattern khác trong nhóm Creational Design Pattern là Factory Method Design Pattern.

Nội dung [[Ẩn](https://gpcoder.com/4352-huong-dan-java-design-pattern-factory-method/)]

* [1 Factory Method Pattern là gì?](https://gpcoder.com/4352-huong-dan-java-design-pattern-factory-method/" \l "Factory_Method_Pattern_la_gi)
* [2 Cài đặt Factory Pattern như thế nào?](https://gpcoder.com/4352-huong-dan-java-design-pattern-factory-method/" \l "Cai_dat_Factory_Pattern_nhu_the_nao)
* [3 Sử dụng Factory Pattern khi nào?](https://gpcoder.com/4352-huong-dan-java-design-pattern-factory-method/" \l "Su_dung_Factory_Pattern_khi_nao)
* [4 Lợi ích của Factory Pattern là gì?](https://gpcoder.com/4352-huong-dan-java-design-pattern-factory-method/" \l "Loi_ich_cua_Factory_Pattern_la_gi)

## Factory Method Pattern là gì?

[Factory Method](https://en.wikipedia.org/wiki/Factory_method_pattern) is a creational design pattern that Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Factory Method Design Pattern hay gọi ngắn gọn là Factory Pattern là một trong những Pattern thuộc nhóm Creational Design Pattern. Nhiệm vụ của Factory Pattern là quản lý và trả về các đối tượng theo yêu cầu, giúp cho việc khởi tạo đổi tượng một cách linh hoạt hơn.

Factory Pattern đúng nghĩa là một nhà máy, và nhà máy này sẽ “sản xuất” các đối tượng theo yêu cầu của chúng ta.

Trong Factory Pattern, chúng ta tạo đối tượng mà không để lộ logic tạo đối tượng ở phía người dùng và tham chiếu đến đối tượng mới được tạo ra bằng cách sử dụng một interface chung.

Factory Pattern được sử dụng khi có một class cha (super-class) với nhiều class con (sub-class), dựa trên đầu vào và phải trả về 1 trong những class con đó.

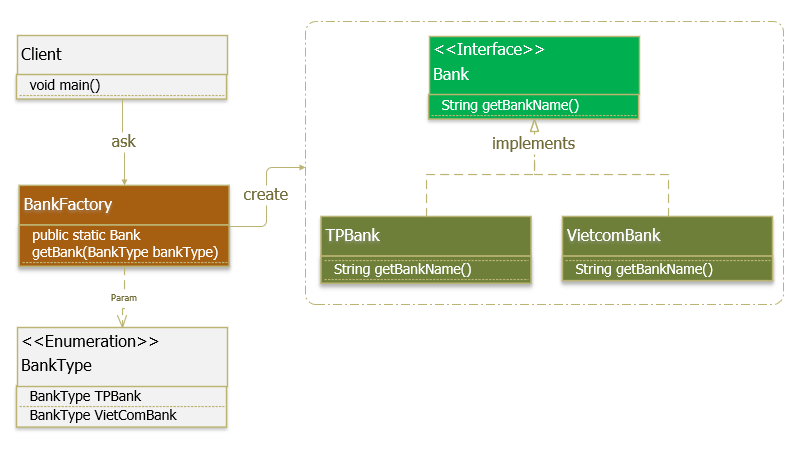
## Cài đặt Factory Pattern như thế nào?

Một Factory Pattern bao gồm các thành phần cơ bản sau:

* Super Class: môt supper class trong Factory Pattern có thể là một interface, abstract class hay một class thông thường.
* Sub Classes: các sub class sẽ implement các phương thức của supper class theo nghiệp vụ riêng của nó.
* Factory Class: một class chịu tránh nhiệm khởi tạo các đối tượng sub class dựa theo tham số đầu vào. Lưu ý: lớp này là [Singleton](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/) hoặc cung cấp một public static method cho việc truy xuất và khởi tạo đối tượng. Factory class sử dụng if-else hoặc switch-case để xác định class con đầu ra.

Ví dụ: Tất cả hệ thống ngân hàng có cung cấp API để truy cập đến hệ thống của họ. Team được giao nhiệm vụ thiết kế một API để client có thể sử dụng dịch vụ của một ngân hàng bất kỳ. Hiện tại, phía client chỉ cần sử dụng dịch vụ của 2 ngân hàng là VietcomBank và TPBank. Tuy nhiên để dễ mở rộng sau này, và phía client mong muốn không cần phải thay đổi code của họ khi cần sử dụng thêm dịch vụ của ngân hàng khác. Với yêu cầu như vậy, chúng ta có thể sử dụng một Pattern phù hợp là Factory Method Pattern.

Hệ thống được minh họa như sau:



Chương trình được cài đặt theo Factory Pattern như sau:

Supper Class:

public interface Bank {

String getBankName();

}

Sub Classes:

package com.gpcoder.patterns.creational.factorymethod;

public class TPBank implements Bank {

@Override

public String getBankName() {

return "TPBank";

}

}

package com.gpcoder.patterns.creational.factorymethod;

public class VietcomBank implements Bank {

@Override

public String getBankName() {

return "VietcomBank";

}

}

Factory class:

public class BankFactory {

private BankFactory() {

}

public static final Bank getBank(BankType bankType) {

switch (bankType) {

case TPBANK:

return new TPBank();

case VIETCOMBANK:

return new VietcomBank();

default:

throw new IllegalArgumentException("This bank type is unsupported");

}

}

}

Bank type:

public enum BankType {

VIETCOMBANK, TPBANK;

}

Client:

public class Client {

public static void main(String[] args) {

Bank bank = BankFactory.getBank(BankType.TPBANK);

System.out.println(bank.getBankName()); // TPBank

}

}

Như bạn thấy, phía client chỉ cần gọi duy nhất một phương thức BankFactory.getBank() là có thể sử dụng được dịch vụ của một ngân hàng bất kỳ.

Khi hệ thống muốn cung cấp thêm dịch vụ của một ngân hàng khác, chẳng hạn VietinBank, thì cần tạo thêm một class mới implement từ interface Bank, và thêm vào logic khởi tạo Bank trong Factory là xong. Nó không làm ảnh hưởng đến code ở phía Client.

## Sử dụng Factory Pattern khi nào?

Factory Pattern được sử dụng khi:

* Chúng ta có một super class với nhiều class con và dựa trên đầu vào, chúng ta cần trả về một class con. Mô hình này giúp chúng ta đưa trách nhiệm của việc khởi tạo một lớp từ phía người dùng (client) sang lớp Factory.
* Chúng ta không biết sau này sẽ cần đến những lớp con nào nữa. Khi cần mở rộng, hãy tạo ra sub class và implement thêm vào factory method cho việc khởi tạo sub class này.

## Lợi ích của Factory Pattern là gì?

Lợi ích của Factory Pattern:

* Factory Pattern giúp giảm sự phụ thuộc giữa các module (loose coupling): cung cấp 1 hướng tiếp cận với Interface thay thì các implement. Giúp chuơng trình độc lập với những lớp cụ thể mà chúng ta cần tạo 1 đối tượng, code ở phía client không bị ảnh hưởng khi thay đổi logic ở factory hay sub class.
* Mở rộng code dễ dàng hơn: khi cần mở rộng, chỉ việc tạo ra sub class và implement thêm vào factory method.
* Khởi tạo các Objects mà che giấu đi xử lí logic của việc khởi tạo đấy. Người dùng không biết logic thực sực được khởi tạo bên dưới phương thức factory.
* Dễ dạng quản lý life cycle của các Object được tạo bởi Factory Pattern.
* Thống nhất về naming convention: giúp cho các developer có thể hiểu về cấu trúc source code.

Bạn có thể thấy Factory Pattern được áp dụng trong:

* JDK: java.util.Calendar, ResourceBundle, NumberFormat, …
* BeanFactory trong Spring Framework.
* SessionFactory trong Hibernate Framework.
* …

Như vậy tôi đã giới thiệu xong về Factory Method Pattern, hy vọng bài viết sẽ giúp ích cho các bạn trong quá trình xây dựng và phát triển ứng dụng. Mọi góp ý và thắc mắc hãy comment bên dưới, tôi sẽ cố gắng trả lời cho các bạn. Hẹn gặp lại ở các bài viết tiếp theo.

Tài liệu tham khảo:

* <https://refactoring.guru/design-patterns/factory-method>
* <https://sourcemaking.com/design_patterns/factory_method>
* <https://www.tutorialspoint.com/design_pattern/factory_pattern.htm>

# Hướng dẫn Java Design Pattern – Builder

Đăng vào 26/09/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 4615 Lượt xem

Các hàm xây dựng (constructor) trong Java được sử dụng để tạo đối tượng và có thể lấy các tham số cần thiết để tạo đối tượng. Vấn đề khi một đối tượng có thể được tạo ra với nhiều tham số (param), một số có thể là bắt buộc và một số khác có thể là tùy chọn tuỳ theo từng yêu cầu của người dùng, tuỳ vào hoàn cảnh của ứng dụng. Chúng ta, có thể tạo ra nhiều constructor theo từng nhu cầu hoặc gán giá trị null cho các param không cần thiết. Tuy nhiên, code rất khó đọc, khó bảo trì, người sử dụng có thể gán nhầm giá trị nếu một loạt các tham số có cùng kiểu. Chúng ta cũng có thể sử dụng một giải pháp khác là sử dụng setter() để thay thế cho constructor. Tuy nhiên, nếu muốn đối tượng này là [immutable](https://gpcoder.com/2149-huong-dan-su-dung-java-string-stringbuffer-va-stringbuilder/" \l "Khai_niem_mutable_immutable) thì không thể.

Do vậy, người ta mong muốn giao công việc này cho một đối tượng chịu trách nhiêm khởi tạo và chia việc khởi tạo đối tượng riêng lẽ, từng bước, để có thể tiến hành khởi tạo riêng biệt ở các hoàn cảnh khác nhau. Và giải pháp được đưa ra là sử dụng Builder Pattern như một người xây dựng.

Nội dung [[Ẩn](https://gpcoder.com/4434-huong-dan-java-design-pattern-builder/)]

* [1 Builder Pattern là gì?](https://gpcoder.com/4434-huong-dan-java-design-pattern-builder/" \l "Builder_Pattern_la_gi)
* [2 Cài đặt Builder Pattern như thế nào?](https://gpcoder.com/4434-huong-dan-java-design-pattern-builder/" \l "Cai_datBuilderPattern_nhu_the_nao)
  + [2.1 Ví dụ sử dụng Builder](https://gpcoder.com/4434-huong-dan-java-design-pattern-builder/" \l "Vi_du_su_dung_Builder)
  + [2.2 Ví dụ sử dụng Builder để tạo đối tượng Immutable](https://gpcoder.com/4434-huong-dan-java-design-pattern-builder/" \l "Vi_du_su_dung_Builder_de_tao_doi_tuong_Immutable)
* [3 Lợi ích của Builder Pattern là gì?](https://gpcoder.com/4434-huong-dan-java-design-pattern-builder/" \l "Loi_ich_cua_Builder_Pattern_la_gi)
* [4 Nhược điểm của Builder Pattern là gì?](https://gpcoder.com/4434-huong-dan-java-design-pattern-builder/" \l "Nhuoc_diem_cua_Builder_Pattern_la_gi)
* [5 Sử dụng Builder Pattern khi nào?](https://gpcoder.com/4434-huong-dan-java-design-pattern-builder/" \l "Su_dung_Builder_Pattern_khi_nao)
* [6 So sánh Builder Pattern với Factory/ Abstract Factory Pattern](https://gpcoder.com/4434-huong-dan-java-design-pattern-builder/" \l "So_sanh_Builder_Pattern_voi_Factory_Abstract_Factory_Pattern)

## Builder Pattern là gì?

[Builder](https://en.wikipedia.org/wiki/Builder_pattern) is a creational design pattern that separate the construction of a complex object from its representation so that the same construction process can create different representations.

Builder pattern là một trong những Creational pattern. Builder pattern là mẫu thiết kế đối tượng được tạo ra để xây dựng một đôi tượng phức tạp bằng cách sử dụng các đối tượng đơn giản và sử dụng tiếp cận từng bước, việc xây dựng các đối tượng đôc lập với các đối tượng khác.

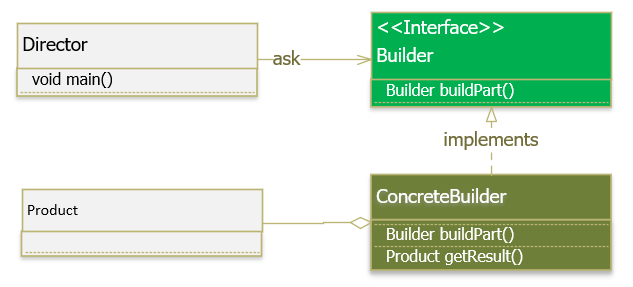
Builder Pattern được xây dựng để khắc phục một số nhược điểm của Factory Pattern và Abstract Factory Pattern khi mà Object có nhiều thuộc tính.

Có ba vấn đề chính với  Factory Pattern và Abstract Factory Pattern khi Object có nhiều thuộc tính:

* Quá nhiều tham số phải truyền vào từ phía client tới Factory Class.
* Một số tham số có thể là tùy chọn nhưng trong Factory Pattern, chúng ta phải gửi tất cả tham số, với tham số tùy chọn nếu không nhập gì thì sẽ truyền là null.
* Nếu một Object có quá nhiều thuộc tính thì việc tạo sẽ phức tạp.

Chúng ta có thể xử lý những vấn đề này với một số lượng lớn các tham số bằng việc cung cấp một hàm khởi tạo với những tham số bắt buộc và các method getter/ setter để cài đặt các tham số tùy chọn. Vấn đề với hướng tiếp cận này là trạng thái của Object sẽ không nhất quán cho tới khi tất cả các thuộc tính được cài đặt một cách rõ ràng. Nếu cần xây dựng một đối tượng Immutable thì cách này cũng không thể thực hiện được.

## Cài đặt Builder Pattern như thế nào?



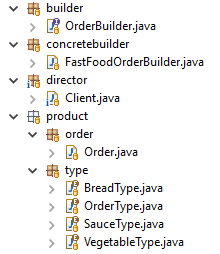
Một builder gồm các thành phần cơ bản sau:

* Product : đại diện cho đối tượng cần tạo, đối tượng này phức tạp, có nhiều thuộc tính.
* Builder : là abstract class hoặc interface khai báo phương thức tạo đối tượng.
* ConcreteBuilder : kế thừa Builder và cài đặt chi tiết cách tạo ra đối tượng. Nó sẽ xác định và nắm giữ các thể hiện mà nó tạo ra, đồng thời nó cũng cung cấp phương thức để trả các các thể hiện mà nó đã tạo ra trước đó.
* Director/ Client: là nơi sẽ gọi tới Builder để tạo ra đối tượng.

Trường hợp đơn giản, chúng ta có thể gộp Builder và ConcreteBuilder thành static nested class bên trong Product.

### Ví dụ sử dụng Builder

Ví dụ: sử dụng Builder cho việc gọi món tại một cửa hàng thức ăn nhanh.



Product:

public class Order {

private OrderType orderType;

private BreadType breadType;

private SauceType sauceType;

private VegetableType vegetableType;

public Order(OrderType orderType, BreadType breadType, SauceType sauceType, VegetableType vegetableType) {

super();

this.orderType = orderType;

this.breadType = breadType;

this.sauceType = sauceType;

this.vegetableType = vegetableType;

}

@Override

public String toString() {

return "Order [orderType=" + orderType + ", breadType=" + breadType + ", sauceType=" + sauceType

+ ", vegetableType=" + vegetableType + "]";

}

public OrderType getOrderType() {

return orderType;

}

public BreadType getBreadType() {

return breadType;

}

public SauceType getSauceType() {

return sauceType;

}

public VegetableType getVegetableType() {

return vegetableType;

}

}

public enum BreadType {

SIMPLE, OMELETTE, FRIED\_EGG, GRILLED\_FISH, PORK, BEEF,

}

public enum OrderType {

ON\_SITE, TAKE\_AWAY;

}

public enum SauceType {

SOY\_SAUCE, FISH\_SAUCE, OLIVE\_OIL, KETCHUP, MUSTARD;

}

public enum VegetableType {

SALAD, CUCUMBER, TOMATO

}

Builder:

package com.gpcoder.patterns.creational.builder.food.builder;

import com.gpcoder.patterns.creational.builder.food.product.order.Order;

import com.gpcoder.patterns.creational.builder.food.product.type.BreadType;

import com.gpcoder.patterns.creational.builder.food.product.type.OrderType;

import com.gpcoder.patterns.creational.builder.food.product.type.SauceType;

import com.gpcoder.patterns.creational.builder.food.product.type.VegetableType;

public interface OrderBuilder {

OrderBuilder orderType(OrderType orderType);

OrderBuilder orderBread(BreadType breadType);

OrderBuilder orderSauce(SauceType sauceType);

OrderBuilder orderVegetable(VegetableType vegetableType);

Order build();

}

ConcreteBuilder:

package com.gpcoder.patterns.creational.builder.food.concretebuilder;

import com.gpcoder.patterns.creational.builder.food.builder.OrderBuilder;

import com.gpcoder.patterns.creational.builder.food.product.order.Order;

import com.gpcoder.patterns.creational.builder.food.product.type.BreadType;

import com.gpcoder.patterns.creational.builder.food.product.type.OrderType;

import com.gpcoder.patterns.creational.builder.food.product.type.SauceType;

import com.gpcoder.patterns.creational.builder.food.product.type.VegetableType;

public class FastFoodOrderBuilder implements OrderBuilder {

private OrderType orderType;

private BreadType breadType;

private SauceType sauceType;

private VegetableType vegetableType;

@Override

public OrderBuilder orderType(OrderType orderType) {

this.orderType = orderType;

return this;

}

@Override

public OrderBuilder orderBread(BreadType breadType) {

this.breadType = breadType;

return this;

}

@Override

public OrderBuilder orderSauce(SauceType sauceType) {

this.sauceType = sauceType;

return this;

}

@Override

public OrderBuilder orderVegetable(VegetableType vegetableType) {

this.vegetableType = vegetableType;

return this;

}

@Override

public Order build() {

return new Order(orderType, breadType, sauceType, vegetableType);

}

}

Director:

package com.gpcoder.patterns.creational.builder.food.director;

import com.gpcoder.patterns.creational.builder.food.concretebuilder.FastFoodOrderBuilder;

import com.gpcoder.patterns.creational.builder.food.product.order.Order;

import com.gpcoder.patterns.creational.builder.food.product.type.BreadType;

import com.gpcoder.patterns.creational.builder.food.product.type.OrderType;

import com.gpcoder.patterns.creational.builder.food.product.type.SauceType;

public class Client {

public static void main(String[] args) {

Order order = new FastFoodOrderBuilder()

.orderType(OrderType.ON\_SITE).orderBread(BreadType.OMELETTE)

.orderSauce(SauceType.SOY\_SAUCE).build();

System.out.println(order);

}

}

Output của chương trình trên:

Order [orderType=ON\_SITE, breadType=OMELETTE, sauceType=SOY\_SAUCE, vegetableType=null]

### Ví dụ sử dụng Builder để tạo đối tượng Immutable

Một vài điểm quan trọng về implement class Product:

* Constructor là private, điều này có nghĩa là class này không thể gọi khởi tạo trực tiếp từ bên ngoài.
* Tất cả các thuộc tính đều là private final, vì vậy nó chỉ được gán giá trị trong constructor và nó chỉ có thể được cung cấp các phương thức getter().
* Việc khởi tạo đối tượng chỉ có thể thông qua Builder.

Một vài điểm quan trọng về implement class Builder:

* Tạo một static nested class (đây được gọi là builder class) và copy tất cả các tham số từ class bên ngoài vào. Chúng ta nên đặt tên class này theo định dạng: [tên class] + Builder. Ví dụ class là BankAccount thì builder class sẽ là BankAccountBuilder.
* Class Builder có một hàm khởi tạo public với tất cả các thuộc tính bắt buộc.
* Class Builder có các method setter() cho các tham số tùy chọn.
* Cung cấp method build() trong Class Builder để trả về đối tượng mà client cần.

Ví dụ: Một tài khoản ngân hàng bao gồm các thông tin: Tên chủ tài khoản, số tài khoản, địa chỉ email, nhận thông báo, sử dụng mobile banking. Một tài khoản được tạo phải có tên chủ tài khoản và số tài khoản. Các thông tin khác tùy theo nhu cầu của khách hàng có thể đăng ký sử dụng.

BankAccount:

package com.gpcoder.patterns.creational.builder.bank;

public class BankAccount {

private final String name; // required

private final String accountNumber; // required

private final String address;

private final String email;

private final boolean newsletter;

private final boolean mobileBanking;

public BankAccount(String name, String accountNumber, String address, String email, boolean newsletter,

boolean mobileBanking) {

super();

this.name = name;

this.accountNumber = accountNumber;

this.address = address;

this.email = email;

this.newsletter = newsletter;

this.mobileBanking = mobileBanking;

}

// Builder class

public static class BankAccountBuilder {

private String name; // required

private String accountNumber; // required

private String address;

private String email;

private boolean newsletter;

private boolean mobileBanking;

public BankAccountBuilder(String name, String accountNumber) {

this.name = name;

this.accountNumber = accountNumber;

}

public BankAccountBuilder withAddress(String address) {

this.address = address;

return this;

}

public BankAccountBuilder withEmail(String email) {

this.email = email;

return this;

}

public BankAccountBuilder wantNewsletter(boolean newsletter) {

this.newsletter = newsletter;

return this;

}

public BankAccountBuilder wantMobileBanking(boolean mobileBanking) {

this.mobileBanking = mobileBanking;

return this;

}

public BankAccount build() {

validateUserObject();

BankAccount bankAccount = new BankAccount(

this.name, this.accountNumber,

this.address, this.email,

this.newsletter, this.mobileBanking);

return bankAccount;

}

private void validateUserObject() {

// Do some basic validations to check

if (this.newsletter && email == null) {

throw new IllegalArgumentException("Email can't be null when client want to receive the new letter");

}

}

}

@Override

public String toString() {

return "BankAccount [name=" + name + ", accountNumber=" + accountNumber + ", address=" + address + ", email="

+ email + ", newsletter=" + newsletter + ", mobileBanking=" + mobileBanking + "]";

}

}

Client:

package com.gpcoder.patterns.creational.builder.bank;

public class Client {

public static void main(String[] args) {

BankAccount newAccount = new BankAccount

.BankAccountBuilder("GP Coder", "0123456789")

.withEmail("contact@gpcoder.com")

.wantNewsletter(true)

.build();

System.out.println(newAccount);

}

}

Output của chương trình:

BankAccount [name=GP Coder, accountNumber=0123456789, address=null, email=contact@gpcoder.com, newsletter=true, mobileBanking=false]

Lưu ý: class BankAccount có hàm khởi tạo là private, nên chỉ có một cách duy nhất để lấy một đối tượng BankAccount là thông qua class BankAccountBuilder.

## Lợi ích của Builder Pattern là gì?

* Hỗ trợ, loại bớt việc phải viết nhiều constructor.
* Code dễ đọc, dễ bảo trì hơn khi số lượng thuộc tính (propery) bắt buộc để tạo một object từ 4 hoặc 5 propery.
* Giảm bớt số lượng constructor, không cần truyền giá trị null cho các tham số không sử dụng.
* Ít bị lỗi do việc gán sai tham số khi mà có nhiều tham số trong constructor: bởi vì người dùng đã biết được chính xác giá trị gì khi gọi phương thức tương ứng.
* Đối tượng được xây dựng an toàn hơn: bởi vì nó đã được tạo hoàn chỉnh trước khi sử dụng.
* Cung cấp cho bạn kiểm soát tốt hơn quá trình xây dựng: chúng ta có thể thêm xử lý kiểm tra ràng buộc trước khi đối tượng được trả về người dùng.
* Có thể tạo đối tượng immutable.

Một số ví dụ sử dụng Builder Pattern trong JDK:

* java.lang.[StringBuilder](https://gpcoder.com/2149-huong-dan-su-dung-java-string-stringbuffer-va-stringbuilder).append().
* java.lang.[StringBuffer](https://gpcoder.com/2149-huong-dan-su-dung-java-string-stringbuffer-va-stringbuilder).append().

## Nhược điểm của Builder Pattern là gì?

Builder Pattern có nhược điểm là duplicate code khá nhiều: do cần phải copy tất cả các thuộc tính từ class Product sang class Builder.

Tăng độ phức tạp của code (tổng thể) do số lượng class tăng lên.

## Sử dụng Builder Pattern khi nào?

* Tạo một đối tượng phức tạp: có nhiều thuộc tính (nhiều hơn 4) và một số bắt buộc (requried), một số không bắt buộc (optional).
* Khi có quá nhiều hàm constructor, bạn nên nghĩ đến Builder.
* Muốn tách rời quá trình xây dựng một đối tượng phức tạp từ các phần tạo nên đối tượng.
* Muốn kiểm soát quá trình xây dựng.
* Khi người dùng (client) mong đợi nhiều cách khác nhau cho đối tượng được xây dựng.

## So sánh Builder Pattern với Factory/ Abstract Factory Pattern

Factory Pattern cũng có thể được sử dụng để xây dựng một đối tượng phức tạp, vậy sự khác biệt của nó với mô hình Builder Pattern là gì?

Sự khác biệt lớn duy nhất giữa Builder Pattern và Factory Pattern cung cấp cho bạn nhiều quyền kiểm soát hơn đối với quá trình tạo đối tượng.

Factory/ Abstract Factory Pattern là câu trả lời cho “WHAT” và Builder Pattern là câu trả lời cho “HOW“.

Trong Builder Pattern, đối tượng được xây dựng từng bước (step by step). Builder Pattern có nhiều bước nhỏ, mỗi bước sẽ có các đơn vị logic nhỏ kèm theo trong đó. Cũng sẽ có một chuỗi (sequence) liên quan. Nó sẽ bắt đầu từ bước 1 và sẽ đi lên tối đa bước n và bước cuối cùng là trả về đối tượng. Nhưng trong Factory Pattern, bạn sẽ không thấy được đối tượng phức tạp được tạo như thế nào, nó không có từng bước xây dựng đối tượng.

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/builder>
* <https://refactoring.guru/design-patterns/builder>
* <https://www.journaldev.com/1425/builder-design-pattern-in-java>

# Hướng dẫn Java Design Pattern – Prototype

Đăng vào 22/09/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 3074 Lượt xem

Trong Java, khi tạo một đối tượng mới thì các thuộc tính của đối tượng chưa được khởi tạo, ngoại trừ giá trị default hoặc được khởi tạo trong constructor. Tuy nhiên, thay vì tạo một đối tượng mới thì ta muốn sao chép một đối tượng khác đã tồn tại. Chúng ta có thể làm được đều này nếu các đối tượng được cài implement theo Prototype Pattern. Prototype được hiểu là khuôn mẫu, tức là đối tượng mới sẽ có bộ khung giống với đối tượng được sao chép. Chi tiết về Prototype pattern như thế nào chúng ta sẽ được giới thiệu trong phần tiếp theo của bài viết này.

Nội dung [[Ẩn](https://gpcoder.com/4413-huong-dan-java-design-pattern-prototype/)]

* [1 Prototype Pattern là gì?](https://gpcoder.com/4413-huong-dan-java-design-pattern-prototype/" \l "Prototype_Pattern_la_gi)
* [2 Cài đặt Prototype Pattern như thế nào?](https://gpcoder.com/4413-huong-dan-java-design-pattern-prototype/" \l "Cai_dat_Prototype_Pattern_nhu_the_nao)
  + [2.1 Ví dụ 1](https://gpcoder.com/4413-huong-dan-java-design-pattern-prototype/" \l "Vi_du_1)
  + [2.2 Ví dụ 2](https://gpcoder.com/4413-huong-dan-java-design-pattern-prototype/" \l "Vi_du_2)
* [3 Lợi ích của Prototype Pattern là gì?](https://gpcoder.com/4413-huong-dan-java-design-pattern-prototype/" \l "Loi_ich_cua_Prototype_Pattern_la_gi)
* [4 Sử dụng Prototype khi nào?](https://gpcoder.com/4413-huong-dan-java-design-pattern-prototype/" \l "Su_dung_Prototype_khi_nao)

## Prototype Pattern là gì?

[Prototype](https://en.wikipedia.org/wiki/Prototype_pattern) is a creational design pattern that specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

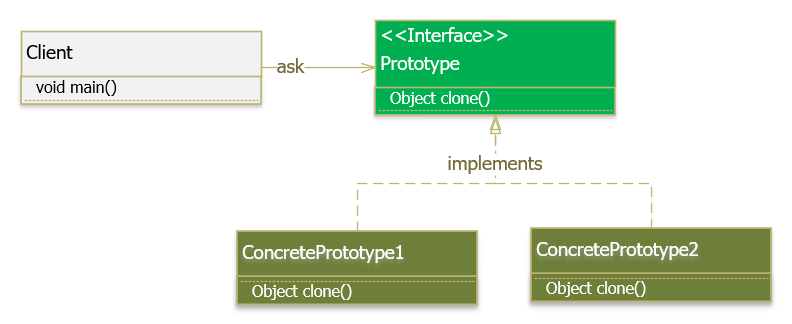
Prototype lets you produce new objects by copying existing ones without compromising their internals. The new object is an exact copy of the prototype but permits modification without altering the original.

Prototype pattern là một trong những Creational pattern. Nó có nhiệm vụ khởi tạo một đối tượng bằng cách clone một đối tượng đã tồn tại thay vì khởi tạo với từ khoá new. Đối tượng mới là một bản sao có thể giống 100% với đối tượng gốc, chúng ta có thể thay đổi dữ liệu của nó mà không ảnh hưởng đến đối tượng gốc.

Prototype Pattern được dùng khi việc tạo một object tốn nhiều chi phí và thời gian trong khi bạn đã có một object tương tự tồn tại.

Trong Java cung cấp mẫu prototype pattern này bằng việc implement interface [Cloneable](https://gpcoder.com/2361-object-cloning-trong-java/) và sử dụng method clone() để tạo object có đầy đủ thuộc tính của đối tượng ban đầu. Các bạn có thể xem lại bài viết về [Object cloning trong java](https://gpcoder.com/2361-object-cloning-trong-java/) để hiểu thêm về interface Clonable trước khi tiếp tục xem phần tiếp theo của bài viết này.

## Cài đặt Prototype Pattern như thế nào?



Một Prototype Pattern gồm các thành phần cơ bản sau:

* Prototype : khai báo một class, interface hoặc abtract class cho việc clone chính nó.
* ConcretePrototype class : các lớp này thực thi interface (hoặc kế thừa từ lớp abstract) được cung cấp bởi Prototype để copy (nhân bản) chính bản thân nó. Các lớp này chính là thể hiện cụ thể phương thức clone(). Lớp này có thể không cần thiết nếu: Prototype là một class và nó đã implement việc clone chính nó.
* Client class : tạo mới object bằng cách gọi Prototype thực hiện clone chính nó.

### Ví dụ 1

Ví dụ: Một công ty có cấu hình máy tính đều giống nhau cho tất cả nhân viên, bao gồm: Hệ điều hành (os), Phần mềm văn phòng (office), Phần mềm diệt virus (antivirus), Trình duyệt (Browser), và một số phần mềm khác (others) tùy theo nhu cầu của mỗi nhân viên sẽ được cài đặt thêm. Việc cài đặt tất cả phần mềm trên rất tốn thời gian, nên anh IT đã nghĩ ra một cách là sẽ tạo ra một bản chuẩn cho một máy tính và có thể clone() lại cấu hình đó cho một nhân viên khác mà không cần phải cài đặt lại từ đầu.

Cách làm này được minh họa như sau:

* Đầu tiên, chúng ta sẽ tạo một lớp Computer và implement từ interface Cloneable.
* Tiếp theo, chúng ta sẽ tạo một computer chuẩn (instance).
* Sau đó, mỗi khi có nhân viên mới, chỉ việc clone() từ computer chuẩn đã tạo. Tùy theo, nhu cầu của mỗi nhân viên có thể thay đổi lại cho phù hợp (setOthers()).

Computer:

package com.gpcoder.patterns.creational.prototype.computer;

public class Computer implements Cloneable {

private String os;

private String office;

private String antivirus;

private String browser;

private String others;

public Computer(String os, String office, String antivirus, String browser, String other) {

super();

this.os = os;

this.office = office;

this.antivirus = antivirus;

this.browser = browser;

this.others = other;

}

@Override

protected Computer clone() {

try {

return (Computer) super.clone();

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

return null;

}

@Override

public String toString() {

return "Computer [os=" + os + ", office=" + office + ", antivirus=" + antivirus + ", browser=" + browser

+ ", others=" + others + "]";

}

public void setOthers(String others) {

this.others = others;

}

}

App:

package com.gpcoder.patterns.creational.prototype.computer;

public class ITApp {

public static void main(String[] args) {

Computer computer1 = new Computer("Window 10", "Word 2013", "BKAV", "Chrome v69", "Skype");

Computer computer2 = computer1.clone();

computer2.setOthers("Skype, Teamviewer, FileZilla Client");

System.out.println(computer1);

System.out.println(computer2);

}

}

Output của chương trình:

Computer [os=Window 10, office=Word 2013, antivirus=BKAV, browser=Chrome v69, other=Skype]

Computer [os=Window 10, office=Word 2013, antivirus=BKAV, browser=Chrome v69, other=Skype, Teamviewer, FileZilla Client]

Như bạn thấy, computer có đầy đủ các giá trị được clone từ computer1 và chúng ta có thể thay đổi giá trị của computer2 mà không ảnh hưởng đến computer1. Code chúng ta đơn giản, dễ sử dụng và nhanh hơn nhiều so với việc gán từng giá trị cho mỗi thuộc tính của đối tượng được tạo mới.

### Ví dụ 2

Một ví dụ khác về Prototype Pattern hơi phức tạp hơn để các bạn có thể thấy rõ được lợi ích của Prototype và việc kết hợp Prototype như thế nào trong [Factory Method Pattern](https://gpcoder.com/4352-huong-dan-java-design-pattern-factory-method/).



Ví dụ: Một bàn cờ vua gồm có: 8 hàng (row) và 8 cột (column). Tương ứng với hàng và cột là các ô (cell) được tô màu đen (black) và trắng (white). Chúng ta, hãy xem các cách implement để vẽ bàn cờ vua như sau:

Cell:

package com.gpcoder.patterns.creational.prototype.board1;

import java.util.concurrent.TimeUnit;

public class Cell {

private String color;

private String coordinate;

public Cell(String color) {

this.color = color;

try {

// More time to create an cell

TimeUnit.MILLISECONDS.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

@Override

public String toString() {

return "Cell [color=" + color + "]";

}

}

Color:

public enum Color {

BLACK, WHITE;

}

Board:

package com.gpcoder.patterns.creational.prototype.board1;

public class Board {

public static final int NO\_OF\_ROWS = 8;

public static final int NO\_OF\_COLUMNS = 8;

private final Cell[][] board;

public Board() {

this.board = new Cell[NO\_OF\_ROWS][NO\_OF\_COLUMNS];

Cell cell;

for (int row = NO\_OF\_ROWS - 1; row >= 0; row--) {

for (int col = NO\_OF\_COLUMNS - 1; col >= 0; col--) {

if ((row + col) % 2 == 0) {

cell = new Cell(Color.WHITE.name());

} else {

cell = new Cell(Color.BLACK.name());

}

board[row][col] = cell;

}

}

}

public void print() {

for (int row = 0; row < NO\_OF\_ROWS; row++) {

for (int col = 0; col < NO\_OF\_COLUMNS; col++) {

System.out.print(board[row][col] + " ");

}

System.out.println();

}

}

}

Chương trình chính: PrototypeExample.java

package com.gpcoder.patterns.creational.prototype.board3;

public class PrototypeExample {

public static void main(String[] args) {

long startTime = System.currentTimeMillis();

Board chessBoard = new Board();

long endTime = System.currentTimeMillis();

System.out.println("Time taken to create a board: " + (endTime - startTime) + " millis");

chessBoard.print();

}

}

Output của chương trình:

Time taken to create a board: 680 millis

Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK]

Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE]

Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK]

Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE]

Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK]

Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE]

Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK]

Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE]

Như bạn thấy, thời gian thực hiện rất lâu (680ms). Nhìn vào chương trình trên, bạn sẽ dễ dàng thấy được là chương trình thực hiện lâu do việc tạo Cell, và mỗi cell chỉ có 2 loại là Cell màu trắng và Cell màu đen. Chương trình trên được sửa lại như sau:

Chúng ta sẽ sử dụng một lớp Factory Method Pattern để quản lý việc khởi tạo đối tượng. Trong lớp này, chúng ta sẽ tạo một Map để lưu giữ các đối tượng đã khởi tạo theo loại tương ứng.

CellFactory:

package com.gpcoder.patterns.creational.prototype.board2;

import java.util.HashMap;

import java.util.Map;

import com.gpcoder.patterns.creational.prototype.board1.Color;

public class CellFactory {

private static final Map<Color, Cell> CELL\_CACHE = new HashMap<>();

private CellFactory() {

}

public static Cell getCell(Color color) {

if (!CELL\_CACHE.containsKey(color)) {

Cell cell = new Cell(color.name());

CELL\_CACHE.put(color, cell);

}

return CELL\_CACHE.get(color);

}

}

Hàm khởi tạo Board được viết lại như sau:

public Board() {

this.board = new Cell[NO\_OF\_ROWS][NO\_OF\_COLUMNS];

Cell cell;

for (int row = NO\_OF\_ROWS - 1; row >= 0; row--) {

for (int col = NO\_OF\_COLUMNS - 1; col >= 0; col--) {

if ((row + col) % 2 == 0) {

cell = CellFactory.getCell(Color.WHITE);

} else {

cell = CellFactory.getCell(Color.BLACK);

}

board[row][col] = cell;

}

}

}

Thực thi chương trình trên lại, chúng ta có kết quả như sau:

Time taken to create a board: 26 millis

Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK]

Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE]

Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK]

Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE]

Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK]

Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE]

Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK]

Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE] Cell [color=BLACK] Cell [color=WHITE]

Như bạn thấy, bây giờ chương trình của chúng ta nhanh lên rất nhiều (26ms) nhờ vào việc implement một đối tượng để cache việc khởi tạo Cell.

Chương trình trên khá tốt, tuy nhiên, nếu bây giờ lớp Cell cần quản lý thêm coordinate và mỗi cell khác nhau sẽ có giá trị khác nhau. Nếu sử dụng với như trên thì khi thay đổi giá trị của một coordinate thì đối tượng gốc cũng bị thay đổi theo. Để giải quyết vấn đề này chúng ta sẽ implement thêm một interface Clonaeble cho class Cell.

Chương trình trên được viết lại như sau:

Cell:

package com.gpcoder.patterns.creational.prototype.board3;

import java.util.concurrent.TimeUnit;

public class Cell implements Cloneable {

private String color;

private String coordinate;

public Cell(String color) {

// Do more time to create an cell

this.color = color;

try {

TimeUnit.MILLISECONDS.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

public void setCoordinate(String coordinate) {

this.coordinate = coordinate;

}

@Override

public String toString() {

return "Cell [color=" + color + ", coordinate=" + coordinate + "]";

}

@Override

protected Cell clone() {

try {

return (Cell) super.clone();

} catch (CloneNotSupportedException e) {

e.printStackTrace();

}

return null;

}

}

CellFactory:

package com.gpcoder.patterns.creational.prototype.board3;

import java.util.HashMap;

import java.util.Map;

import com.gpcoder.patterns.creational.prototype.board1.Color;

public class CellFactory {

private static final Map<Color, Cell> CELL\_CACHE = new HashMap<>();

private CellFactory() {

}

public static Cell getCell(Color color) {

if (!CELL\_CACHE.containsKey(color)) {

Cell cell = new Cell(color.name());

CELL\_CACHE.put(color, cell);

}

return CELL\_CACHE.get(color).clone();

}

}

Board:

package com.gpcoder.patterns.creational.prototype.board3;

import com.gpcoder.patterns.creational.prototype.board1.Color;

public class Board {

public static final int NO\_OF\_ROWS = 8;

public static final int NO\_OF\_COLUMNS = 8;

private final Cell[][] board;

public Board() {

this.board = new Cell[NO\_OF\_ROWS][NO\_OF\_COLUMNS];

Cell cell;

for (int row = NO\_OF\_ROWS - 1; row >= 0; row--) {

for (int col = NO\_OF\_COLUMNS - 1; col >= 0; col--) {

if ((row + col) % 2 == 0) {

cell = CellFactory.getCell(Color.WHITE);

} else {

cell = CellFactory.getCell(Color.BLACK);

}

cell.setCoordinate(String.format("%dx%d", row, col));

board[row][col] = cell;

}

}

}

public void print() {

for (int row = 0; row < NO\_OF\_ROWS; row++) {

for (int col = 0; col < NO\_OF\_COLUMNS; col++) {

System.out.print(board[row][col] + " ");

}

System.out.println();

}

}

}

Thực thi chương trình trên lại, chúng ta có kết quả sau:

Time taken to create a board: 34 millis

Cell [color=WHITE, coordinate=0x0] Cell [color=BLACK, coordinate=0x1] Cell [color=WHITE, coordinate=0x2] Cell [color=BLACK, coordinate=0x3] Cell [color=WHITE, coordinate=0x4] Cell [color=BLACK, coordinate=0x5] Cell [color=WHITE, coordinate=0x6] Cell [color=BLACK, coordinate=0x7]

Cell [color=BLACK, coordinate=1x0] Cell [color=WHITE, coordinate=1x1] Cell [color=BLACK, coordinate=1x2] Cell [color=WHITE, coordinate=1x3] Cell [color=BLACK, coordinate=1x4] Cell [color=WHITE, coordinate=1x5] Cell [color=BLACK, coordinate=1x6] Cell [color=WHITE, coordinate=1x7]

Cell [color=WHITE, coordinate=2x0] Cell [color=BLACK, coordinate=2x1] Cell [color=WHITE, coordinate=2x2] Cell [color=BLACK, coordinate=2x3] Cell [color=WHITE, coordinate=2x4] Cell [color=BLACK, coordinate=2x5] Cell [color=WHITE, coordinate=2x6] Cell [color=BLACK, coordinate=2x7]

Cell [color=BLACK, coordinate=3x0] Cell [color=WHITE, coordinate=3x1] Cell [color=BLACK, coordinate=3x2] Cell [color=WHITE, coordinate=3x3] Cell [color=BLACK, coordinate=3x4] Cell [color=WHITE, coordinate=3x5] Cell [color=BLACK, coordinate=3x6] Cell [color=WHITE, coordinate=3x7]

Cell [color=WHITE, coordinate=4x0] Cell [color=BLACK, coordinate=4x1] Cell [color=WHITE, coordinate=4x2] Cell [color=BLACK, coordinate=4x3] Cell [color=WHITE, coordinate=4x4] Cell [color=BLACK, coordinate=4x5] Cell [color=WHITE, coordinate=4x6] Cell [color=BLACK, coordinate=4x7]

Cell [color=BLACK, coordinate=5x0] Cell [color=WHITE, coordinate=5x1] Cell [color=BLACK, coordinate=5x2] Cell [color=WHITE, coordinate=5x3] Cell [color=BLACK, coordinate=5x4] Cell [color=WHITE, coordinate=5x5] Cell [color=BLACK, coordinate=5x6] Cell [color=WHITE, coordinate=5x7]

Cell [color=WHITE, coordinate=6x0] Cell [color=BLACK, coordinate=6x1] Cell [color=WHITE, coordinate=6x2] Cell [color=BLACK, coordinate=6x3] Cell [color=WHITE, coordinate=6x4] Cell [color=BLACK, coordinate=6x5] Cell [color=WHITE, coordinate=6x6] Cell [color=BLACK, coordinate=6x7]

Cell [color=BLACK, coordinate=7x0] Cell [color=WHITE, coordinate=7x1] Cell [color=BLACK, coordinate=7x2] Cell [color=WHITE, coordinate=7x3] Cell [color=BLACK, coordinate=7x4] Cell [color=WHITE, coordinate=7x5] Cell [color=BLACK, coordinate=7x6] Cell [color=WHITE, coordinate=7x7]

Như bạn thấy, sau khi áp dụng Prototype thì chúng ta có thể thay đổi giá trị mà không ảnh hưởng đến đối tượng gốc.

## Lợi ích của Prototype Pattern là gì?

* Cãi thiện performance: giảm chi phí để tạo ra một đối tượng mới theo chuẩn, điều này sẽ làm tăng hiệu suất so với việc sử dụng từ khóa new để tạo đối tượng mới.
* Giảm độ phức tạp cho việc khởi tạo đối tượng: do mỗi lớp chỉ implement cách clone của chính nó.
* Giảm việc phân lớp, tránh việc tạo nhiều lớp con cho việc khởi tạo đối tượng như của [Abstract Factory Pattern](https://gpcoder.com/4365-huong-dan-java-design-pattern-abstract-factory/).
* Khởi tạo object mới bằng cách thay đổi một vài thuộc tính của object (các object có ít điểm khác biệt nhau): Một hệ thống linh động sẽ để cho chúng ta tự định nghĩa một hành động nào đó thông qua sự kết hợp với một object (nghĩa là một phương thức của một class) hơn là định nghĩa một class mới.
* Khởi tạo object mới bằng cách thay đổi cấu trúc: Rất nhiều ứng dụng xây dựng hệ thống từ nhiều phần và các phần con. Các phần con lại khởi tạo từ nhiều phần con khác (chia nhỏ bài toán). Prototype pattern cũng hỗ trợ điều này. Nghĩa là các phần đó có thể được khởi tạo từ việc copy một nguyên mẫu từ một “cấu trúc” khác. Miễn là các phần kết hợp đều thể hiện clone() và được sử dụng với cấu trúc khác nhau làm nguyên mẫu. Xem thêm về [Object cloning trong java](https://gpcoder.com/2361-object-cloning-trong-java/) bạn sẽ thấy rõ điều này.

## Sử dụng Prototype khi nào?

* Chúng ta có một object và cần phải tạo 1 ọbject mới khác dựa trên object bạn đầu mà không thể sử dụng toán tử new hay các hàm contructor để khởi tạo. Vì sao vậy? Lý do đơn giản là ở đây chúng ta ko hề được biết thông tin nội tại của object đó hoặc object đó đã có thể bị che dấu đi nhiều thông tin khác mà chỉ cho ta một thông tin rất giới hạn không đủ để hiểu được. Do vậy ta ko thể dùng toán tử new để khởi tạo nó được. Giải pháp: để cho chính object mẫu tự xác định thông tin và dữ liệu sao chép.
* Khởi tạo đối tượng lúc run-time: chúng ta có thể xác định đối tượng cụ thể sẽ được khởi tạo lúc runtime nếu class được implement / extend từ một Prototype.
* Cấu hình một ứng dụng với dynamic class.
* Muốn truyền đối tượng vào một hàm nào đó để xử lý, thay vì truyền đối tượng gốc có thể ảnh hưởng dữ liệu thì ta có thể truyền đối tượng sao chép.
* Chi phí của việc tạo mới đối tượng (bằng cách sử dụng toán tử new) là lớn.
* Ẩn độ phức tạp của việc khởi tạo đối tượng từ phía Client.

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/prototype>
* <https://refactoring.guru/design-patterns/prototype>
* <https://www.tutorialspoint.com/design_pattern/prototype_pattern.htm>
* <https://www.javatpoint.com/prototype-design-pattern>
* <http://www.javacamp.org/designPattern/>

# Hướng dẫn Java Design Pattern – Adapter

Đăng vào 15/10/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 4281 Lượt xem

Trong các bài viết trước chúng ta đã cùng tìm hiểu về các Pattern thuộc nhóm [Creational Design Pattern](https://gpcoder.com/tag/creational-design-pattern/). Trong bài này, chúng ta sẽ tìm hiểu qua một Pattern khác thuộc nhóm Structural Design Pattern là Adpater Pattern.

Nội dung [[Ẩn](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/)]

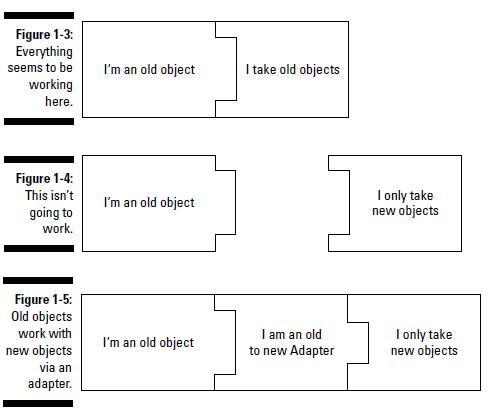
* [1 Adapter Pattern là gì?](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/" \l "Adapter_Pattern_la_gi)
* [2 Cài đặt Adapter Pattern như thế nào?](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/" \l "Cai_dat_Adapter_Pattern_nhu_the_nao)
  + [2.1 Ví dụ Adapter Pattern với ứng dụng Translation](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/" \l "Vi_du_Adapter_Pattern_voi_ung_dung_Translation)
  + [2.2 Ví dụ Adapter Pattern với BufferedReader](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/" \l "Vi_du_Adapter_Pattern_voiBufferedReader)
* [3 Lợi ích của Adapter Pattern là gì?](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/" \l "Loi_ich_cuaAdapter_Pattern_la_gi)
* [4 Sử dụng Adapter Pattern khi nào?](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/" \l "Su_dung_Adapter_Pattern_khi_nao)

## Adapter Pattern là gì?

[Adapter Pattern](https://vi.wikipedia.org/wiki/Adapter_pattern) is a structural design pattern that convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

Adapter Pattern (Người chuyển đổi) là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Adapter Pattern cho phép các inteface (giao diện) không liên quan tới nhau có thể làm việc cùng nhau. Đối tượng giúp kết nối các interface gọi là Adapter.

Adapter Pattern giữ vai trò trung gian giữa hai lớp, chuyển đổi interface của một hay nhiều lớp có sẵn thành một interface khác, thích hợp cho lớp đang viết. Điều này cho phép các lớp có các interface khác nhau có thể dễ dàng giao tiếp tốt với nhau thông qua interface trung gian, không cần thay đổi code của lớp có sẵn cũng như lớp đang viết.



Adapter Pattern còn gọi là Wrapper Pattern do cung cấp một interface “bọc ngoài” tương thích cho một hệ thống có sẵn, có dữ liệu và hành vi phù hợp nhưng có interface không tương thích với lớp đang viết.

Ví dụ:

* Cái phích cắm điện có 3 chân nhưng ổ điện chỉ có 2 lỗ thì phải dùng thêm 1 cái bộ chuyển để chuyển từ 3 chân sang 2 chân – bộ chuyển này cũng được gọi là Adapter.



* Một ví dụ khác là laptop không sử dụng nguồn điện xoay chiều 224V, nên để laptop có thể sử dụng được nguồn điện 224V cần có một adapter làm cầu nối trung gian để chuyển nguồn điện xoay chiều 224V thành nguồn điện 1 chiều 12V.



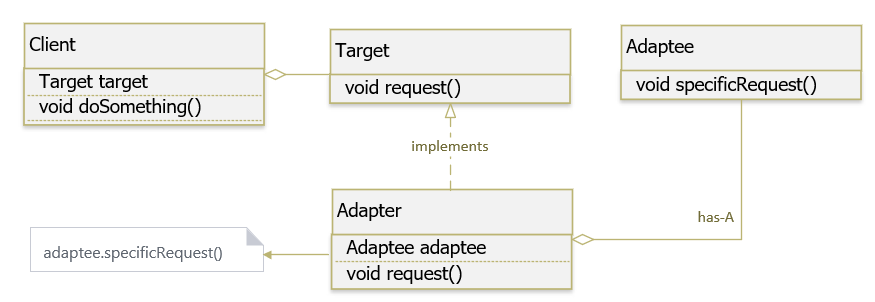
## Cài đặt Adapter Pattern như thế nào?

Một Adapter Pattern bao gồm các thành phần cơ bản sau:

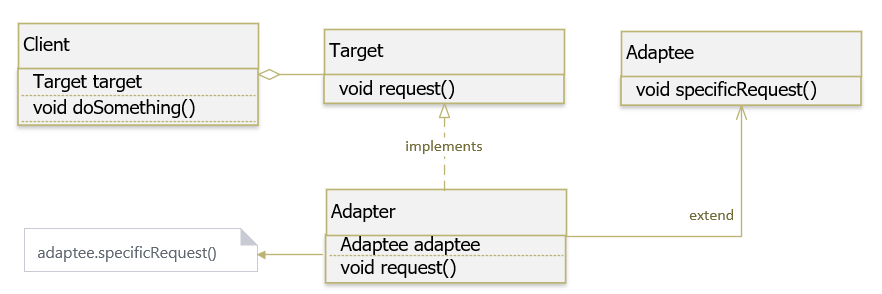
* Adaptee: định nghĩa interface không tương thích, cần được tích hợp vào.
* Adapter: lớp tích hợp, giúp interface không tương thích tích hợp được với interface đang làm việc. Thực hiện việc chuyển đổi interface cho Adaptee và kết nối Adaptee với Client.
* Target: một interface chứa các chức năng được sử dụng bởi Client (domain specific).
* Client: lớp sử dụng các đối tượng có interface Target.

Có hai cách để thực hiện Adapter Pattern dựa theo cách cài đặt (implement) của chúng:

* Object Adapter – Composition (Tổng hợp): trong mô hình này, một lớp mới (Adapter) sẽ tham chiếu đến một (hoặc nhiều) đối tượng của lớp có sẵn với interface không tương thích (Adaptee), đồng thời cài đặt interface mà người dùng mong muốn (Target). Trong lớp mới này, khi cài đặt các phương thức của interface người dùng mong muốn, sẽ gọi phương thức cần thiết thông qua đối tượng thuộc lớp có interface không tương thích.



* Class Adapter – Inheritance (Kế thừa) : trong mô hình này, một lớp mới (Adapter) sẽ kế thừa lớp có sẵn với interface không tương thích (Adaptee), đồng thời cài đặt interface mà người dùng mong muốn (Target). Trong lớp mới, khi cài đặt các phương thức của interface người dùng mong muốn, phương thức này sẽ gọi các phương thức cần thiết mà nó thừa kế được từ lớp có interface không tương thích.



So sánh Class Adapter với Object Adapter:

* Sự khác biệt chính là Class Adapter sử dụng Inheritance (kế thừa) để kết nối Adapter và Adaptee trong khi Object Adapter sử dụng Composition (tổng hợp) để kết nối Adapter và Adaptee.
* Trong cách tiếp cận Class Adapter, nếu một Adaptee là một class và không phải là một interface thì Adapter sẽ là một lớp con của Adaptee. Do đó, nó sẽ không phục vụ tất cả các lớp con khác theo cùng một cách vì Adapter là một lớp phụ cụ thể của Adaptee.

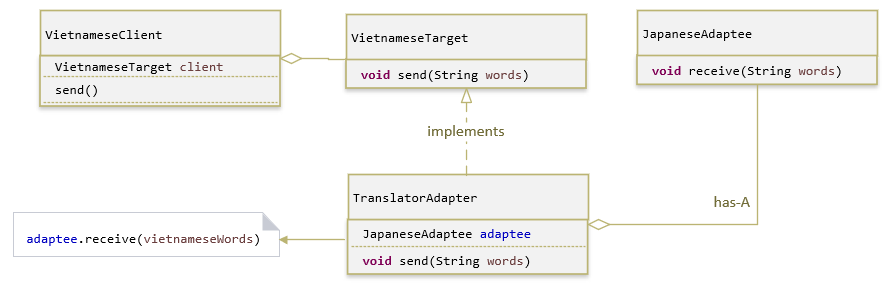
Tại sao Object Adapter lại tốt hơn?

* Nó sử dụng Composition để giữ một thể hiện của Adaptee, cho phép một Adapter hoạt động với nhiều Adaptee nếu cần thiết.

### Ví dụ Adapter Pattern với ứng dụng Translation

Một người Việt muốn trao đổi với một người Nhật. Tuy nhiên, 2 người này không biết ngôn ngữ của nhau nên cần phải có một người để chuyển đổi từ ngôn ngữ tiếng Việt sang ngôn ngữ tiếng Nhật. Chúng ta sẽ mô hình hóa trường hợp này với Adapter Pattern như sau:

* Client: người Việt sẽ là Client trong ví dụ này,vì anh ta cần gửi một số message cho người Nhật.
* Target: đây là nội dung message được Client cung cấp cho thông dịch viên (Translator / Adapter).
* Adapter: thông dịch viên (Translator) sẽ là Adapter, nhận message tiếng Việt từ Client và chuyển đổi nó sang tiếng Nhật trước khi gởi cho người Nhật.
* Adaptee: đây là interface hoặc class được người Nhật sử dụng để nhận message được chuyển đổi từ thông dịch viên (Translator).



VietnameseTarget.java

package com.gpcoder.patterns.structural.adapter;

public interface VietnameseTarget {

void send(String words);

}

JapaneseAdaptee.java

package com.gpcoder.patterns.structural.adapter;

public class JapaneseAdaptee {

public void receive(String words) {

System.out.println("Retrieving words from Adapter ...");

System.out.println(words);

}

}

TranslatorAdapter.java

package com.gpcoder.patterns.structural.adapter;

public class TranslatorAdapter implements VietnameseTarget {

private JapaneseAdaptee adaptee;

public TranslatorAdapter(JapaneseAdaptee adaptee) {

this.adaptee = adaptee;

}

@Override

public void send(String words) {

System.out.println("Reading Words ...");

System.out.println(words);

String vietnameseWords = this.translate(words);

System.out.println("Sending Words ...");

adaptee.receive(vietnameseWords);

}

private String translate(String vietnameseWords) {

System.out.println("Translated!");

return "こんにちは";

}

}

VietnameseClient.java

package com.gpcoder.patterns.structural.adapter;

public class VietnameseClient {

public static void main(String[] args) {

VietnameseTarget client = new TranslatorAdapter(new JapaneseAdaptee());

client.send("Xin chào");

}

}

Output của chương trình trên:

Reading Words ...

Xin chào

Translated!

Sending Words ...

Retrieving words from Adapter ...

こんにちは

### Ví dụ Adapter Pattern với BufferedReader

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

System.out.print("Enter your name: ");

String s = br.readLine();

Như bạn thấy:

* System.in: đây là một Adaptee. System.in là một static instance của lớp InputStream, nó đọc dữ liệu từ Console và trả về 1 byte stream.
* BufferedReader : đây là Target, nó chấp nhận dữ liệu là một character stream.
* InputStreamReader : đây là một Adapter ở giữa hai interface không tương thích: System.in và BufferedReader giúp cho chúng có thể hoạt động được với nhau.
* Client: là ứng dụng sẽ làm việc với Target interface.

## Lợi ích của Adapter Pattern là gì?

Việc sử dụng Adapter Pattern đem lại các lợi ích sau:

* Cho phép nhiều đối tượng có interface giao tiếp khác nhau có thể tương tác và giao tiếp với nhau.
* Tăng khả năng sử dụng lại thư viện với interface không thay đổi do không có mã nguồn.

Bên cạnh những lợi ích trên, nó cũng nó một số khuyết điểm nhỏ sau:

* Tất cả các yêu cầu được chuyển tiếp, do đó làm tăng thêm một ít chi phí.
* Đôi khi có quá nhiều Adapter được thiết kế trong một chuỗi Adapter (chain) trước khi đến được yêu cầu thực sự.

## Sử dụng Adapter Pattern khi nào?

Có thể dùng Adapter Pattern trong những trường hợp sau:

* Adapter Pattern giúp nhiều lớp có thể làm việc với nhau dễ dàng mà bình thường không thể. Một trường hợp thường gặp phải và có thể áp dụng Adapter Pattern là khi không thể kế thừa lớp A, nhưng muốn một lớp B có những xử lý tương tự như lớp A. Khi đó chúng ta có thể cài đặt B theo Object Adapter, các xử lý của B sẽ gọi những xử lý của A khi cần.
* Khi muốn sử dụng một lớp đã tồn tại trước đó nhưng interface sử dụng không phù hợp như mong muốn.
* Khi muốn tạo ra những lớp có khả năng sử dụng lại, chúng phối hợp với các lớp không liên quan hay những lớp không thể đoán trước được và những lớp này không có những interface tương thích.
* Cần phải có sự chuyển đổi interface từ nhiều nguồn khác nhau.
* Khi cần đảm bảo nguyên tắc [Open/ Close](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Open-Closed_principle_OCP) trong một ứng dụng.

Một vài class sử dụng Adapter Pattern:

* java.util.Arrays#asList()
* java.io.InputStreamReader(InputStream) (returns a Reader)
* java.io.OutputStreamWriter(OutputStream) (returns a Writer)
* javax.xml.bind.annotation.adapters.XmlAdapter#marshal() và #unmarshal()

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/adapter>
* <https://refactoring.guru/design-patterns/adapter>
* <https://www.javatpoint.com/adapter-pattern>
* <https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm>
* <https://www.javagists.com/adapter-design-pattern>
* <https://www.oodesign.com/adapter-pattern.html>

# Hướng dẫn Java Design Pattern – Bridge

Đăng vào 22/10/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 2479 Lượt xem

Một thành phần trong OOP thường có 2 phần: Phần trừu tượng (abstraction) định nghĩa các chức năng và phần thực thi (implementation) các chức năng được định nghĩa trong phần trừu tượng. Hai phần này liên hệ với nhau thông qua quan hệ kế thừa. Những thay đổi trong phần trừu tượng dẫn đến các thay đổi trong phần thực thi.

Bridge Pattern được sử dụng để tách thành phần trừu tượng (abstraction) và thành phần thực thi (implementation) riêng biệt. Do đó, các thành phần này có thể thay đổi một cách độc lập mà không ảnh hưởng đến các thành phần khác. Thay vì liên hệ với nhau bằng quan hệ kế thừa, hai thành phần này liên hệ với nhau thông qua quan hệ “chứa trong” (object composition).

Nội dung [[Ẩn](https://gpcoder.com/4520-huong-dan-java-design-pattern-bridge/)]

* [1 Bridge Pattern là gì?](https://gpcoder.com/4520-huong-dan-java-design-pattern-bridge/" \l "Bridge_Pattern_la_gi)
* [2 Cài đặt Bridge Pattern như thế nào?](https://gpcoder.com/4520-huong-dan-java-design-pattern-bridge/" \l "Cai_dat_Bridge_Pattern_nhu_the_nao)
* [3 Lợi ích của Bridge Pattern là gì?](https://gpcoder.com/4520-huong-dan-java-design-pattern-bridge/" \l "Loi_ich_cua_Bridge_Pattern_la_gi)
* [4 Sử dụng Bridge Pattern khi nào?](https://gpcoder.com/4520-huong-dan-java-design-pattern-bridge/" \l "Su_dung_Bridge_Pattern_khi_nao)

## Bridge Pattern là gì?

Decouple an abstraction from its implementation so that the two can vary independently.

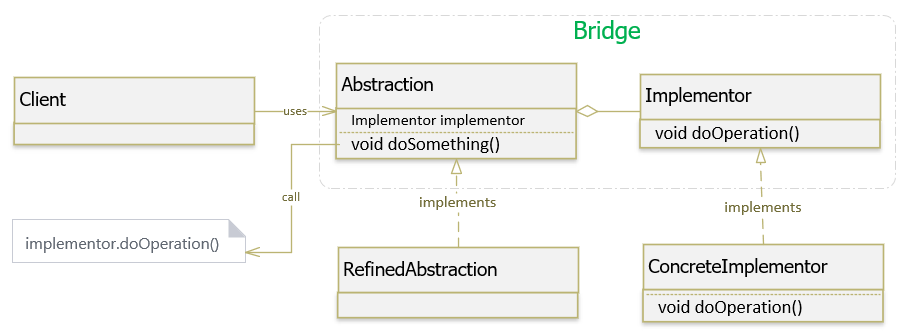
Bridge Pattern là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Ý tưởng của nó là tách tính trừu tượng (abstraction) ra khỏi tính hiện thực (implementation) của nó. Từ đó có thể dễ dàng chỉnh sửa hoặc thay thế mà không làm ảnh hưởng đến những nơi có sử dụng lớp ban đầu.

Điều đó có nghĩa là, ban đầu chúng ta thiết kế một class với rất nhiều xử lý, bây giờ chúng ta không muốn để những xử lý đó trong class đó nữa. Vì thế, chúng ta sẽ tạo ra một class khác và move các xử lý đó qua class mới. Khi đó, trong lớp cũ sẽ giữ một đối tượng thuộc về lớp mới, và đối tượng này sẽ chịu trách nhiệm xử lý thay cho lớp ban đầu.

Bridge Pattern khá giống với mẫu [Adapter Pattern](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/) ở chỗ là sẽ nhờ vào một lớp khác để thực hiện một số xử lý nào đó. Tuy nhiên, ý nghĩa và mục đích sử dụng của hai mẫu thiết kế này hoàn toàn khác nhau:

* Adapter Pattern hay còn gọi là Wrapper pattern được dùng để biến đổi một class/ interface sang một dạng khác có thể sử dụng được. Adapter Pattern giúp các lớp không tương thích hoạt động cùng nhau mà bình thường là không thể.
* Bridge Pattern được sử dụng được sử dụng để tách thành phần trừu tượng (abstraction) và thành phần thực thi (implementation) riêng biệt.
* Adapter Pattern làm cho mọi thứ có thể hoạt động với nhau sau khi chúng đã được thiết kế (đã tồn tại). Bridge Pattern nên được thiết kế trước khi phát triển hệ thống để Abstraction và Implementation có thể thực hiện một cách độc lập.

## Cài đặt Bridge Pattern như thế nào?

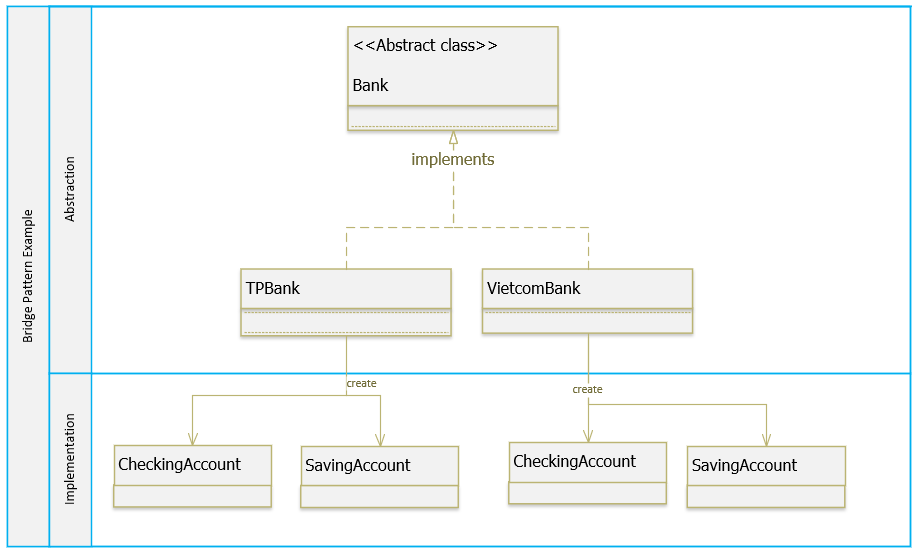


Một Bridge Pattern bao gồm các thành phần sau:

* Client: đại diện cho khách hàng sử dụng các chức năng thông qua Abstraction.
* Abstraction : định ra một abstract interface quản lý việc tham chiếu đến đối tượng hiện thực cụ thể (Implementor).
* Refined Abstraction (**AbstractionImpl)** : hiện thực (implement) các phương thức đã được định ra trong Abstraction bằng cách sử dụng một tham chiếu đến một đối tượng của Implementer.
* Implementor : định ra các interface cho các lớp hiện thực. Thông thường nó là interface định ra các tác vụ nào đó của Abstraction.
* ConcreteImplementor : hiện thực Implementor interface.

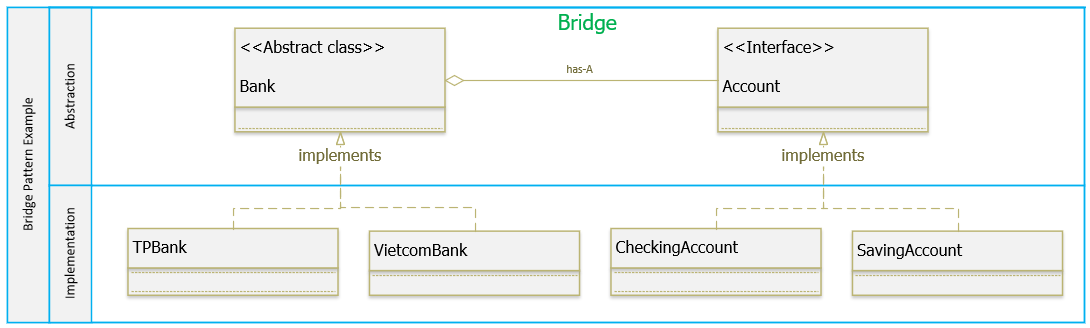
Ví dụ:

Một hệ thống ngân hàng cung cấp các loại tài khoản khác nhau cho khách hàng, chẳng hạn: Checking account và Saving account. Chúng ta có sơ đồ như sau:



Với cách thiết kế như vậy, khi hệ thống cần cung cấp thêm một loại tài khoản khác, chúng ta phải tạo class mới cho tất cả các ngân hàng, số lượng class tăng lên rất nhiều.

Bây giờ, chúng ta sẽ sử dụng Bridge Pattern để tái cấu trúc lại hệ thống trên như sau:



Với cấu trúc mới như vậy, khi có thêm một loại tài khoản mới, chúng ta đơn chỉ việc thêm vào một implement mới cho Account, các thành phần khác của Bank không bị ảnh hưởng. Hoặc cần thêm một ngân hàng mới, chẳng hạn VietinBank chúng ta chỉ cần thêm implement mới cho Bank, các thành phần khác cũng không bị ảnh hưởng và số lượng class chỉ tăng lên 1.

Code cho chương trên như sau:

Account:

public interface Account {

void openAccount();

}

CheckingAccount:

public class CheckingAccount implements Account {

@Override

public void openAccount() {

System.out.println("Checking Account");

}

}

SavingAccount:

public class SavingAccount implements Account {

@Override

public void openAccount() {

System.out.println("Saving Account");

}

}

Bank:

public abstract class Bank {

protected Account account;

public Bank(Account account) {

this.account = account;

}

public abstract void openAccount();

}

VietcomBank:

public class VietcomBank extends Bank {

public VietcomBank(Account account) {

super(account);

}

@Override

public void openAccount() {

System.out.print("Open your account at VietcomBank is a ");

account.openAccount();

}

}

TPBank:

public class TPBank extends Bank {

public TPBank(Account account) {

super(account);

}

@Override

public void openAccount() {

System.out.print("Open your account at TPBank is a ");

account.openAccount();

}

}

Client:

public class Client {

public static void main(String[] args) {

Bank vietcomBank = new VietcomBank(new CheckingAccount());

vietcomBank.openAccount();

Bank tpBank = new TPBank(new CheckingAccount());

tpBank.openAccount();

}

}

Output của chương trình trên:

Open your account at VietcomBank is a Checking Account

Open your account at TPBank is a Checking Account

## Lợi ích của Bridge Pattern là gì?

* Giảm sự phục thuộc giữa abstraction và implementation (loose coupling): tính kế thừa trong OOP thường gắn chặt abstraction và implementation lúc build chương trình. Bridge Pattern có thể được dùng để cắt đứt sự phụ thuộc này và cho phép chúng ta chọn implementation phù hợp lúc runtime.
* Giảm số lượng những lớp con không cần thiết: một số trường hợp sử dụng tính inheritance sẽ tăng số lượng subclass rất nhiều. Ví dụ: trường hợp chương trình view hình ảnh trên các hệ điều hành khác nhau, ta có 6 loại hình (JPG, PNG, GIF, BMP, JPEG, TIFF) và 3 hệ điều hành (Window, MacOS, Ubuntu). Sử dụng inheritance trong trường hợp này sẽ làm ta thiết kế 18 lớp: JpgWindow, PngWindow, GifWindow, …. Trong khi áp dụng Bridge sẽ giảm số lượng lớp xuống 9 lớp: 6 lớp ứng với từng implement của Image và 3 lớp ứng với từng hệ điều hành, mỗi hệ điều hành sẽ gồm một tham chiếu đến đối tượng Image cụ thể.
* Code sẽ gọn gàn hơn và kích thước ứng dụng sẽ nhỏ hơn: do giảm được số class không cần thiết.
* Dễ bảo trì hơn: các Abstraction và Implementation của nó sẽ dễ dàng thay đổi lúc runtime cũng như khi cần thay đổi thêm bớt trong tương lai.
* Dễ dàng mở rộng về sau: thông thường các ứng dụng lớn thường yêu cầu chúng ta thêm module cho ứng dụng có sẵn nhưng không được sửa đổi framework/ứng dụng có sẵn vì các framework/ứng dụng đó có thể được công ty nâng cấp lên version mới. Bridge Pattern sẽ giúp chúng ta trong trường hợp này.
* Cho phép ẩn các chi tiết implement từ client: do abstraction và implementation hoàn toàn độc lập nên chúng ta có thể thay đổi một thành phần mà không ảnh hưởng đến phía Client. Ví dụ, các lớp của chương trình view ảnh sẽ độc lập với thuật toán vẽ ảnh trong các implementation. Như vậy ta có thể update chương trình xem ảnh khi có một thuật toán vẽ ảnh mới mà không cần phải sửa đổi nhiều.

## Sử dụng Bridge Pattern khi nào?

* Khi bạn muốn tách ràng buộc giữa Abstraction và Implementation, để có thể dễ dàng mở rộng độc lập nhau.
* Cả Abstraction và Implementation của chúng nên được mở rộng bằng subsclass.
* Sử dụng ở những nơi mà những thay đổi được thực hiện trong implement không ảnh hưởng đến phía client.

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/bridge>
* <https://refactoring.guru/design-patterns/bridge>
* <https://vi.wikipedia.org/wiki/Bridge_pattern>
* <https://www.javatpoint.com/bridge-pattern>
* <https://www.tutorialspoint.com/design_pattern/bridge_pattern.htm>
* <http://www.thejavageek.com/2016/10/20/bridge-design-pattern-implementation/>

# Hướng dẫn Java Design Pattern – Composite

Đăng vào 02/11/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 2666 Lượt xem

Trong các bài viết trước chúng ta đã tìm hiểu về [Bridge Pattern](https://gpcoder.com/4520-huong-dan-java-design-pattern-bridge/) và [Adapter Pattern](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/). Trong bài viết này chúng ta tiếp tục tìm hiểu một Design Pattern khác thuộc nhóm cấu trúc là Composite Pattern.

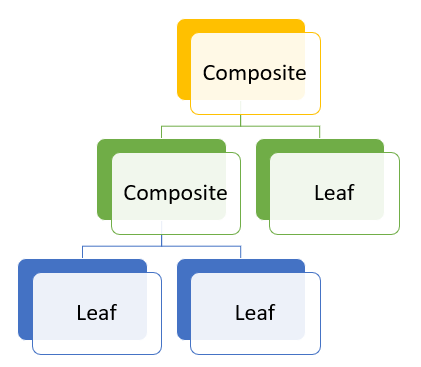
Nội dung [[Ẩn](https://gpcoder.com/4554-huong-dan-java-design-pattern-composite/)]

* [1 Composite Pattern là gì?](https://gpcoder.com/4554-huong-dan-java-design-pattern-composite/" \l "Composite_Pattern_la_gi)
* [2 Cài đặt Composite Pattern như thế nào?](https://gpcoder.com/4554-huong-dan-java-design-pattern-composite/" \l "Cai_dat_Composite_Pattern_nhu_the_nao)
* [3 Lợi ích của Composite Pattern là gì?](https://gpcoder.com/4554-huong-dan-java-design-pattern-composite/" \l "Loi_ich_cua_Composite_Pattern_la_gi)
* [4 Sử dụng Composite Pattern khi nào?](https://gpcoder.com/4554-huong-dan-java-design-pattern-composite/" \l "Su_dung_Composite_Pattern_khi_nao)

## Composite Pattern là gì?

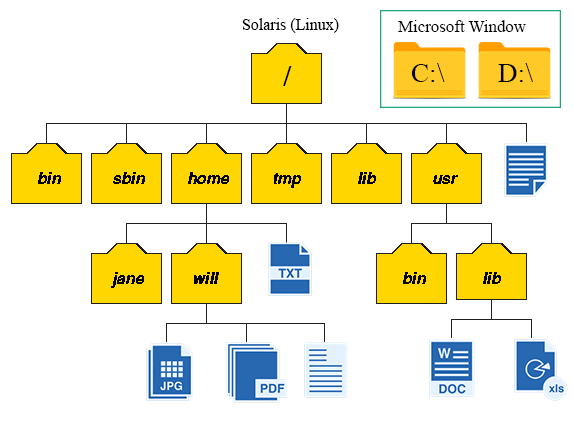
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Composite là một mẫu thiết kế thuộc nhóm cấu trúc (Structural Pattern). Composite Pattern là một sự tổng hợp những thành phần có quan hệ với nhau để tạo ra thành phần lớn hơn. Nó cho phép thực hiện các tương tác với tất cả đối tượng trong mẫu tương tự nhau.



Composite Pattern được sử dụng khi chúng ta cần xử lý một nhóm đối tượng tương tự theo cách xử lý 1 object. Composite pattern sắp xếp các object theo cấu trúc cây để diễn giải 1 phần cũng như toàn bộ hệ thống phân cấp. Pattern này tạo một lớp chứa nhóm đối tượng của riêng nó. Lớp này cung cấp các cách để sửa đổi nhóm của cùng 1 object. Pattern này cho phép Client có thể viết code giống nhau để tương tác với composite object này, bất kể đó là một đối tượng riêng lẻ hay tập hợp các đối tượng.

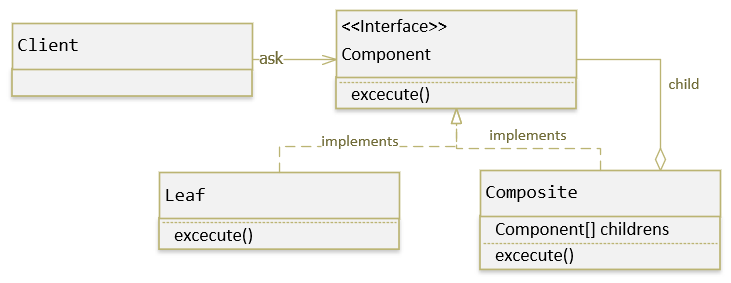
Ví dụ: Một chương trình quản lý một hệ thống tập tin với cấu trúc như hình sau:



Một hệ thống tập tin là một cấu trúc cây có chứa các nhánh là các thư mục (folder – composite), cũng như các nút lá là các tệp (file – leaf). Một folder có thể chứa một hoặc nhiều file hoặc folder. Do đó, folder là một đối tượng phức tạp và file là một đối tượng đơn giản. File và Folder có nhiều thao tác và thuộc tính chung, chẳng hạn như: di chuyển (cut) , sao chép (copy), liệt kê (view) hoặc các thuộc tính thư mục như tên tệp và kích thước.

Với cấu trúc như vậy sẽ dễ dàng và thuận tiện hơn để quản lý file và folder thống nhất bằng cách xây dựng một Interface có đầy đủ các phương thức và thuộc tính chung cho cả file và folder.

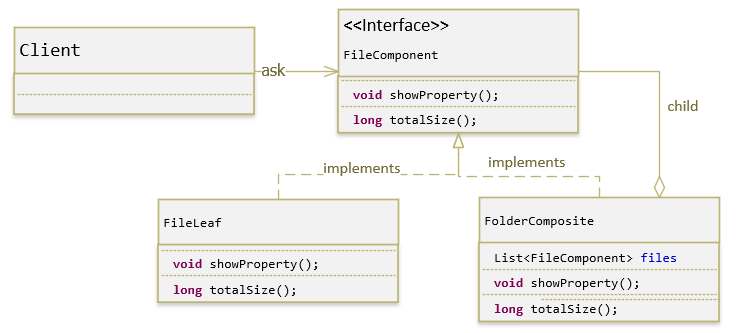
## Cài đặt Composite Pattern như thế nào?



Một Composite Pattern bao gồm các thành phần cơ bản sau:

* Base Component : là một interface hoặc abstract class quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.
* Leaf : là lớp hiện thực (implements) các phương thức của Component. Nó là các object không có con.
* Composite : lưu trữ tập hợp các Leaf và cài đặt các phương thức của Base Component. Composite cài đặt các phương thức được định nghĩa trong interface Component bằng cách ủy nhiệm cho các thành phần con xử lý.
* Client: sử dụng Base Component để làm việc với các đối tượng trong Composite.

Ví dụ: Cài đặt Composite Pattern về chương trình quản lý một hệ thống tập tin ở trên.



* Trước hết chúng ta định nghĩa một Inteface Component (FileComponent) có các phương thức chung cho cả folder và file. Để đơn giản, tôi chỉ tạo 2 phương thức showProperty() và totalSize(). Hai phương thức này sẽ cung cấp thông tin về file và tổng kích thước của nó.
* Tiếp theo, chúng ta sẽ tạo một class Leaf cài đặt các phương thức của Component (FileLeaf). Một class Composite chứa tập hợp các Leaf và cài đặt các phương thức của Component (FolderComposite).
* Cuối cùng, chúng ta sẽ tạo một class Client gọi các phương thức của FileComponent và FolderComposite. Cách gọi các phương thức của 2 class này hoàn toàn giống nhau do cùng implement một Component (FileComponent).

FileComponent.java

package com.gpcoder.patterns.structural.composite;

public interface FileComponent {

void showProperty();

long totalSize();

}

FileLeaf.java

package com.gpcoder.patterns.structural.composite;

public class FileLeaf implements FileComponent {

private String name;

private long size;

public FileLeaf(String name, long size) {

super();

this.name = name;

this.size = size;

}

@Override

public long totalSize() {

return size;

}

@Override

public void showProperty() {

System.out.println("FileLeaf [name=" + name + ", size=" + size + "]");

}

}

FolderComposite.java

package com.gpcoder.patterns.structural.composite;

import java.util.ArrayList;

import java.util.List;

public class FolderComposite implements FileComponent {

private List<FileComponent> files = new ArrayList<>();

public FolderComposite(List<FileComponent> files) {

this.files = files;

}

@Override

public void showProperty() {

for (FileComponent file : files) {

file.showProperty();

}

}

@Override

public long totalSize() {

long total = 0;

for (FileComponent file : files) {

total += file.totalSize();

}

return total;

}

}

Client.java

package com.gpcoder.patterns.structural.composite;

import java.util.Arrays;

import java.util.List;

public class Client {

public static void main(String[] args) {

FileComponent file1 = new FileLeaf("file 1", 10);

FileComponent file2 = new FileLeaf("file 2", 5);

FileComponent file3 = new FileLeaf("file 3", 12);

List<FileComponent> files = Arrays.asList(file1, file2, file3);

FileComponent folder = new FolderComposite(files);

folder.showProperty();

System.out.println("Total Size: " + folder.totalSize());

}

}

Output của chương trình trên:

FileLeaf [name=file 1, size=10]

FileLeaf [name=file 2, size=5]

FileLeaf [name=file 3, size=12]

Total Size: 27

## Lợi ích của Composite Pattern là gì?

* Cung cấp cùng một cách sử dụng đối với từng đối tượng riêng lẻ hoặc nhóm các đối tượng với nhau.

## Sử dụng Composite Pattern khi nào?

* Composite Pattern chỉ nên được áp dụng khi nhóm đối tượng phải hoạt động như một đối tượng duy nhất (theo cùng một cách).
* Composite Pattern có thể được sử dụng để tạo ra một cấu trúc giống như cấu trúc cây.

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/composite>
* <https://refactoring.guru/design-patterns/composite>
* <https://www.tutorialspoint.com/design_pattern/composite_pattern.htm>
* <https://www.javatpoint.com/composite-pattern>
* <https://vi.wikipedia.org/wiki/Composite_pattern>
* <https://www.journaldev.com/1535/composite-design-pattern-in-java>

# Hướng dẫn Java Design Pattern – Decorator

Đăng vào 08/11/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 2739 Lượt xem

Một trong những khía cạnh quan trọng nhất trong quá trình phát triển một ứng dụng mà các lập trình viên phải đối đầu là sự thay đổi. Khi muốn thêm hoặc loại bỏ một tính năng của một đối tượng, điều đầu tiên chúng ta nghĩ đến là thừa kế (extends). Tuy nhiên, thừa kế không khả thi vì nó là static, chúng ta không thể thêm các lớp con mới vào một chương trình khi nó đã được biên dịch và thực thi. Để giải quyết vấn đề này, chúng ta có thể sử dụng Decorator Pattern được giới thiệu trong phần tiếp theo của bài viết này.

Nội dung [[Ẩn](https://gpcoder.com/4574-huong-dan-java-design-pattern-decorator/)]

* [1 Decorator Pattern là gì?](https://gpcoder.com/4574-huong-dan-java-design-pattern-decorator/" \l "Decorator_Pattern_la_gi)
* [2 Cài đặt Decorator Pattern như thế nào?](https://gpcoder.com/4574-huong-dan-java-design-pattern-decorator/" \l "Cai_dat_Decorator_Pattern_nhu_the_nao)
* [3 Lợi ích của Decorator Pattern là gì?](https://gpcoder.com/4574-huong-dan-java-design-pattern-decorator/" \l "Loi_ich_cua_Decorator_Pattern_la_gi)
* [4 Sử dụng Decorator Pattern khi nào?](https://gpcoder.com/4574-huong-dan-java-design-pattern-decorator/" \l "Su_dung_Decorator_Pattern_khi_nao)
* [5 So sánh Decorator và Adapter](https://gpcoder.com/4574-huong-dan-java-design-pattern-decorator/" \l "So_sanh_Decorator_va_Adapter)

## Decorator Pattern là gì?

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator pattern là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Nó cho phép người dùng thêm chức năng mới vào đối tượng hiện tại mà không muốn ảnh hưởng đến các đối tượng khác. Kiểu thiết kế này có cấu trúc hoạt động như một lớp bao bọc (wrap) cho lớp hiện có. Mỗi khi cần thêm tính năng mới, đối tượng hiện có được wrap trong một đối tượng mới (decorator class).

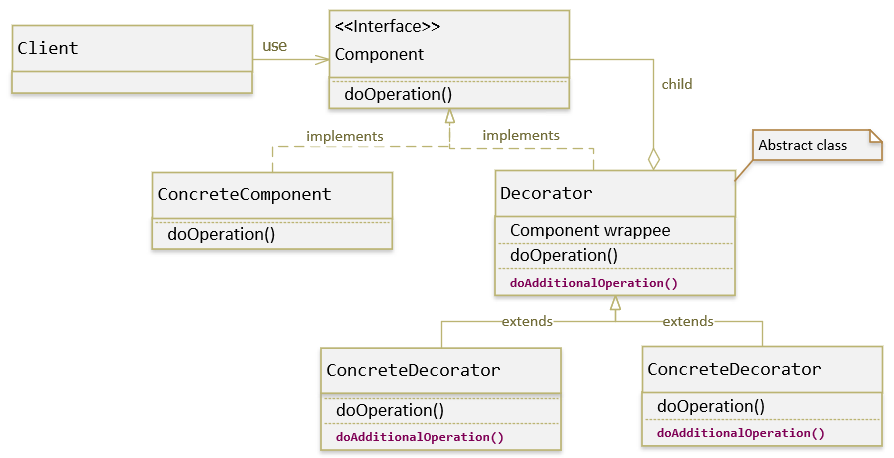
Decorator pattern sử dụng [composition](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/" \l "Cai_dat_Adapter_Pattern_nhu_the_nao) thay vì inheritance (thừa kế) để mở rộng đối tượng. Decorator pattern còn được gọi là Wrapper hay Smart Proxy.

## Cài đặt Decorator Pattern như thế nào?

Decorator pattern hoạt động dựa trên một đối tượng đặc biệt, được gọi là decorator (hay wrapper). Nó có cùng một interface như một đối tượng mà nó cần bao bọc (wrap), vì vậy phía client sẽ không nhận thấy khi bạn đưa cho nó một wrapper thay vì đối tượng gốc.

Tất cả các wrapper có một trường để lưu trữ một giá trị của một đối tượng gốc. Hầu hết các wrapper khởi tạo trường đó với một đối tượng được truyền vào constructor của chúng.

Vậy làm thế nào để có thể thay đổi hành vi của đối tượng? Như đã đề cập, wrapper có cùng interface với các đối tượng đích. Khi bạn gọi một phương thức decorator, nó thực hiện cùng một phương thức trong một đối tượng được wrap và sau đó thêm một cái gì đó (tính năng mới) vào kết quả, công việc này tùy thuộc vào logic nghiệp vụ.



Các thành phần trong mẫu thiết kế Decorator:

* Component: là một interface quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.
* ConcreteComponent : là lớp hiện thực (implements) các phương thức của Component.
* Decorator : là một abstract class dùng để duy trì một tham chiếu của đối tượng Component và đồng thời cài đặt các phương thức của Component  interface.
* ConcreteDecorator : là lớp hiện thực (implements) các phương thức của Decorator, nó cài đặt thêm các tính năng mới cho Component.
* Client : đối tượng sử dụng Component.

Ví dụ:

Để đơn giản hơn, chúng ta xem ví dụ về một hệ thống quản lý dự án, nơi nhân viên đang làm việc với các vai trò khác nhau, chẳng hạn như thành viên nhóm (team member), trưởng nhóm (team lead) và người quản lý (manager). Một thành viên trong nhóm chịu trách nhiệm thực hiện các nhiệm vụ được giao và phối hợp với các thành viên khác để hoàn thành nhiệm vụ nhóm. Mặt khác, một trưởng nhóm phải quản lý và cộng tác với các thành viên trong nhóm của mình và lập kế hoạch nhiệm vụ của họ. Tương tự như vậy, một người quản lý có thêm một số trách nhiệm đối với một trưởng nhóm như quản lý yêu cầu dự án, tiến độ, phân công công việc.

Sau đây là các thành phần tham gia vào hệ thống và hành vi của chúng:

* Employee: thực hiện công việc (doTask), tham gia vào dự án (join), rời khỏi dự án (terminate).
* Team member: báo cáo task được giao (report task), cộng tác với các thành viên khác (coordinate with others).
* Team lead: lên kế hoạch (planning), hỗ trợ các thành viên phát triển (motivate), theo dõi chất lượng công việc và thời gian (monitor).
* Manager: tạo các yêu cầu dự án (create requirement), giao nhiệm vụ cho thành viên (assign task), quản lý tiến độ dự án (progress management).

Với cách làm thông thường, chúng ta có sơ đồ như sau:

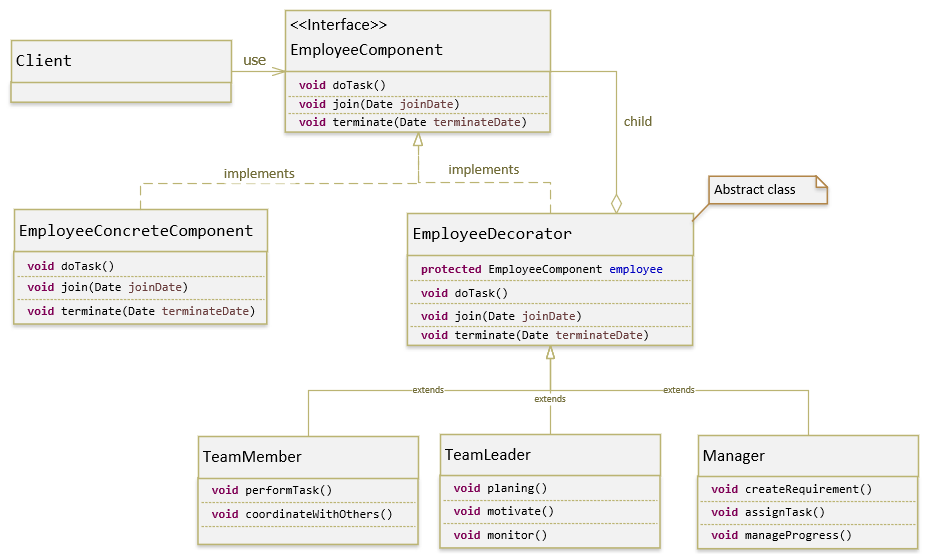


Bất cứ khi nào một thành viên trong nhóm trở thành một Team Lead, chúng ta phải tạo một đối tượng mới của Team Lead và đối tượng trước đó tham chiếu vào nhân viên đó (Team Member trong nhóm) có thể bị hủy hoặc lưu trữ. Đó không phải là cách tiếp cận được khuyến nghị khi nhân viên vẫn là một phần của tổ chức của bạn. Tương tự như trường hợp với Manager, khi một nhân viên trở thành người quản lý từ một Team Lead / Team Member.

Một trường hợp khác là khi một nhân viên có thể thực hiện trách nhiệm của một Team Member trong nhóm cũng như trách nhiệm của Team Lead hoặc một Manager. Trong trường hợp đó, bạn cần tạo hai đối tượng cho cùng một nhân viên là hoàn toàn sai.

Trong các kịch bản này, một Team Member/ Team Lead có thể có thêm trách nhiệm trong lúc thực hiện (run-time). Và trách nhiệm của họ có thể được chỉ định / thu hồi trong lúc run-time.

Hãy xem sơ đồ bên dưới để thấy được cách chúng ta áp dụng Decorator Pattern như thế nào trong trường hợp này.



Như bạn thấy, với Decorator hệ thống của chúng ta linh hoạt hơn rất nhiều. Chúng ta có thể dễ dàng gán một nhân viên sang vai trò TeamMember, TeamLeader, Manager.

EmployeeComponent.java

package com.gpcoder.patterns.structural.decorator;

import java.text.DateFormat;

import java.text.SimpleDateFormat;

import java.util.Calendar;

import java.util.Date;

public interface EmployeeComponent {

String getName();

void doTask();

void join(Date joinDate);

void terminate(Date terminateDate);

default String formatDate(Date theDate) {

DateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");

return sdf.format(theDate);

}

default void showBasicInformation() {

System.out.println("-------");

System.out.println("The basic information of " + getName());

join(Calendar.getInstance().getTime());

Calendar terminateDate = Calendar.getInstance();

terminateDate.add(Calendar.MONTH, 6);

terminate(terminateDate.getTime());

}

}

EmployeeConcreteComponent.java

package com.gpcoder.patterns.structural.decorator;

import java.util.Date;

public class EmployeeConcreteComponent implements EmployeeComponent {

private String name;

public EmployeeConcreteComponent (String name) {

this.name = name;

}

@Override

public String getName() {

return name;

}

@Override

public void join(Date joinDate) {

System.out.println(this.getName() + " joined on " + formatDate(joinDate));

}

@Override

public void terminate(Date terminateDate) {

System.out.println(this.getName() + " terminated on " + formatDate(terminateDate));

}

@Override

public void doTask() {

// Unassigned task

}

}

EmployeeDecorator.java

package com.gpcoder.patterns.structural.decorator;

import java.util.Date;

public abstract class EmployeeDecorator implements EmployeeComponent {

protected EmployeeComponent employee;

protected EmployeeDecorator(EmployeeComponent employee) {

this.employee = employee;

}

@Override

public String getName() {

return employee.getName();

}

@Override

public void join(Date joinDate) {

employee.join(joinDate);

}

@Override

public void terminate(Date terminateDate) {

employee.terminate(terminateDate);

}

}

Manager.java

package com.gpcoder.patterns.structural.decorator;

public class Manager extends EmployeeDecorator {

protected Manager(EmployeeComponent employee) {

super(employee);

}

public void createRequirement() {

System.out.println(this.employee.getName() + " is create requirements.");

}

public void assignTask() {

System.out.println(this.employee.getName() + " is assigning tasks.");

}

public void manageProgress() {

System.out.println(this.employee.getName() + " is managing the progress.");

}

@Override

public void doTask() {

employee.doTask();

createRequirement();

assignTask();

manageProgress();

}

}

TeamLeader.java

package com.gpcoder.patterns.structural.decorator;

public class TeamLeader extends EmployeeDecorator {

protected TeamLeader(EmployeeComponent employee) {

super(employee);

}

public void planing() {

System.out.println(this.employee.getName() + " is planing.");

}

public void motivate() {

System.out.println(this.employee.getName() + " is motivating his members.");

}

public void monitor() {

System.out.println(this.employee.getName() + " is monitoring his members.");

}

@Override

public void doTask() {

employee.doTask();

planing();

motivate();

monitor();

}

}

TeamMember.java

package com.gpcoder.patterns.structural.decorator;

public class TeamMember extends EmployeeDecorator {

protected TeamMember(EmployeeComponent employee) {

super(employee);

}

public void reportTask() {

System.out.println(this.employee.getName() + " is reporting his assigned tasks.");

}

public void coordinateWithOthers() {

System.out.println(this.employee.getName() + " is coordinating with other members of his team.");

}

@Override

public void doTask() {

employee.doTask();

reportTask();

coordinateWithOthers();

}

}

Client.java

package com.gpcoder.patterns.structural.decorator;

public class Client {

public static void main(String[] args) {

System.out.println("NORMAL EMPLOYEE: ");

EmployeeComponent employee = new EmployeeConcreteComponent("GPCoder");

employee.showBasicInformation();

employee.doTask();

System.out.println("\nTEAM LEADER: ");

EmployeeComponent teamLeader = new TeamLeader(employee);

teamLeader.showBasicInformation();

teamLeader.doTask();

System.out.println("\nMANAGER: ");

EmployeeComponent manager = new Manager(employee);

manager.showBasicInformation();

manager.doTask();

System.out.println("\nTEAM LEADER AND MANAGER: ");

EmployeeComponent teamLeaderAndManager = new Manager(teamLeader);

teamLeaderAndManager.showBasicInformation();

teamLeaderAndManager.doTask();

}

}

Output của chương trình:

NORMAL EMPLOYEE:

-------

The basic information of GPCoder

GPCoder joined on 04/11/2018

GPCoder terminated on 04/05/2019

TEAM LEADER:

-------

The basic information of GPCoder

GPCoder joined on 04/11/2018

GPCoder terminated on 04/05/2019

GPCoder is planing.

GPCoder is motivating his members.

GPCoder is monitoring his members.

MANAGER:

-------

The basic information of GPCoder

GPCoder joined on 04/11/2018

GPCoder terminated on 04/05/2019

GPCoder is create requirements.

GPCoder is assigning tasks.

GPCoder is managing the progress.

TEAM LEADER AND MANAGER:

-------

The basic information of GPCoder

GPCoder joined on 04/11/2018

GPCoder terminated on 04/05/2019

GPCoder is planing.

GPCoder is motivating his members.

GPCoder is monitoring his members.

GPCoder is create requirements.

GPCoder is assigning tasks.

GPCoder is managing the progress.

## Lợi ích của Decorator Pattern là gì?

* Tăng cường khả năng mở rộng của đối tượng, bởi vì những thay đổi được thực hiện bằng cách implement trên các lớp mới.
* Client sẽ không nhận thấy sự khác biệt khi bạn đưa cho nó một wrapper thay vì đối tượng gốc.
* Một đối tượng có thể được bao bọc bởi nhiều wrapper cùng một lúc.
* Cho phép thêm hoặc xóa tính năng của một đối tượng lúc thực thi (run-time).

## Sử dụng Decorator Pattern khi nào?

* Khi muốn thêm tính năng mới cho các đối tượng mà không ảnh hưởng đến các đối tượng này.
* Khi không thể mở rộng một đối tượng bằng cách thừa kế (inheritance). Chẳng hạn, một class sử dụng từ khóa final, muốn mở rộng class này chỉ còn cách duy nhất là sử dụng decorator.
* Trong một số nhiều trường hợp mà việc sử dụng kế thừa sẽ mất nhiều công sức trong việc viết code. Ví dụ trên là một trong những trường hợp như vậy.

## So sánh Decorator và Adapter

Giống nhau:

* Cả hai đều là [structural pattern](https://gpcoder.com/tag/structuaral-pattern/) như định nghĩa của GOF.
* Cả hai sử dụng cách [composition](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/" \l "Cai_dat_Adapter_Pattern_nhu_the_nao) để cài đặt.

Khác nhau:

* Decorator cho phép thêm một tính năng mới vào một object nhưng không được phép sử dụng thừa kế. Nó cho phép thay đổi lúc thực thi (run-time). Adapter được sử dụng khi bạn có một interface, và bạn muốn ánh xạ interface đó đến một đối tượng khác có vai trò chức năng tương tự, nhưng là một interface khác.
* Decorator có xu hướng hoạt động trên một đối tượng. Adapter có xu hướng hoạt động trên nhiều đối tượng (có thể wrap nhiều interface).

Tài liệu tham khảo:

* <https://refactoring.guru/design-patterns/decorator>
* <https://sourcemaking.com/design_patterns/decorator>
* <https://www.javatpoint.com/decorator-pattern>
* <https://www.tutorialspoint.com/design_pattern/decorator_pattern.htm>
* <https://techblog.vn/design-pattern-decorator-pattern>

# Hướng dẫn Java Design Pattern – Facade

Đăng vào 16/11/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 2168 Lượt xem

Khi bạn làm việc với một số lượng lớn các đối tượng trong một hệ thống hay một thư viện phức tạp. Bạn phải tự khởi tạo tất cả các đối tượng này, theo dõi các thay đổi của nó, thứ tự logic của bạn phải xử lý chính xác với bên thứ ba, … Cuối cùng, logic nghiệp vụ các lớp của hệ thống bạn trở nên gắn kết chặt chẽ cùng với thư viện của bên thứ ba. Dẫn đến code của bạn trở nên khó hiểu, khó mở rộng và bảo trì.

Trong phần tiếp theo bài viết này chúng ta sẽ tìm hiểu Facade Pattern để thấy cách nó giải quyết đề này như thế nào.

Nội dung [[Ẩn](https://gpcoder.com/4604-huong-dan-java-design-pattern-facade/)]

* [1 Facade Pattern là gì?](https://gpcoder.com/4604-huong-dan-java-design-pattern-facade/" \l "Facade_Pattern_la_gi)
* [2 Cài đặt Facade Pattern như thế nào?](https://gpcoder.com/4604-huong-dan-java-design-pattern-facade/" \l "Cai_dat_Facade_Pattern_nhu_the_nao)
* [3 Lợi ích của Facade Pattern là gì?](https://gpcoder.com/4604-huong-dan-java-design-pattern-facade/" \l "Loi_ich_cua_Facade_Pattern_la_gi)
* [4 Sử dụng Facade Pattern khi nào?](https://gpcoder.com/4604-huong-dan-java-design-pattern-facade/" \l "Su_dung_Facade_Pattern_khi_nao)

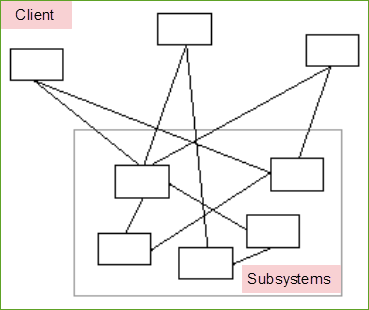
## Facade Pattern là gì?

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

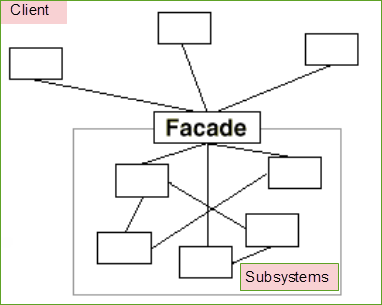
Facade Pattern là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Pattern này cung cấp một giao diện chung đơn giản thay cho một nhóm các giao diện có trong một hệ thống con (subsystem). Facade Pattern định nghĩa một giao diện ở một cấp độ cao hơn để giúp cho người dùng có thể dễ dàng sử dụng hệ thống con này.

Facade Pattern cho phép các đối tượng truy cập trực tiếp giao diện chung này để giao tiếp với các giao diện có trong hệ thống con. Mục tiêu là che giấu các hoạt động phức tạp bên trong hệ thống con, làm cho hệ thống con dễ sử dụng hơn.

Ví dụ khi không có Facade, hệ thống của chúng ta trông giống như sau:

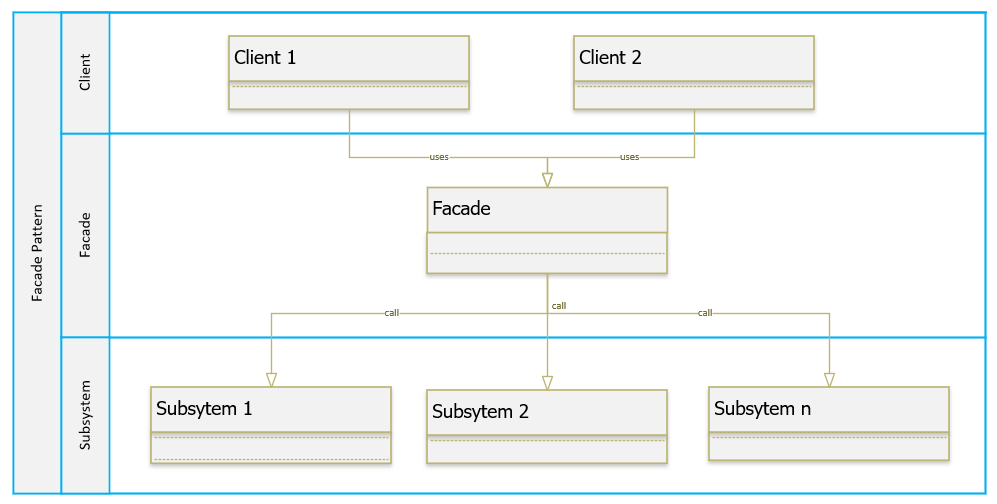


Khi có Facade, nó trông giống như sau:



Façade Pattern tương tự với [Adapter Pattern](https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/). Hai Pattern này làm việc theo cùng một cách, nhưng mục đích sử dụng của chúng khác nhau. Adapter Pattern chuyển đổi mã nguồn để làm việc được với mã nguồn khác. Nhưng Façade Pattern cho phép bao bọc mã nguồn gốc để nó có thể giao tiếp với mã nguồn khác dễ dàng hơn.

## Cài đặt Facade Pattern như thế nào?



Các thành phần cơ bản của một Facade Pattern:

* Facade: biết rõ lớp của hệ thống con nào đảm nhận việc đáp ứng yêu cầu của client, sẽ chuyển yêu cầu của client đến các đối tượng của hệ thống con tương ứng.
* Subsystems: cài đặt các chức năng của hệ thống con, xử lý công việc được gọi bởi Facade. Các lớp này không cần biết Facade và không tham chiếu đến nó.
* Client: đối tượng sử dụng Facade để tương tác với các subsystem.

Các đối tượng Facade thường là [Singleton](https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/)bởi vì chỉ cần duy nhất một đối tượng Facade.

Ví dụ: Một công ty bán hàng online, chẳng hạn Tiki cung cấp nhiều lựa chọn cho khách hàng khi mua sản phẩm. Khi một sản phẩm được mua, nó sẽ qua các bước xử lý: lấy thông tin về tài khoản mua hàng, thanh toán, vận chuyển, gửi Email/ SMS thông báo.

Ứng dụng của chúng ta được thiết kế với Facade Pattern, bao gồm các class như sau:

* Thông tin về tài khoản (AccountService) : lấy thông tin cơ bản của khách hàng thông qua email được cung cấp.
* Dịch vụ thanh toán (PaymentService) : có thể thanh toán thông qua Paypal, thẻ tín dụng (Credit Card), tài khoản ngân hàng trực tuyến (E-banking), Tiền mặt (cash).
* Dịch vụ vận chuyển (ShippingService) : có thể chọn Free Shipping, Standard Shipping, Express Shipping.
* Dịch vụ email (EmailService) : có thể gửi mail cho khách hàng về tình hình đặt hàng, thanh toán, vận chuyển, nhận hàng.
* Dịch vụ tin nhắn (SMS) : có thể gửi thông báo SMS cho khách hàng khi thanh toán online.
* ShopFacade : là một Facade Pattern, class này bao gồm các dịch vụ có bên trong hệ thống. Nó cung cấp một vài phương thức để Client có thể dễ dàng mua hàng. Tùy vào nghiệp vụ mà nó sẽ sử dụng những dịch tương ứng, chẳng hạn dịch vụ SMS chỉ được sử dụng nếu khách hàng đăng ký mua hàng thông qua hình thức thanh toán online (Paypal, E-banking, …).
* Client : là người dùng cuối sử dụng ShopFacade để mua hàng.



AccountService.java

package com.gpcoder.patterns.structural.facade;

public class AccountService {

public void getAccount(String email) {

System.out.println("Getting the account of " + email);

}

}

PaymentService.java

package com.gpcoder.patterns.structural.facade;

public class PaymentService {

public void paymentByPaypal() {

System.out.println("Payment by Paypal");

}

public void paymentByCreditCard() {

System.out.println("Payment by Credit Card");

}

public void paymentByEbankingAccount() {

System.out.println("Payment by E-banking account");

}

public void paymentByCash() {

System.out.println("Payment by cash");

}

}

ShippingService.java

package com.gpcoder.patterns.structural.facade;

public class ShippingService {

public void freeShipping() {

System.out.println("Free Shipping");

}

public void standardShipping() {

System.out.println("Standard Shipping");

}

public void expressShipping() {

System.out.println("Express Shipping");

}

}

EmailService.java

package com.gpcoder.patterns.structural.facade;

public class EmailService {

public void sendMail(String mailTo) {

System.out.println("Sending an email to " + mailTo);

}

}

SmsService.java

package com.gpcoder.patterns.structural.facade;

public class SmsService {

public void sendSMS(String mobilePhone) {

System.out.println("Sending an mesage to " + mobilePhone);

}

}

ShopFacade.java

package com.gpcoder.patterns.structural.facade;

public class ShopFacade {

private static final ShopFacade INSTANCE = new ShopFacade();

private AccountService accountService;

private PaymentService paymentService;

private ShippingService shippingService;

private EmailService emailService;

private SmsService smsService;

private ShopFacade() {

accountService = new AccountService();

paymentService = new PaymentService();

shippingService = new ShippingService();

emailService = new EmailService();

smsService = new SmsService();

}

public static ShopFacade getInstance() {

return INSTANCE;

}

public void buyProductByCashWithFreeShipping(String email) {

accountService.getAccount(email);

paymentService.paymentByCash();

shippingService.freeShipping();

emailService.sendMail(email);

System.out.println("Done\n");

}

public void buyProductByPaypalWithStandardShipping(String email, String mobilePhone) {

accountService.getAccount(email);

paymentService.paymentByPaypal();

shippingService.standardShipping();

emailService.sendMail(email);

smsService.sendSMS(mobilePhone);

System.out.println("Done\n");

}

}

Client.java

package com.gpcoder.patterns.structural.facade;

package com.gpcoder.patterns.structural.facade;

public class Client {

public static void main(String[] args) {

ShopFacade.getInstance().buyProductByCashWithFreeShipping("contact@gpcoder.com");

ShopFacade.getInstance().buyProductByPaypalWithStandardShipping("gpcodervn@gmail.com", "0988.999.999");

}

}

Output của chương trình trên:

Getting the account of contact@gpcoder.com

Payment by cash

Free Shipping

Sending an email to contact@gpcoder.com

Done

Getting the account of gpcodervn@gmail.com

Payment by Paypal

Standard Shipping

Sending an email to gpcodervn@gmail.com

Sending an mesage to 0988.999.999

Done

Như bạn thấy phía Client chỉ sử dụng một phương thức duy nhất là có thể mua được sản phẩm mặc dù bên dưới hệ thống có rất nhiều dịch vụ xử lý khác nhau. Nếu không có Facade, phía Client sẽ không biết sử dụng những dịch vụ nào để có thể mua được sản phẩm. Khi phát sinh thêm một dịch vụ sẽ rất khó khăn khi sửa đổi và code phía Client cũng sẽ bị ảnh hưởng.

## Lợi ích của Facade Pattern là gì?

* Giúp cho hệ thống của bạn trở nên đơn giản hơn trong việc sử dụng và trong việc hiểu nó, vì một mẫu Facade có các phương thức tiện lợi cho các tác vụ chung.
* Giảm sự phụ thuộc của các mã code bên ngoài với hiện thực bên trong của thư viện, vì hầu hết các code đều dùng Facade, vì thế cho phép sự linh động trong phát triển các hệ thống.
* Đóng gói tập nhiều hàm API được thiết kế không tốt bằng một hàm API đơn có thiết kế tốt hơn.

## Sử dụng Facade Pattern khi nào?

* Khi hệ thống có rất nhiều lớp làm người sử dụng rất khó để có thể hiểu được quy trình xử lý của chương trình. Và khi có rất nhiều hệ thống con mà mỗi hệ thống con đó lại có những giao diện riêng lẻ của nó nên rất khó cho việc sử dụng phối hợp. Khi đó có thể sử dụng Facade Pattern để tạo ra một giao diện đơn giản cho người sử dụng một hệ thống phức tạp.
* Khi người sử dụng phụ thuộc nhiều vào các lớp cài đặt. Việc áp dụng Façade Pattern sẽ tách biệt hệ thống con của người dùng và các hệ thống con khác, do đó tăng khả năng độc lập và khả chuyển của hệ thống con, dễ chuyển đổi nâng cấp trong tương lai.
* Khi bạn muốn phân lớp các hệ thống con. Dùng Façade Pattern để định nghĩa cổng giao tiếp chung cho mỗi hệ thống con, do đó giúp giảm sự phụ thuộc của các hệ thống con vì các hệ thống này chỉ giao tiếp với nhau thông qua các cổng giao diện chung đó.
* Khi bạn muốn bao bọc, che giấu tính phức tạp trong các hệ thống con đối với phía Client.

Tài liệu tham khảo:

* <https://refactoring.guru/design-patterns/facade>
* <https://sourcemaking.com/design_patterns/facade>
* <https://www.javatpoint.com/facade-pattern>
* <https://www.tutorialspoint.com/design_pattern/facade_pattern.htm>
* <https://www.javagists.com/facade-design-pattern>
* <http://aptech.fpt.edu.vn/chitiet.php?id=412>
* Design Patterns: Elements of Reusable Object-Oriented Software – GOF

# Hướng dẫn Java Design Pattern – Flyweight

Đăng vào 22/11/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 2580 Lượt xem

Trong một số tình huống trong phát triển phần mềm, chúng ta có thể cải thiện hiệu suất của ứng dụng với việc sử dụng Cache. Hãy tưởng tượng rất nhiều đối tượng được tạo ra và lãng phí bộ nhớ. Mô hình Flyweight được tạo ra để tránh vấn đề này và tối ưu hóa hiệu suất.

Nội dung [[Ẩn](https://gpcoder.com/4626-huong-dan-java-design-pattern-flyweight/)]

* [1 Flyweight Pattern là gì?](https://gpcoder.com/4626-huong-dan-java-design-pattern-flyweight/" \l "Flyweight_Pattern_la_gi)
* [2 Hai trạng thái của Flyweight Object](https://gpcoder.com/4626-huong-dan-java-design-pattern-flyweight/" \l "Hai_trang_thai_cuaFlyweight_Object)
* [3 Cài đặt Flyweight Pattern như thế nào?](https://gpcoder.com/4626-huong-dan-java-design-pattern-flyweight/" \l "Cai_dat_Flyweight_Pattern_nhu_the_nao)
* [4 Lợi ích của Flyweight Pattern là gì?](https://gpcoder.com/4626-huong-dan-java-design-pattern-flyweight/" \l "Loi_ich_cua_Flyweight_Pattern_la_gi)
* [5 Sử dụng Flyweight Pattern khi nào?](https://gpcoder.com/4626-huong-dan-java-design-pattern-flyweight/" \l "Su_dung_Flyweight_Pattern_khi_nao)

## Flyweight Pattern là gì?

Use sharing to support large numbers of fine-grained objects efficiently.

Flyweight Pattern là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Nó cho phép tái sử dụng đối tượng tương tự đã tồn tại bằng cách lưu trữ chúng hoặc tạo đối tượng mới khi không tìm thấy đối tượng phù hợp.

Flyweight Pattern được sử dụng khi chúng ta cần tạo một số lượng lớn các đối tượng của 1 lớp nào đó. Do mỗi đối tượng đều đòi hỏi chiếm giữ một khoảng không gian bộ nhớ, nên với một số lượng lớn đối tượng được tạo ra có thể gây nên vấn đề nghiêm trọng đặc biệt đối với các thiết bị có dung lượng nhớ thấp. Flyweight Pattern có thể được áp dụng để giảm tải cho bộ nhớ thông qua cách chia sẻ các đối tượng. Vì vậy performance của hệ thống được tối ưu.

Flyweight object là immutable, nghĩa là không thể thay đổi khi nó đã được khởi tạo.

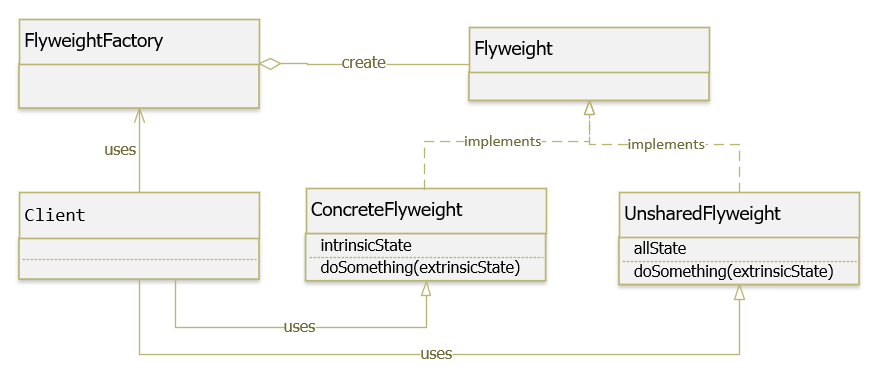
## Hai trạng thái của Flyweight Object

Trạng thái của flyweight object là một phần quan trọng trong việc thiết kế Flyweight Pattern. Mục tiêu chính của Flyweight Pattern là giảm bộ nhớ bằng cách chia sẽ các đối tượng. Điều này có thể đạt được bằng cách tách các thuộc tính của đối tượng thành hai trạng thái: độc lập và phụ thuộc. Hay còn gọi là Intrinsic (trạng thái nội tại) và Extrinsic (trạng thái bên ngoài).

* Intrinsic State (trạng thái nội tại) : Trạng thái này chứa dữ liệu không thể thay đổi (unchangeable) và không phụ thuộc (independent) vào ngữ cảnh (context) của đối tượng Flyweight . Những dữ liệu đó có thể được lưu trữ vĩnh viễn bên trong đối tượng Flyweight. Vì vậy mà Flyweight object có thể chia sẻ. Dữ liệu nội tại là phi trạng thái (stateless) và thường không thay đổi (unchanged). Tính năng này cho phép khả năng tái tạo các thuộc tính đối tượng Flyweight giữa các đối tượng tương tự khác. Điều quan trọng cần lưu ý là các đối tượng Flyweight chỉ nên nhận trạng thái bên trong của chúng thông qua các tham số của hàm tạo và không cung cấp các phương thức setter hay các biến public.
* Extrinsic State (trạng thái bên ngoài) : Trạng thái bên ngoài thể hiện tính chất phụ thuộc ngữ cảnh của đối tượng flyweight. Trạng thái này chứa các thuộc tính và dữ liệu được áp dụng hoặc được tính toán trong thời gian thực thi (runtime). Do đó, những dữ liệu đó không được lưu trữ trong bộ nhớ. Vì trạng thái bên ngoài là phụ thuộc ngữ cảnh và có thể thay đổi nên các đối tượng đó không thể được chia sẻ. Do đó, client chịu trách nhiệm truyền dữ liệu liên quan đến trạng thái bên ngoài cho đối tượng flyweight khi cần thiết, có thể thông qua các tham số (argument).

Tóm lại, tốt nhất là xem xét cách mỗi dữ liệu hoạt động bên trong đối tượng khi tạo đối tượng flyweight. Nghĩa là thành phần nào không được thay đổi, cần lưu trữ, thành phần nào không được thay đổi, không cần lưu trữ. Điều này giúp cho chúng ta tiết kiệm được bộ nhớ và thực thi được hiệu quả.

## Cài đặt Flyweight Pattern như thế nào?



Các thành phần trong mẫu thiết kế Flyweight:

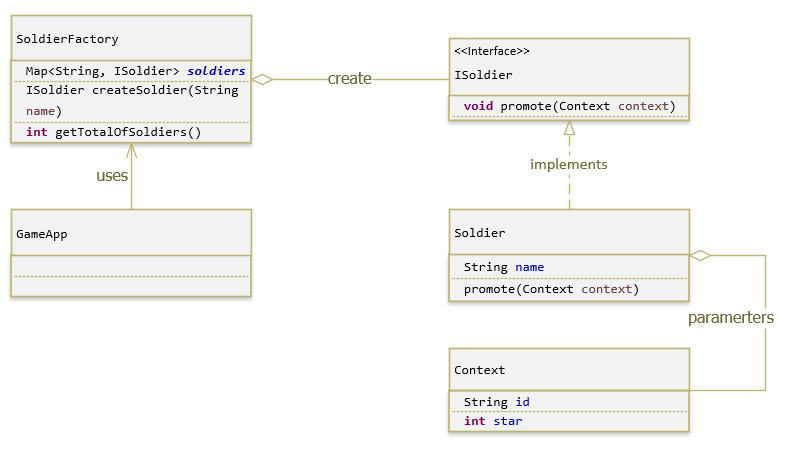
* Flyweight : là một interface/ abstract class, định nghĩa các các thành phần của một đối tượng.
* ConcreteFlyweight : triển khai các phương thức đã được định nghĩa trong Flyweight. Việc triển khai này phải thực hiện các khả năng của trạng thái nội tại. Đó là dữ liệu phải không thể thay đổi (unchangeable) và có thể chia sẻ (shareable). Các đối tượng là phi trạng thái (stateless) trong triển khai này. Vì vậy, đối tượng ConcreteFlyweight giống nhau có thể được sử dụng trong các ngữ cảnh khác nhau.
* UnsharedFlyweight : mặc dù mẫu thiết kế Flyweight cho phép chia sẻ thông tin, nhưng có thể tạo ra các thể hiện không được chia sẻ (not shared). Trong những trường hợp này, thông tin của các đối tượng có thể là stateful.
* FlyweightFactory (Cache): lớp này có thể là một Factory Pattern được sử dụng để giữ tham chiếu đến đối tượng Flyweight đã được tạo ra. Nó cung cấp một phương thức để truy cập đối tượng Flyweight được chia sẽ. FlyweightFactory bao gồm một Pool (có thể là HashMap, không cho phép bên ngoài truy cập vào) để lưu trữ đối tượng Flyweight trong bộ nhớ. Nó sẽ trả về đối tượng Flyweight đã tồn tại khi được yêu cầu từ Client hoặc tạo mới nếu không tồn tại.
* Client : sử dụng FlyweightFactory để khởi tạo đối tượng Flyweight.

Ví dụ:

Một ứng dụng game bao gồm rất nhiều Solider (lính), được chia thành các loại: Yuri, Spy, Doctor, … Mỗi Solider sẽ có id và cấp độ khác nhau. Thời gian để tạo một loại Solider là 3 giây.

Chương trình chúng ta được cài đặt với Flyweigth Pattern như sau:

* ISoldier : đóng vai trò là một Flyweight. Định nghĩa phương thức promote với tham số là context, giá trị tham số này được quyết định bởi Client.
* Context : đóng vai trò là Extrinsic State. Tùy vào ngữ cảnh khác nhau sẽ có giá trị khác nhau.
* Soldier : đóng vai trò là ConcreteFlyweight. Triển khai các phương thức đã được định nghĩa trong Flyweight. Nó có một giá trị name (ứng với loại lính – Intrinsic State) không thay đổi, có thể chia sẻ được.
* SoldierFactory : đóng vai trò là FlyweightFactory. Nó giữ một private Map để lưu giữ các loại soldier đã được tạo. Một phương thức createSoldier() để tạo soldier ứng với tên loại được truyền vào. Một phương thức getTotalOfSoldiers() để kiểm tra số lượng các loại soldier đã tạo.
* GameApp : đóng vai trò là Client. Sử dụng SoldierFactory để tạo các Soldier.



ISoldier.java

package com.gpcoder.patterns.structural.flyweight;

/\*\*

\* Flyweight

\*/

public interface ISoldier {

void promote(Context context);

}

Context.java

package com.gpcoder.patterns.structural.flyweight;

/\*\*

\* Extrinsic State

\*/

public class Context {

private String id;

private int star;

public Context(String id, int star) {

this.id = id;

this.star = star;

}

public String getId() {

return this.id;

}

public int getStar() {

return this.star;

}

}

Soldier.java

package com.gpcoder.patterns.structural.flyweight;

/\*\*

\* Concrete Flyweight

\*/

public class Soldier implements ISoldier {

private final String name; // Intrinsic State

public Soldier(String name) {

this.name = name;

System.out.println("Soldier is created! - " + name);

}

@Override

public void promote(Context context) {

System.out.println(name + " " + context.getId() + " promoted " + context.getStar());

}

}

SoldierFactory.java

package com.gpcoder.patterns.structural.flyweight;

import java.util.HashMap;

import java.util.Map;

/\*\*

\* FlyweightFactory

\*/

public class SoldierFactory {

private static final Map<String, ISoldier> soldiers = new HashMap<>();

private SoldierFactory() {

throw new IllegalStateException();

}

public static synchronized ISoldier createSoldier(String name) {

ISoldier soldier = soldiers.get(name);

if (soldier == null) {

waitingForCreateASoldier();

soldier = new Soldier(name);

soldiers.put(name, soldier);

}

return soldier;

}

public static synchronized int getTotalOfSoldiers() {

return soldiers.size();

}

private static void waitingForCreateASoldier() {

try {

Thread.sleep(3000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

GameApp.java

package com.gpcoder.patterns.structural.flyweight;

import java.time.Duration;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* Client

\*/

public class GameApp {

private static List<ISoldier> soldiers = new ArrayList<>();

public static void main(String[] args) {

long startTime = System.currentTimeMillis();

createSoldier(5, "Yuri", 1);

createSoldier(5, "Spy", 1);

createSoldier(3, "Spy", 3);

createSoldier(2, "Yuri", 2);

long endTime = System.currentTimeMillis();

System.out.println("---");

System.out.println("Total soldiers made : " + soldiers.size());

System.out.println("Total time worked : " + Duration.ofMillis(endTime - startTime).getSeconds() + " seconds");

System.out.println("Total type of soldiers made : " + SoldierFactory.getTotalOfSoldiers());

}

private static void createSoldier(int numberOfSoldier, String soldierName, int numberOfStar) {

for (int i = 1; i <= numberOfSoldier; i++) {

Context star = new Context("Soldier" + (soldiers.size() + 1), numberOfStar);

ISoldier soldier = SoldierFactory.createSoldier(soldierName);

soldier.promote(star);

soldiers.add(soldier);

}

}

}

Output của chương trình trên như sau:

Soldier is created! - Yuri

Yuri Soldier1 promoted 1

Yuri Soldier2 promoted 1

Yuri Soldier3 promoted 1

Yuri Soldier4 promoted 1

Yuri Soldier5 promoted 1

Soldier is created! - Spy

Spy Soldier6 promoted 1

Spy Soldier7 promoted 1

Spy Soldier8 promoted 1

Spy Soldier9 promoted 1

Spy Soldier10 promoted 1

Spy Soldier11 promoted 3

Spy Soldier12 promoted 3

Spy Soldier13 promoted 3

Yuri Soldier14 promoted 2

Yuri Soldier15 promoted 2

---

Total soldiers made : 15

Total time worked : 6 seconds

Total type of soldiers made : 2

Như bạn thấy, chúng ta đã tạo ra 15 soldier chỉ với 6 giây (do chỉ tạo 2 lần ứng với 2 loại Yuri và Spy, mỗi loại mất 3 giây). Nếu không sử dụng Flyweight mà tạo từng soldier riêng lẻ, chúng ta sẽ mất đến 45 giây (do mỗi soldier mất 3 giây và có tất cả 15 soldier).

## Lợi ích của Flyweight Pattern là gì?

* Giảm số lượng đối tượng được tạo ra bằng cách chia sẻ đối tượng. Vì vậy, tiết kiệm bộ nhớ và các thiết bị lưu trữ cần thiết.
* Cãi thiện khả năng cache dữ liệu vì thời gian đáp ứng nhanh.
* Tăng performance.

## Sử dụng Flyweight Pattern khi nào?

* Khi có một số lớn các đối tượng được ứng dụng tạo ra một cách lặp đi lặp lại.
* Khi việc tạo ra đối tượng đòi hỏi nhiều bộ nhớ và thời gian.
* Khi muốn tái sử dụng đối tượng đã tồn tại thay vì phải tốn thời gian để tạo mới.
* Khi nhóm đối tượng chứa nhiều đối tượng tương tự và hai đối tượng trong nhóm không khác nhau nhiều.

Tài liệu tham khảo:

* <https://refactoring.guru/design-patterns/flyweight>
* <https://sourcemaking.com/design_patterns/flyweight>
* <https://www.tutorialspoint.com/design_pattern/flyweight_pattern.htm>
* Design Patterns: Elements of Reusable Object-Oriented Software – GOF

# Hướng dẫn Java Design Pattern – Proxy

Đăng vào 30/11/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 2221 Lượt xem

Khi cần điều khiển truy nhập tới một đối tượng được thực hiện từ quá trình khởi tạo nó cho tới khi thực sự cần sử dụng nó. Hoặc cần bảo vệ quyền truy xuất vào các phương thức của object thực. Trong trường hợp như vậy, ta nên dùng mẫu thiết kế Proxy.

Nội dung [[Ẩn](https://gpcoder.com/4644-huong-dan-java-design-pattern-proxy/)]

* [1 Proxy Pattern là gì?](https://gpcoder.com/4644-huong-dan-java-design-pattern-proxy/" \l "Proxy_Pattern_la_gi)
* [2 Phân loại Proxy](https://gpcoder.com/4644-huong-dan-java-design-pattern-proxy/" \l "Phan_loai_Proxy)
* [3 Cài đặt Proxy Pattern như thế nào?](https://gpcoder.com/4644-huong-dan-java-design-pattern-proxy/" \l "Cai_dat_Proxy_Pattern_nhu_the_nao)
  + [3.1 Ví dụ Virtual Proxy](https://gpcoder.com/4644-huong-dan-java-design-pattern-proxy/" \l "Vi_du_Virtual_Proxy)
  + [3.2 Ví dụ Protection proxy](https://gpcoder.com/4644-huong-dan-java-design-pattern-proxy/" \l "Vi_du_Protection_proxy)
  + [3.3 Ví dụ Remote Proxy](https://gpcoder.com/4644-huong-dan-java-design-pattern-proxy/" \l "Vi_du_Remote_Proxy)
  + [3.4 Ví dụ Smart Proxy](https://gpcoder.com/4644-huong-dan-java-design-pattern-proxy/" \l "Vi_du_Smart_Proxy)
* [4 Lợi ích của Proxy Pattern là gì?](https://gpcoder.com/4644-huong-dan-java-design-pattern-proxy/" \l "Loi_ich_cua_Proxy_Pattern_la_gi)
* [5 Sử dụng Proxy Pattern khi nào?](https://gpcoder.com/4644-huong-dan-java-design-pattern-proxy/" \l "Su_dung_Proxy_Pattern_khi_nao)
* [6 So sánh Proxy Pattern với Decorator Pattern](https://gpcoder.com/4644-huong-dan-java-design-pattern-proxy/" \l "So_sanh_Proxy_Pattern_voi_Decorator_Pattern)

## Proxy Pattern là gì?

Provide a surrogate or placeholder for another object to control access to it.

Proxy Pattern là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern).

Proxy có nghĩa là “ủy quyền” hay “đại diện”. Mục đích xây dựng Proxy pattern cũng chính vì muốn tạo ra một đối tượng sẽ ủy quyền, thay thế cho một đối tượng khác.

Proxy Pattern là mẫu thiết kế mà ở đó tất cả các truy cập trực tiếp đến một đối tượng nào đó sẽ được chuyển hướng vào một đối tượng trung gian (Proxy Class). Mẫu Proxy (người đại diện) đại diện cho một đối tượng khác thực thi các phương thức, phương thức đó có thể được định nghĩa lại cho phù hợp với múc đích sử dụng.

Để đơn giản hơn bạn có thể nghĩ đến khái niệm HTTP proxy trong mạng máy tính, nó là một gateway giữa trình duyệt (client) và máy chủ (subject). HTTP proxy giúp nâng cao trải nghiệm người dùng, tăng tốc với lưu đệm các dữ liệu, loại bỏ các trang quảng cáo, giới hạn các vùng thông tin được xem… Proxy Pattern cũng có chung một mục đích như với HTTP proxy.

Proxy Pattern còn được gọi là Surrogate (thay thế) hoặc Placeholder (trình giữ chỗ).

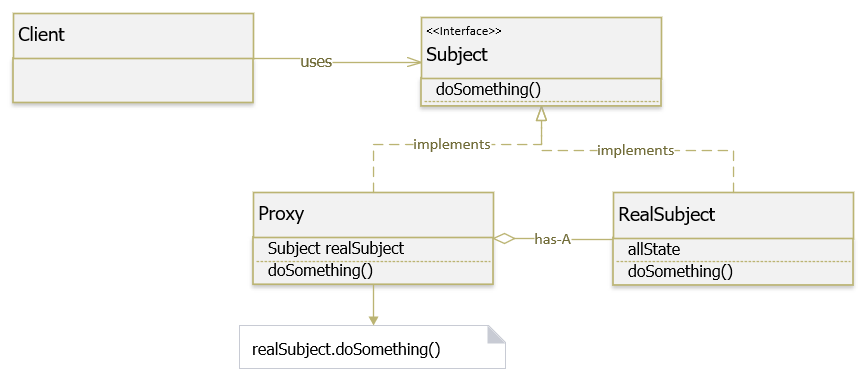
## Phân loại Proxy

* Virtual Proxy : Virtual Proxy tạo ra một đối tượng trung gian mỗi khi có yêu cầu tại thời điểm thực thi ứng dụng, nhờ đó làm tăng hiệu suất của ứng dụng.
* Protection Proxy : Phạm vi truy cập của các client khác nhau sẽ khác nhau. Protection proxy sẽ kiểm tra các quyền truy cập của client khi có một dịch vụ được yêu cầu.
* Remote Proxy : Client truy cập qua Remote Proxy để chiếu tới một đối tượng được bảo về nằm bên ngoài ứng dụng (trên cùng máy hoặc máy khác).
* Monitor Proxy : Monitor Proxy sẽ thiết lập các bảo mật trên đối tượng cần bảo vệ, ngăn không cho client truy cập một số trường quan trọng của đối tượng. Có thể theo dõi, giám sát, ghi log việc truy cập, sử dụng đối tượng.
* Firewall Proxy : bảo vệ đối tượng từ chối các yêu cầu xuất xứ từ các client không tín nhiệm.
* Cache Proxy : Cung cấp không gian lưu trữ tạm thời cho các kết quả trả về từ đối tượng nào đó, kết quả này sẽ được tái sử dụng cho các client chia sẻ chung một yêu cầu gửi đến. Loại Proxy này hoạt động tương tự như Flyweight Pattern.
* Smart Reference Proxy : Là nơi kiểm soát các hoạt động bổ sung mỗi khi đối tượng được tham chiếu.
* Synchronization Proxy : Đảm bảo nhiều client có thể truy cập vào cùng một đối tượng mà không gây ra xung đột. Khi một client nào đó chiếm dụng khóa khá lâu khiến cho số lượng các client trong danh sách hàng đợi cứ tăng lên, và do đó hoạt động của hệ thống bị ngừng trệ, có thể dẫn đến hiện tượng “tắc nghẽn”.
* Copy-On-Write Proxy : Loại này đảm bảo rằng sẽ không có client nào phải chờ vô thời hạn. Copy-On-Write Proxy là một thiết kế rất phức tạp.

## Cài đặt Proxy Pattern như thế nào?

Proxy Pattern có những đặc điểm chung sau đây:

* Cung cấp mức truy cập gián tiếp vào một đối tượng.
* Tham chiếu vào đối tượng đích và chuyển tiếp các yêu cầu đến đối tượng đó.
* Cả Proxy và đối tượng đích đều kế thừa hoặc thực thi chung một lớp giao diện. Mã máy dịch cho lớp giao diện thường “nhẹ” hơn các lớp cụ thể và do đó có thể giảm được thời gian tải dữ liệu giữa server và client.



Các thành phần tham gia vào mẫu Proxy Pattern:

* Subject : là một interface định nghĩa các phương thực để giao tiếp với client. Đối tượng này xác định giao diện chung cho RealSubject và Proxy để Proxy có thể được sử dụng bất cứ nơi nào mà RealSubject mong đợi.
* Proxy : là một class sẽ thực hiện các bước kiểm tra và gọi tới đối tượng của class service thật để thực hiện các thao tác sau khi kiểm tra. Nó duy trì một tham chiếu đến RealSubject để Proxy có thể truy cập nó. Nó cũng thực hiện các giao diện tương tự như RealSubject để Proxy có thể được sử dụng thay cho RealSubject. Proxy cũng điều khiển truy cập vào RealSubject và có thể tạo hoặc xóa đối tượng này.
* RealSubject : là một class service sẽ thực hiện các thao tác thực sự. Đây là đối tượng chính mà proxy đại diện.
* Client : Đối tượng cần sử dụng RealSubject nhưng thông qua Proxy.

### Ví dụ Virtual Proxy

Trì hoãn việc tạo ra real subject bên trong proxy class. Chỉ đến khi cần, proxy class mới thật sự khởi tạo real class. Loại Proxy này làm việc theo cơ chế Lazy Loading.

Lazy Loading là một khái niệm mà ứng dụng trì hoãn việc tải các đối tượng cho đến thời điểm mà người dùng cần nó. Nói một cách đơn giản là tải theo yêu cầu của người dùng chứ không phải tải đối tượng không cần thiết. Lợi ích của việc này là giảm thiểu số lượng yêu cầu, giảm thiểu số lượng tài nguyên thừa cần tải cho tới khi người dùng cần đến chúng thực sự. Chúng ta đã thấy điều này qua cơ chế Lazy load của [Hibernate](http://hibernate.org/).

Nó giải quyết vấn đề rất lớn về hiệu suất, lý do là vì proxy class có chi phí khởi tạo rất ít, việc duy trì nó không mất nhiều tài nguyên hệ thống. Trong khi đó Real class thường rất tốn chi phí , vì thế với virtual proxies, chỉ khi nào cần thiết, real class mới được khởi tạo.

Ví dụ một website hiển thị ảnh, có thể có rất nhều ảnh trên một trang hay một ảnh được hiển thị nhiều lần. Trường hợp này chúng ta chỉ cần load ảnh khi nó cần hiển thị (khi ta scroll tới nơi đặt image) hoặc là nó chưa được load (không như các website truyền thống là load hình ngay khi load website, nó rất tốn tài nguyên và đôi khi không cần thiết do người dùng không scroll tới nơi đặt image, có thể load 1 ảnh nhiều lần).

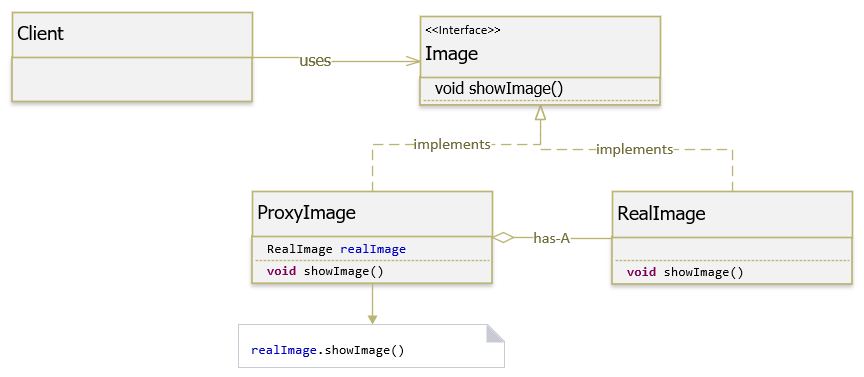


Image.java

package com.gpcoder.patterns.structural.proxy.virtual;

public interface Image {

void showImage();

}

RealImage.java

package com.gpcoder.patterns.structural.proxy.virtual;

public class RealImage implements Image {

private String url;

public RealImage(String url) {

this.url = url;

System.out.println("Image loaded: " + this.url);

}

@Override

public void showImage() {

System.out.println("Image showed: " + this.url);

}

}

ProxyImage.java

package com.gpcoder.patterns.structural.proxy.virtual;

public class ProxyImage implements Image {

private Image realImage;

private String url;

public ProxyImage(String url) {

this.url = url;

System.out.println("Image unloaded: " + this.url);

}

@Override

public void showImage() {

if (realImage == null) {

realImage = new RealImage(this.url);

} else {

System.out.println("Image already existed: " + this.url);

}

realImage.showImage();

}

}

Client.java

package com.gpcoder.patterns.structural.proxy.virtual;

public class Client {

public static void main(String[] args) {

System.out.println("Init proxy image: ");

ProxyImage proxyImage = new ProxyImage("https://gpcoder.com/favicon.ico");

System.out.println("---");

System.out.println("Call real service 1st: ");

proxyImage.showImage();

System.out.println("---");

System.out.println("Call real service 2nd: ");

proxyImage.showImage();

}

}

Output của chương trình:

Init proxy image:

Image unloaded: https://gpcoder.com/favicon.ico

---

Call real service 1st:

Image loaded: https://gpcoder.com/favicon.ico

Image showed: https://gpcoder.com/favicon.ico

---

Call real service 2nd:

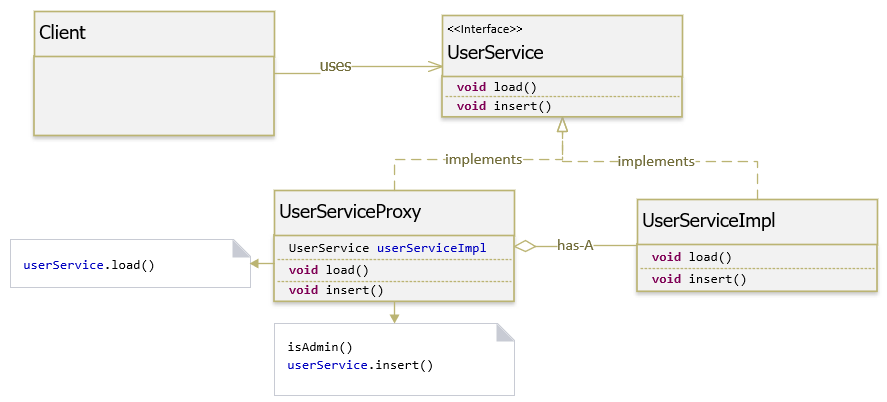
Image already existed: https://gpcoder.com/favicon.ico

Image showed: https://gpcoder.com/favicon.ico

Như bạn thấy image chỉ thật sử được load lên khi proxy class gọi hàm.

### Ví dụ Protection proxy

Yêu cầu người gọi proxy class phải chứng thực trước khi proxy class truy xuất vào real class. Cái này rất hữu dụng khi bạn bạn viết library cho một bên khác sử dụng và yêu cầu họ xác thực trước khi gọi hàm.



UserService.java

package com.gpcoder.patterns.structural.proxy.protection;

public interface UserService {

void load();

void insert();

}

UserServiceImpl.java

package com.gpcoder.patterns.structural.proxy.protection;

public class UserServiceImpl implements UserService {

private String name;

public UserServiceImpl(String name) {

this.name = name;

}

@Override

public void load() {

System.out.println(name + " loaded");

}

@Override

public void insert() {

System.out.println(name + " inserted");

}

}

UserServiceProxy.java

package com.gpcoder.patterns.structural.proxy.protection;

public class UserServiceProxy implements UserService {

private String role;

private UserService userService;

public UserServiceProxy(String name, String role) {

this.role = role;

userService = new UserServiceImpl(name);

}

@Override

public void load() {

userService.load();

}

@Override

public void insert() {

if (isAdmin()) {

userService.insert();

} else {

throw new IllegalAccessError("Access denied");

}

}

private boolean isAdmin() {

return "admin".equalsIgnoreCase(this.role);

}

}

Client.java

package com.gpcoder.patterns.structural.proxy.protection;

public class Client {

public static void main(String[] args) {

UserService admin = new UserServiceProxy("gpcoder", "admin");

admin.load();

admin.insert();

UserService customer = new UserServiceProxy("customer", "guest");

customer.load();

customer.insert();

}

}

Output của chương trình:

gpcoder loaded

gpcoder inserted

customer loaded

Exception in thread "main"

java.lang.IllegalAccessError: Access denied

at com.gpcoder.patterns.structural.proxy.rotection.UserServiceProxy.insert(UserServiceProxy.java:23)

at com.gpcoder.patterns.structural.proxy.rotection.Client.main(Client.java:11)

Các bạn thấy khi Client muốn gọi hàm insert(), trong Proxy luôn xác thực quyền của user trước khi thực hiện nó.

### Ví dụ Remote Proxy

Với Remote Proxy, proxy class và real class nằm ở 2 địa chỉ khác nhau. Thông qua network, proxy class sẽ encode và gửi request tới real class để khởi tạo, truy xuất, …

Chúng ta có thể thấy remote Proxy khi implements Java RMI hoặc thông dụng nhất là ở WebService. Bên phía client sẽ có 1 proxy class, client sẽ khởi tạo proxy class và gọi tới real class nằm ở 1 địa chỉ khác.

### Ví dụ Smart Proxy

Proxy Class sẽ thay đổi hoặc thêm 1 số thao tác trước khi gọi tới real class. Một số trường hợp thường thấy là:

* Ở lần đâu tiên khởi tạo real class, Proxy class sẽ lưu lại thông tin của real class vào cache và hữu dụng cho lần tái sử dụng sau. Ta có thể thấy việc xuất hiện ở lazy-load ở các connection xuống database.
* Trước khi gọi real class, ta có thể lock real class lại và không cho các thread khác phải chờ cho tới khi thread hiện tại release real class.
* Đếm số lượng reference tới real class.

Ở Virtual Proxy, mình cũng có lồng 1 ví dụ của Smart Proxy vào. Khi ta khởi tạo RealImage, mình đã lưu cache RealImage lại và lần sau gọi hàm ShowImage(), ta không phải khởi tạo lại RealImage.

## Lợi ích của Proxy Pattern là gì?

* Cãi thiện Performance thông qua lazy loading, chỉ tải các tài nguyên khi chúng được yêu cầu.
* Nó cung cấp sự bảo vệ cho đối tượng thực từ thế giới bên ngoài.
* Giảm chi phí khi có nhiều truy cập vào đối tượng có chi phí khởi tạo ban đầu lớn.
* Dễ nâng cấp, bảo trì.

## Sử dụng Proxy Pattern khi nào?

* Khi muốn bảo vệ quyền truy xuất vào các phương thức của object thực.
* Khi cần một số thao tác bổ sung trước khi thực hiện phương thức của object thực.
* Khi tạo đối tượng ban đầu là theo yêu cầu hoặc hệ thống yêu cầu sự chậm trễ khi tải một số tài nguyên nhất định (lazy loading).
* Khi có nhiều truy cập vào đối tượng có chi phí khởi tạo ban đầu lớn.
* Khi đối tượng gốc tồn tại trong môi trường từ xa (remote).
* Khi đối tượng gốc nằm trong một hệ thống cũ hoặc thư viện của bên thứ ba.
* Khi muốn theo dõi trạng thái và vòng đời đối tượng.

## So sánh Proxy Pattern với Decorator Pattern

Cấu trúc của Proxy Pattern và Decorator Pattern là tương tự nhau (bạn có thể coi ở class diagram). Hai Pattern này đều Wrap một đối tượng thực bên trong nó. Tuy nhiên, khác nhau thật sự giữa Proxy Pattern và Decorator Pattern nằm ở mục đích sử dụng. Với Decorator Pattern, người sử dụng sẽ nhắm tới mục tiêu là có thể thêm tính năng động vào một đối tượng có trước, trong khi đó Proxy Pattern cho phép ta tạo ra một đại diện cho một đối tượng khác.

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/proxy>
* <https://refactoring.guru/design-patterns/proxy>
* <https://www.javatpoint.com/proxy-pattern>
* <https://www.tutorialspoint.com/design_pattern/proxy_pattern.htm>
* <https://appstechviet.wordpress.com/2016/08/23/design-pattern-the-proxy-pattern/>
* Design Patterns: Elements of Reusable Object-Oriented Software – GOF

# Hướng dẫn Java Design Pattern – Chain of Responsibility

Đăng vào 03/12/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 2044 Lượt xem

Trong các viết trước chúng ta đã cùng tìm hiểu về các Design Pattern thuộc nhóm cấu trúc ([Structuaral Pattern](https://gpcoder.com/tag/structuaral-pattern/)). Trong bài viết hôm nay chúng ta sẽ cùng tìm hiểu một Pattern thuộc nhóm hành vi (Behavior Pattern) là Chain of Responsibility.

Toàn bộ các  mẫu hành vi (Behavior Pattern) xoay quanh nguyên tắc “thành phần đối tượng (composition) hơn là thừa kế (inheritance)”. Nguyên tắc này nói rằng, thay vì mở rộng từ một lớp hiện có, thiết kế lớp của bạn để tham chiếu một lớp hiện có mà bạn muốn sử dụng. Trong Java, nó được thực hiện bằng cách khai báo một biến tham chiếu đối tượng của lớp hiện có. Sau đó, bằng cách khởi tạo đối tượng thông qua hàm tạo (constructor) hoặc phương thức (setter), và cuối cùng sử dụng đối tượng được tham chiếu.

Nội dung [[Ẩn](https://gpcoder.com/4665-huong-dan-java-design-pattern-chain-of-responsibility/)]

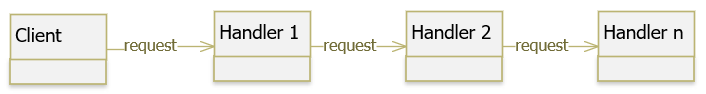
* [1 Chain of Responsibility Pattern là gì?](https://gpcoder.com/4665-huong-dan-java-design-pattern-chain-of-responsibility/" \l "Chain_of_Responsibility_Pattern_la_gi)
* [2 Cài đặt Chain of Responsibility Pattern như thế nào?](https://gpcoder.com/4665-huong-dan-java-design-pattern-chain-of-responsibility/" \l "Cai_dat_Chain_of_Responsibility_Pattern_nhu_the_nao)
  + [2.1 Ví dụ Chain of Responsibility Pattern với ứng dụng Logger](https://gpcoder.com/4665-huong-dan-java-design-pattern-chain-of-responsibility/" \l "Vi_du_Chain_of_Responsibility_Pattern_voi_ung_dung_Logger)
  + [2.2 Ví dụ Chain of Responsibility Pattern với ứng dụng LeaveRequest](https://gpcoder.com/4665-huong-dan-java-design-pattern-chain-of-responsibility/" \l "Vi_du_Chain_of_Responsibility_Pattern_voi_ung_dung_LeaveRequest)
* [3 Lợi ích của Chain of Responsibility Pattern là gì?](https://gpcoder.com/4665-huong-dan-java-design-pattern-chain-of-responsibility/" \l "Loi_ich_cua_Chain_of_Responsibility_Pattern_la_gi)
* [4 Sử dụng Chain of Responsibility Pattern khi nào?](https://gpcoder.com/4665-huong-dan-java-design-pattern-chain-of-responsibility/" \l "Su_dung_Chain_of_Responsibility_Pattern_khi_nao)

## Chain of Responsibility Pattern là gì?

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along with the chain until an object handles it.

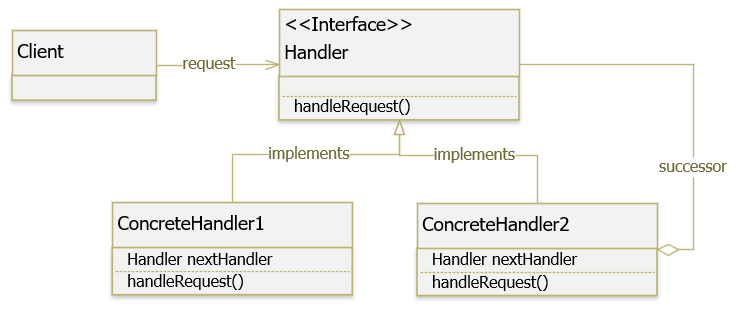
Chain of Responsibility (COR) là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern).

Chain of Responsiblity cho phép một đối tượng gửi một yêu cầu nhưng không biết đối tượng nào sẽ nhận và xử lý nó. Điều này được thực hiện bằng cách kết nối các đối tượng nhận yêu cầu thành một chuỗi (chain) và gửi yêu cầu theo chuỗi đó cho đến khi có một đối tượng xử lý nó.



Chain of Responsibility Pattern hoạt động như một danh sách liên kết ([Linked list](https://gpcoder.com/2555-linkedlist-trong-java/)) với việc [đệ quy](https://gpcoder.com/2327-giai-thuat-de-quy-trong-java/) duyệt qua các phần tử (recursive traversal).

## Cài đặt Chain of Responsibility Pattern như thế nào?



Các thành phần tham gia mẫu Chain of Responsibility:

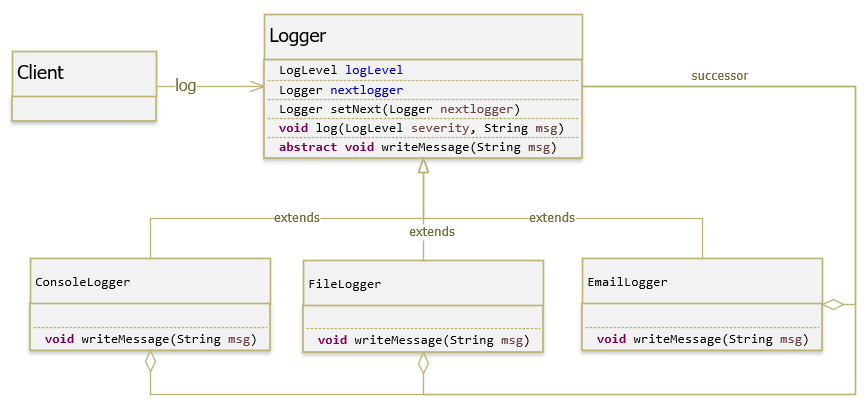
* Handler : định nghĩa 1 interface để xử lý các yêu cầu. Gán giá trị cho đối tượng successor (không bắt buộc).
* ConcreteHandler : xử lý yêu cầu. Có thể truy cập đối tượng successor (thuộc class Handler). Nếu đối tượng ConcreateHandler không thể xử lý được yêu cầu, nó sẽ gởi lời yêu cầu cho successor của nó.
* Client : tạo ra các yêu cầu và yêu cầu đó sẽ được gửi đến các đối tượng tiếp nhận.

Client gửi một yêu cầu để được xử lý gửi nó đến chuỗi (chain) các trình xử lý (handers), đó là các lớp mở rộng lớp Handler. Mỗi Hanlder trong chuỗi lần lượt cố gắng xử lý yêu cầu nhận được từ Client. Nếu trình xử lý đầu tiên (ConcreteHandler) có thể xử lý nó, thì yêu cầu sẽ được xử lý. Nếu không được xử lý thì sẽ gửi đến trình xử lý tiếp theo trong chuỗi (ConcreteHandler + 1).

Các bạn có thể tham khảo cách implement về Chain of Responsibility trong các thư viện của Java như:

* java.util.logging.Logger#log()
* javax.servlet.Filter#doFilter()

### Ví dụ Chain of Responsibility Pattern với ứng dụng Logger



Trong ví dụ này, chúng ta sẽ tự xây dựng hệ thống Logger cho riêng mình như sau:

* Logger: là một abstract class Handler, cho phép thực hiện một chain logger dựa vào giá trị LogLevel ứng với từng Handler. Nếu mức độ lỗi (severity) lớn hơn hoặc bằng với LogLevel mà nó có thể handle thì sẽ thực hiện writeMessage(), đồng thời gọi Hanlder kế tiếp nếu có.
* ConsoleLogger, FileLogger, EmailLogger: đây là các ConcreteHandler, nó xác định LogLevel mà nó có thể xử lý, và cài đặt phương thức writeMessage() cho riêng nó.
* Client : sử dụng Logger để ghi log.
* LogLevel : là một enum (constant), dùng để xác định các mức độ ghi log.
* AppLogger: là một lớp tiện ích, tạo chuỗi Handler để xử lý ghi log.

LogLevel.java

package com.gpcoder.patterns.behavioral.cor.logger;

public enum LogLevel {

NONE(0), INFO(1), DEBUG(2), WARNING(4), ERROR(8), FATAL(16), ALL(32);

private int level;

private LogLevel(int level) {

this.level = level;

}

public int getLevel() {

return level;

}

}

Logger.java

package com.gpcoder.patterns.behavioral.cor.logger;

public abstract class Logger {

protected LogLevel logLevel;

protected Logger nextlogger; // The next Handler in the chain

public Logger(LogLevel logLevel) {

this.logLevel = logLevel;

}

// Set the next logger to make a list/chain of Handlers.

public Logger setNext(Logger nextlogger) {

this.nextlogger = nextlogger;

return nextlogger;

}

public void log(LogLevel severity, String msg) {

if (logLevel.getLevel() <= severity.getLevel()) {

writeMessage(msg);

}

if (nextlogger != null) {

nextlogger.log(severity, msg);

}

}

protected abstract void writeMessage(String msg);

}

ConsoleLogger.java

package com.gpcoder.patterns.behavioral.cor.logger;

public class ConsoleLogger extends Logger {

public ConsoleLogger(LogLevel logLevel) {

super(logLevel);

}

@Override

protected void writeMessage(String msg) {

System.out.println("Console logger: " + msg);

}

}

FileLogger.java

package com.gpcoder.patterns.behavioral.cor.logger;

public class FileLogger extends Logger {

public FileLogger(LogLevel logLevel) {

super(logLevel);

}

@Override

protected void writeMessage(String msg) {

System.out.println("File logger: " + msg);

}

}

EmailLogger.java

package com.gpcoder.patterns.behavioral.cor.logger;

public class EmailLogger extends Logger {

public EmailLogger(LogLevel logLevel) {

super(logLevel);

}

@Override

protected void writeMessage(String msg) {

System.out.println("Email logger: " + msg);

}

}

AppLogger.java

package com.gpcoder.patterns.behavioral.cor.logger;

public class AppLogger {

public static Logger getLogger() {

Logger consoleLogger = new ConsoleLogger(LogLevel.DEBUG);

Logger fileLogger = consoleLogger.setNext(new FileLogger(LogLevel.ERROR));

fileLogger.setNext(new EmailLogger(LogLevel.FATAL));

return consoleLogger;

}

}

Client.java

package com.gpcoder.patterns.behavioral.cor.logger;

/\*\*

\* Chain of Responsibility Pattern Examle

\*

\* @author gpcoder.com

\*/

public class Client {

public static void main(String[] args) {

// Build the chain of responsibility

Logger logger = AppLogger.getLogger();

// Handled by ConsoleLogger since the console has a LogLevel of DEBUG

logger.log(LogLevel.INFO, "Info message");

logger.log(LogLevel.DEBUG, "Debug message");

// Handled by ConsoleLogger and FileLogger

logger.log(LogLevel.ERROR, "Error message");

// Handled by ConsoleLogger, FileLogger, EmailLogger

logger.log(LogLevel.FATAL, "Factal message");

}

}

Output của chương trình:

Console logger: Debug message

Console logger: Error message

File logger: Error message

Console logger: Factal message

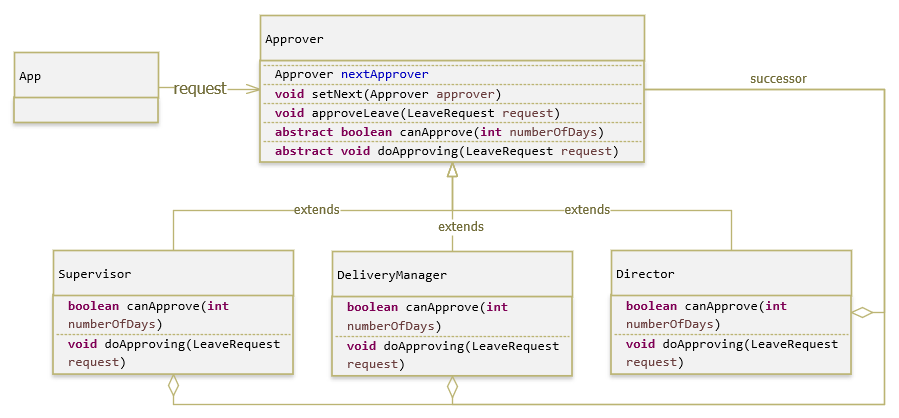
File logger: Factal message

Email logger: Factal message

Ở ví dụ này, chain của chúng ta sẽ kết thúc khi tất cả các đối tượng trong chain có thể xử lý đã xử lý.

### Ví dụ Chain of Responsibility Pattern với ứng dụng LeaveRequest

Một ví dụ khác là ứng dụng phê duyệt xin nghỉ phép. Nếu xin nghỉ <=3 ngày thì Supervisor có thể phê duyệt (approve). Nếu xin nghỉ <=5 ngày thì DeliveryManager có thể approve. Nếu xin nghỉ >5 ngày thì phải được approve bởi Director. Quy trình này có thể linh động tùy theo quy mô phát triển của công ty.



LeaveRequest.java

package com.gpcoder.patterns.behavioral.cor.leaverequest;

public class LeaveRequest {

private int days;

public LeaveRequest(int days) {

this.days = days;

}

public int getDays() {

return days;

}

}

Approver.java

package com.gpcoder.patterns.behavioral.cor.leaverequest;

public abstract class Approver {

protected Approver nextApprover;

public void approveLeave(LeaveRequest request) {

System.out.println("Checking permission for " + this.getClass().getSimpleName());

if (this.canApprove(request.getDays())) {

this.doApproving(request);

} else if (nextApprover != null) {

nextApprover.approveLeave(request);

}

}

public void setNext(Approver approver) {

this.nextApprover = approver;

}

protected abstract boolean canApprove(int numberOfDays);

protected abstract void doApproving(LeaveRequest request);

}

Supervisor.java

package com.gpcoder.patterns.behavioral.cor.leaverequest;

public class Supervisor extends Approver {

@Override

protected boolean canApprove(int numberOfDays) {

return numberOfDays <= 3;

}

@Override

protected void doApproving(LeaveRequest request) {

System.out.println("Leave request approved for " + request.getDays() + " days by Supervisor");

}

}

DeliveryManager.java

package com.gpcoder.patterns.behavioral.cor.leaverequest;

public class DeliveryManager extends Approver {

@Override

protected boolean canApprove(int numberOfDays) {

return numberOfDays <= 5;

}

@Override

protected void doApproving(LeaveRequest request) {

System.out.println("Leave request approved for " + request.getDays() + " days by Delivery Manager");

}

}

Director.java

package com.gpcoder.patterns.behavioral.cor.leaverequest;

public class Director extends Approver {

@Override

protected boolean canApprove(int numberOfDays) {

return numberOfDays > 5;

}

@Override

protected void doApproving(LeaveRequest request) {

System.out.println("Leave request approved for " + request.getDays() + " days by Director");

}

}

LeaveRequestWorkFlow.java

package com.gpcoder.patterns.behavioral.cor.leaverequest;

public class LeaveRequestWorkFlow {

public static Approver getApprover() {

Approver supervisor = new Supervisor();

Approver manager = new DeliveryManager();

Approver director = new Director();

supervisor.setNext(manager);

manager.setNext(director);

return supervisor;

}

}

App.java

package com.gpcoder.patterns.behavioral.cor.leaverequest;

public class App {

public static void main(String[] args) {

LeaveRequestWorkFlow.getApprover().approveLeave(new LeaveRequest(2));

System.out.println("---");

LeaveRequestWorkFlow.getApprover().approveLeave(new LeaveRequest(5));

System.out.println("---");

LeaveRequestWorkFlow.getApprover().approveLeave(new LeaveRequest(7));

}

}

Output của chương trình:

Checking permission for Supervisor

Leave request approved for 2 days by Supervisor

---

Checking permission for Supervisor

Checking permission for DeliveryManager

Leave request approved for 5 days by Delivery Manager

---

Checking permission for Supervisor

Checking permission for DeliveryManager

Checking permission for Director

Leave request approved for 7 days by Director

Ở ví dụ này, chain của chúng ta sẽ kết thúc khi có một đối tượng có thể xử lý.

## Lợi ích của Chain of Responsibility Pattern là gì?

* Giảm kết nối (loose coupling): Thay vì một đối tượng có khả năng xử lý yêu cầu chứa tham chiếu đến tất cả các đối tượng khác, nó chỉ cần một tham chiếu đến đối tượng tiếp theo. Tránh sự liên kết trực tiếp giữa đối tượng gửi yêu cầu (sender) và các đối tượng nhận yêu cầu (receivers).
* Tăng tính linh hoạt : đảm bảo [Open/Closed Principle](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Open-Closed_principle_OCP).
* Phân chia trách nhiệm cho các đối tượng: đảm bảo [Single Responsibility Principle](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Single_responsibility_principle_SRP).
* Có khả năng thay đổi dây chuyền (chain) trong thời gian chạy.
* Không đảm bảo có đối tượng xử lý yêu cầu.

## Sử dụng Chain of Responsibility Pattern khi nào?

* Có nhiều hơn một đối tượng có khả thực xử lý một yêu cầu trong khi đối tượng cụ thể nào xử lý yêu cầu đó lại phụ thuộc vào ngữ cảnh sử dụng.
* Muốn gửi yêu cầu đến một trong số vài đối tượng nhưng không xác định đối tượng cụ thể nào sẽ xử lý yêu cầu đó.
* Khi cần phải thực thi các trình xử lý theo một thứ tự nhất định..
* Khi một tập hợp các đối tượng xử lý có thể thay đổi động: tập hợp các đối tượng có khả năng xử lý yêu cầu có thể không biết trước, có thể thêm bớt hay thay đổi thứ tự sau này.

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/chain_of_responsibility>
* <https://refactoring.guru/design-patterns/chain-of-responsibility>
* <https://www.javatpoint.com/chain-of-responsibility-pattern>
* <https://www.tutorialspoint.com/design_pattern/chain_of_responsibility_pattern.htm>
* Design Patterns: Elements of Reusable Object-Oriented Software – GOF

# Hướng dẫn Java Design Pattern – Command

Đăng vào 07/12/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 2105 Lượt xem

Đôi khi chúng ta cần gửi các yêu cầu cho các đối tượng mà không biết bất cứ điều gì về hoạt động được yêu cầu hoặc người nhận yêu cầu. Chẳng hạn chúng có một ứng dụng văn bản, khi click lên button undo/ redo, save, … yêu cầu sẽ được chuyển đến hệ thống xử lý, chúng ta sẽ không thể biết được đối tượng nào sẽ nhận xử lý, cách nó thực hiện như thế nào. Command Pattern là một Pattern được thiết kế cho những ứng dụng như vậy.

Nội dung [[Ẩn](https://gpcoder.com/4686-huong-dan-java-design-pattern-command/)]

* [1 Command Pattern là gì?](https://gpcoder.com/4686-huong-dan-java-design-pattern-command/" \l "Command_Pattern_la_gi)
* [2 Cài đặt Command Pattern như thế nào?](https://gpcoder.com/4686-huong-dan-java-design-pattern-command/" \l "Cai_dat_Command_Pattern_nhu_the_nao)
  + [2.1 Ví dụ Command Pattern trong ứng dụng mở tài khoản ngân hàng](https://gpcoder.com/4686-huong-dan-java-design-pattern-command/" \l "Vi_du_Command_Pattern_trong_ung_dung_mo_tai_khoan_ngan_hang)
  + [2.2 Ví dụ Command Pattern trong ứng dụng quản lý văn bản](https://gpcoder.com/4686-huong-dan-java-design-pattern-command/" \l "Vi_du_Command_Pattern_trong_ung_dung_quan_ly_van_ban)
* [3 Lợi ích của Command Pattern là gì?](https://gpcoder.com/4686-huong-dan-java-design-pattern-command/" \l "Loi_ich_cua_Command_Pattern_la_gi)
* [4 Sử dụng Command Pattern khi nào?](https://gpcoder.com/4686-huong-dan-java-design-pattern-command/" \l "Su_dung_Command_Pattern_khi_nao)

## Command Pattern là gì?

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

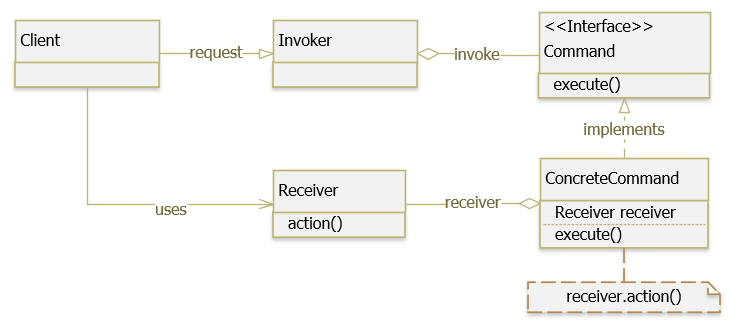
Command Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó cho phép chuyển yêu cầu thành đối tượng độc lập, có thể được sử dụng để tham số hóa các đối tượng với các yêu cầu khác nhau như log, queue (undo/redo), transtraction.

Nói cho dễ hiểu, Command Pattern cho phép tất cả những Request gửi đến object được lưu trữ trong chính object đó dưới dạng một object Command. Khái niệm Command Object giống như một class trung gian được tạo ra để lưu trữ các câu lệnh và trạng thái của object tại một thời điểm nào đó.

Command dịch ra nghĩa là ra lệnh. Commander nghĩa là chỉ huy, người này không làm mà chỉ ra lệnh cho người khác làm. Như vậy, phải có người nhận lệnh và thi hành lệnh. Người ra lệnh cần cung cấp một class đóng gói những mệnh lệnh. Người nhận mệnh lệnh cần phân biệt những interface nào để thực hiện đúng mệnh lệnh.

Command Pattern còn được biết đến như là Action hoặc Transaction.

## Cài đặt Command Pattern như thế nào?



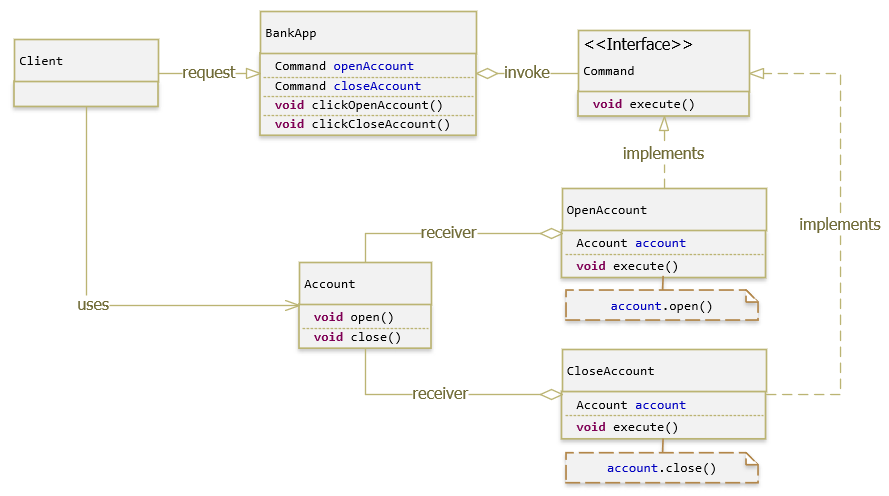
Các thành phần tham gia trong Command Pattern:

* Command : là một interface hoặc abstract class, chứa một phương thức trừu tượng thực thi (execute) một hành động (operation). Request sẽ được đóng gói dưới dạng Command.
* ConcreteCommand : là các implementation của Command. Định nghĩa một sự gắn kết giữa một đối tượng Receiver và một hành động. Thực thi execute() bằng việc gọi operation đang hoãn trên Receiver. Mỗi một ConcreteCommand sẽ phục vụ cho một case request riêng.
* Client : tiếp nhận request từ phía người dùng, đóng gói request thành ConcreteCommand thích hợp và thiết lập receiver của nó.
* Invoker : tiếp nhận ConcreteCommand từ Client và gọi execute() của ConcreteCommand để thực thi request.
* Receiver : đây là thành phần thực sự xử lý business logic cho case request. Trong phương execute() của ConcreteCommand chúng ta sẽ gọi method thích hợp trong Receiver.

Như vậy, Client và Invoker sẽ thực hiện việc tiếp nhận request. Còn việc thực thi request sẽ do Command, ConcreteCommand và Receiver đảm nhận.

### Ví dụ Command Pattern trong ứng dụng mở tài khoản ngân hàng

Một hệ thống ngân hàng cung cấp ứng dụng cho khách hàng (client) có thể mở (open) hoặc đóng (close) tài khoản trực tuyến. Hệ thống này được thiết kế theo dạng module, mỗi module sẽ thực hiện một nhiệm vụ riêng của nó, chẳng hạn mở tài khoản (OpenAccount), đóng tài khoản (CloseAccount). Do hệ thống không biết mỗi module sẽ làm gì, nên khi có yêu cầu client (chẳng hạn clickOpenAccount, clickCloseAccount), nó sẽ đóng gói yêu cầu này và gọi module xử lý.



Ứng dụng của chúng ta bao gồm các lớp xử lý sau:

* Account : là một request class.
* Command : là một interface của Command Pattern, cung cấp phương thức execute().
* OpenAccount, CloseAccount : là các ConcreteCommand, cài đặt các phương thức của Command, sẽ thực hiện các xử lý thực tế.
* BankApp : là một class, hoạt động như Invoker, gọi execute() của ConcreteCommand để thực thi request.
* Client : tiếp nhận request từ phía người dùng, đóng gói request thành ConcreteCommand thích hợp và gọi thực thi các Command.

Account.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | package com.gpcoder.patterns.behavioral.command.bank;    public class Account {      private String name;        public Account(String name) {          this.name = name;      }        public void open() {          System.out.println("Account [" + name + "] Opened\n");      }        public void close() {          System.out.println("Account [" + name + "] Closed\n");      }  } |

Command.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | package com.gpcoder.patterns.behavioral.command.bank;    public interface Command {        void execute();  } |

OpenAccount.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package com.gpcoder.patterns.behavioral.command.bank;    public class OpenAccount implements Command {        private Account account;        public OpenAccount(Account account) {          this.account = account;      }        @Override      public void execute() {          account.open();      }  } |

CloseAccount.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package com.gpcoder.patterns.behavioral.command.bank;    public class CloseAccount implements Command {        private Account account;        public CloseAccount(Account account) {          this.account = account;      }        @Override      public void execute() {          account.close();      }  } |

BankApp.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | package com.gpcoder.patterns.behavioral.command.bank;    public class BankApp {        private Command openAccount;      private Command closeAccount;        public BankApp(Command openAccount, Command closeAccount) {          this.openAccount = openAccount;          this.closeAccount = closeAccount;      }        public void clickOpenAccount() {          System.out.println("User click open an account");          openAccount.execute();      }        public void clickCloseAccount() {          System.out.println("User click close an account");          closeAccount.execute();      }  } |

Client.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package com.gpcoder.patterns.behavioral.command.bank;    public class Client {        public static void main(String[] args) {          Account account = new Account("gpcoder");            Command open = new OpenAccount(account);          Command close = new CloseAccount(account);          BankApp bankApp = new BankApp(open, close);            bankApp.clickOpenAccount();          bankApp.clickCloseAccount();      }  } |

Output của chương trình:

|  |  |
| --- | --- |
| 1  2  3  4  5 | User click open an account  Account [gpcoder] Opened    User click close an account  Account [gpcoder] Closed |

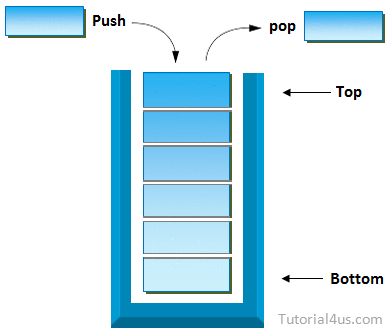
### Ví dụ Command Pattern trong ứng dụng quản lý văn bản

Ứng dụng văn bản cần một chức năng để thêm hoặc lưu trữ những hành động undo hay redo.

Lớp Document chỉ cung cấp phương thức ghi thêm một dòng văn bản mới hoặc xóa một dòng văn bản đã ghi trước đó.

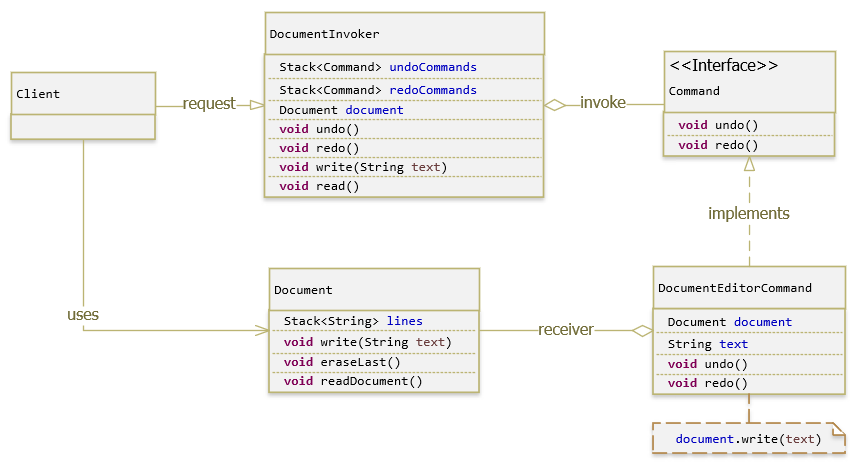
Chúng ta sẽ xây dựng một interface Command để cung cấp hành động undo/ redo. Để sử dụng Command chúng ta cần một DocumentInvoker, lớp này sử dụng tính năng của [Stack](https://vi.wikipedia.org/wiki/Ngăn_xếp) để lưu lại lịch sử những lần thêm mới và những lần xóa, tương ứng với undoCommands và redoCommands.

[Stack](https://vi.wikipedia.org/wiki/Ngăn_xếp) (ngăn xếp) là một cấu trúc dữ liệu trừu tượng hoạt động theo nguyên lý “vào sau ra trước” (Last In First Out (LIFO).



Một số phương thức của Stack:

* push() : thêm phần tử trên đỉnh của Stack.
* pop() : xóa phần tử trên đỉnh của Stack và trả về phần tử bị xóa.
* peek() : lấy phần tử trên đỉnh của Stack, nhưng không xóa nó khỏi Stack.
* clear() : xóa tất cả các phần tử trong Stack.
* isEmpty() : kiểm tra Stack có chứa phần tử nào không.



Document.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | package com.gpcoder.patterns.behavioral.command.document;    import java.util.Stack;    public class Document {      private Stack<String> lines = new Stack<>();        public void write(String text) {          lines.push(text);      }        public void eraseLast() {          if (!lines.isEmpty()) {              lines.pop();          }      }        public void readDocument() {          for (String line : lines) {              System.out.println(line);          }      }  } |

Command.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | package com.gpcoder.patterns.behavioral.command.document;    public interface Command {      void undo();        void redo();  } |

DocumentEditorCommand.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | package com.gpcoder.patterns.behavioral.command.document;    public class DocumentEditorCommand implements Command {        private Document document;      private String text;        public DocumentEditorCommand(Document document, String text) {          this.document = document;          this.text = text;          this.document.write(text);      }        public void undo() {          document.eraseLast();      }        public void redo() {          document.write(text);      }  } |

DocumentInvoker.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | package com.gpcoder.patterns.behavioral.command.document;    import java.util.Stack;    public class DocumentInvoker {      private Stack<Command> undoCommands = new Stack<>();      private Stack<Command> redoCommands = new Stack<>();      private Document document = new Document();        public void undo() {          if (!undoCommands.isEmpty()) {              Command cmd = undoCommands.pop();              cmd.undo();              redoCommands.push(cmd);          } else {              System.out.println("Nothing to undo");          }      }        public void redo() {          if (!redoCommands.isEmpty()) {              Command cmd = redoCommands.pop();              cmd.redo();              undoCommands.push(cmd);          } else {              System.out.println("Nothing to redo");          }      }        public void write(String text) {          Command cmd = new DocumentEditorCommand(document, text);          undoCommands.push(cmd);          redoCommands.clear();      }        public void read() {          document.readDocument();      }  } |

Client.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | package com.gpcoder.patterns.behavioral.command.document;    public class Client {        public static void main(String[] args) {          DocumentInvoker instance = new DocumentInvoker();          instance.write("The 1st text. ");          instance.undo();          instance.read(); // EMPTY            instance.redo();          instance.read(); // The 1st text.            instance.write("The 2nd text. ");          instance.write("The 3rd text. ");          instance.read(); // The 1st text. The 2nd text. The 3rd text.          instance.undo(); // The 1st text. The 2nd text.          instance.undo(); // The 1st text.          instance.undo(); // EMPTY          instance.undo(); // Nothing to undo      }  } |

Output của chương trình:

|  |  |
| --- | --- |
| 1  2  3  4  5 | The 1st text.  The 1st text.  The 2nd text.  The 3rd text.  Nothing to undo |

## Lợi ích của Command Pattern là gì?

* Dễ dàng thêm các Command mới trong hệ thống mà không cần thay đổi trong các lớp hiện có. Đảm bảo [Open/Closed Principle](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Open-Closed_principle_OCP).
* Tách đối tượng gọi operation từ đối tượng thực sự thực hiện operation. Giảm kết nối giữa Invoker và Receiver.
* Cho phép tham số hóa các yêu cầu khác nhau bằng một hành động để thực hiện.
* Cho phép lưu các yêu cầu trong hàng đợi.
* Đóng gói một yêu cầu trong một đối tượng. Dễ dàng chuyển dữ liệu dưới dạng đối tượng giữa các thành phần hệ thống.

## Sử dụng Command Pattern khi nào?

* Khi cần tham số hóa các đối tượng theo một hành động thực hiện.
* Khi cần tạo và thực thi các yêu cầu vào các thời điểm khác nhau.
* Khi cần hỗ trợ tính năng undo, log , callback hoặc transaction.

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/command>
* <https://refactoring.guru/design-patterns/command>
* <https://www.javatpoint.com/command-pattern>
* <https://www.tutorialspoint.com/design_pattern/command_pattern.htm>
* <https://quachson.wordpress.com/2011/04/22/design-pattern-command/>
* Design Patterns: Elements of Reusable Object-Oriented Software – GOF

# Hướng dẫn Java Design Pattern – Interpreter

Đăng vào 10/12/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 1444 Lượt xem

Nội dung [[Ẩn](https://gpcoder.com/4702-huong-dan-java-design-pattern-interpreter/)]

* [1 Interpreter Pattern là gì?](https://gpcoder.com/4702-huong-dan-java-design-pattern-interpreter/" \l "Interpreter_Pattern_la_gi)
* [2 Cài đặt Interpreter Pattern như thế nào?](https://gpcoder.com/4702-huong-dan-java-design-pattern-interpreter/" \l "Cai_dat_Interpreter_Pattern_nhu_the_nao)
  + [2.1 Ví dụ Interpreter Pattern trong ứng dụng calculator theo ngôn ngữ tự nhiên](https://gpcoder.com/4702-huong-dan-java-design-pattern-interpreter/" \l "Vi_duInterpreter_Pattern_trong_ung_dung_calculator_theo_ngon_ngu_tu_nhien)
  + [2.2 Ví dụ sử dụng Interpreter để chuyển chữ cái La Mã sang số thập phân](https://gpcoder.com/4702-huong-dan-java-design-pattern-interpreter/" \l "Vi_du_su_dung_Interpreter_de_chuyen_chu_cai_La_Ma_sang_so_thap_phan)
* [3 Lợi ích của Interpreter Pattern là gì?](https://gpcoder.com/4702-huong-dan-java-design-pattern-interpreter/" \l "Loi_ich_cua_Interpreter_Pattern_la_gi)
* [4 Sử dụng Interpreter Pattern khi nào?](https://gpcoder.com/4702-huong-dan-java-design-pattern-interpreter/" \l "Su_dung_Interpreter_Pattern_khi_nao)

## Interpreter Pattern là gì?

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Interpreter Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern).

Interpreter nghĩa là thông dịch, mẫu này nói rằng “để xác định một biểu diễn ngữ pháp của một ngôn ngữ cụ thể, cùng với một thông dịch viên sử dụng biểu diễn này để diễn dịch các câu trong ngôn ngữ”.

Nói cho dễ hiểu, Interpreter Pattern giúp người lập trình có thể “xây dựng” những đối tượng “động” bằng cách đọc mô tả về đối tượng rồi sau đó “xây dựng” đối tượng đúng theo mô tả đó.

Metadata (mô tả) –> [Interpreter Pattern] –> Đối tượng tương ứng.

Interpreter Pattern có hạn chế về phạm vi áp dụng. Mẫu này thường được sử dụng để định nghĩa bộ ngữ pháp đơn giản (grammar), trong các công cụ quy tắc đơn giản (rule), …

## Cài đặt Interpreter Pattern như thế nào?



Các thành phần tham gia mẫu Interpreter:

* Context : là phần chứa thông tin biểu diễn mẫu chúng ta cần xây dựng.
* Expression : là một interface hoặc abstract class, định nghĩa phương thức interpreter chung cho tất cả các node trong cấu trúc cây phân tích ngữ pháp. Expression được biểu diễn như một cấu trúc cây phân cấp, mỗi implement của Expression có thể gọi một node.
* TerminalExpression (biểu thức đầu cuối): cài đặt các phương thức của Expression, là những biểu thức có thể được diễn giải trong một đối tượng duy nhất, chứa các xử lý logic để đưa thông tin của context thành đối tượng cụ thể.
* NonTerminalExpression (biểu thức không đầu cuối): cài đặt các phương thức của Expression, biểu thức này chứa một hoặc nhiều biểu thức khác nhau, mỗi biểu thức có thể là biểu thức đầu cuối hoặc không phải là biểu thức đầu cuối. Khi một phương thức interpret() của lớp biểu thức không phải là đầu cuối được gọi, nó sẽ gọi đệ quy đến tất cả các biểu thức khác mà nó đang giữ.
* Client : đại diện cho người dùng sử dụng lớp Interpreter Pattern. Client sẽ xây dựng cây biểu thức đại diện cho các lệnh được thực thi, gọi phương thức interpreter() của node trên cùng trong cây, có thể truyền context để thực thi tất cả các lệnh trong cây.

### Ví dụ Interpreter Pattern trong ứng dụng calculator theo ngôn ngữ tự nhiên

Trong ví dụ bên dưới chúng ta sẽ xây dựng ứng dụng calculator theo ngôn ngữ tự nhiên. Ví dụ: 20 cộng 8 = 28 hay 10 trừ 4 = 6

Chương trình của chúng ta như sau:



InterpreterEngineContext.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | package com.gpcoder.patterns.behavioral.interpreter.math;    public class InterpreterEngineContext {        public int add(String input) {          String[] tokens = interpret(input);          int num1 = Integer.parseInt(tokens[0]);          int num2 = Integer.parseInt(tokens[1]);          return (num1 + num2);      }        public int subtract(String input) {          String[] tokens = interpret(input);          int num1 = Integer.parseInt(tokens[0]);          int num2 = Integer.parseInt(tokens[1]);          return (num1 - num2);      }        private String[] interpret(String input) {          String str = input.replaceAll("[^0-9]", " ");          str = str.replaceAll("( )+", " ").trim();          return str.split(" ");      }  } |

Expression.java

|  |  |
| --- | --- |
| 1  2  3  4  5 | package com.gpcoder.patterns.behavioral.interpreter.math;    public interface Expression {      int interpret(InterpreterEngineContext context);  } |

AddExpression.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package com.gpcoder.patterns.behavioral.interpreter.math;    public class AddExpression implements Expression {        private String expression;        public AddExpression(String expression) {          this.expression = expression;      }        @Override      public int interpret(InterpreterEngineContext context) {          return context.add(expression);      }  } |

SubtractExpression.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package com.gpcoder.patterns.behavioral.interpreter.math;    public class SubtractExpression implements Expression {        private String expression;        public SubtractExpression(String expression) {          this.expression = expression;      }        @Override      public int interpret(InterpreterEngineContext context) {          return context.subtract(expression);      }  } |

Client.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | package com.gpcoder.patterns.behavioral.interpreter.math;    public class Client {        public static void main(String args[]) {          System.out.println("20 cộng 8 = " + interpret("20 cộng 8"));          System.out.println("10 trừ 4 = " + interpret("10 trừ 4"));      }        private static int interpret(String input) {          Expression exp = null;          if (input.contains("cộng")) {              exp = new AddExpression(input);          } else if (input.contains("trừ")) {              exp = new SubtractExpression(input);          } else {              throw new UnsupportedOperationException();          }          return exp.interpret(new InterpreterEngineContext());      }  } |

Output của chương trình:

|  |  |
| --- | --- |
| 1  2 | 20 cộng 8 = 28  10 trừ 4 = 6 |

### Ví dụ sử dụng Interpreter để chuyển chữ cái La Mã sang số thập phân

Chữ số La Mã được thể hiện bằng chữ cái của bảng chữ cái: I=1;  V=5;  X=10;  L=50;  C=100;  D=500;  M=1000.

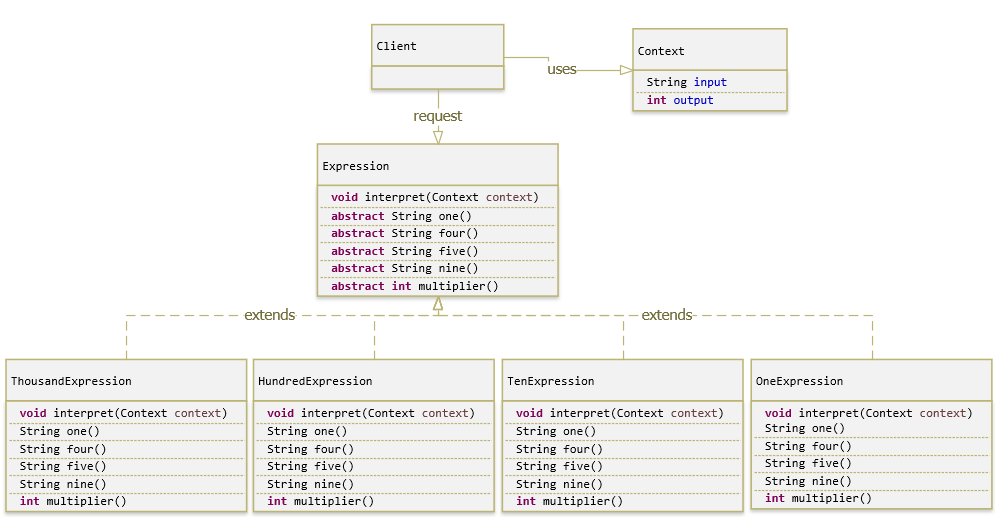
Có bốn nguyên tắc cơ bản để đọc và viết chữ số La Mã:

* Một chữ cái có thể lặp lại giá trị của nó nhiều lần, tối đa ba lần. Ví dụ: XXX = 30, CC = 200,…
* Nếu một hoặc nhiều chữ cái được đặt sau một chữ cái có giá trị lớn hơn, cộng số trước đó. Ví dụ:
  + VI = 6 (5 + 1 = 6)
  + LXX = 70 (50 + 10 + 10 = 70)
  + MCC = 1200 (1000 + 100 + 100 = 1200)
* Nếu một chữ cái được đặt trước một chữ cái có giá trị lớn hơn, trừ đi số trước đó. Ví dụ:
  + IV = 4 (5 – 1 = 4)
  + XC = 90 (100 – 10 = 90)
  + CM = 900 (1000 – 100 = 900)

Chi tiết về chữ cái La Mã, các bạn xem thêm ở các link bên dưới:

* <https://en.wikipedia.org/wiki/Roman_numerals>
* <https://www.factmonster.com/math/numbers/roman-numerals>
* <https://www.artofmanliness.com/articles/even-when-not-in-romeyou-should-know-your-roman-numerals/>

Chương trình của chúng ta như sau:



Context.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package com.gpcoder.patterns.behavioral.interpreter.roman;    public class Context {      private String input;      private int output;        public Context(String input) {          this.input = input;      }        public void setInput(String input) {          this.input = input;      }        public String getInput() {          return input;      }        public int getOutput() {          return output;      }        public void setOutput(int output) {          this.output = output;      }  } |

Expression.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | package com.gpcoder.patterns.behavioral.interpreter.roman;    public abstract class Expression {        public void interpret(Context context) {          if (context.getInput().length() == 0) {              return;          }            if (context.getInput().startsWith(nine())) {              context.setOutput(context.getOutput() + 9 \* multiplier());              context.setInput(context.getInput().substring(2));          } else if (context.getInput().startsWith(four())) {              context.setOutput(context.getOutput() + 4 \* multiplier());              context.setInput(context.getInput().substring(2));          } else if (context.getInput().startsWith(five())) {              context.setOutput(context.getOutput() + 5 \* multiplier());              context.setInput(context.getInput().substring(1));          }            while (context.getInput().startsWith(one())) {              context.setOutput(context.getOutput() + 1 \* multiplier());              context.setInput(context.getInput().substring(1));          }      }        public abstract String one();        public abstract String four();        public abstract String five();        public abstract String nine();        public abstract int multiplier();  } |

ThousandExpression.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | package com.gpcoder.patterns.behavioral.interpreter.roman;    public class ThousandExpression extends Expression {        @Override      public String one() {          return "M";      }        @Override      public String four() {          return " ";      }        @Override      public String five() {          return " ";      }        @Override      public String nine() {          return " ";      }        @Override      public int multiplier() {          return 1000;      }  } |

HundredExpression.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | package com.gpcoder.patterns.behavioral.interpreter.roman;    public class HundredExpression extends Expression {        @Override      public String one() {          return "C";      }        @Override      public String four() {          return "CD";      }        @Override      public String five() {          return "D";      }        @Override      public String nine() {          return "CM";      }        @Override      public int multiplier() {          return 100;      }  } |

TenExpression.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | package com.gpcoder.patterns.behavioral.interpreter.roman;    public class TenExpression extends Expression {        @Override      public String one() {          return "X";      }        @Override      public String four() {          return "XL";      }        @Override      public String five() {          return "L";      }        @Override      public String nine() {          return "XC";      }        @Override      public int multiplier() {          return 10;      }    } |

OneExpression.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | package com.gpcoder.patterns.behavioral.interpreter.roman;    public class OneExpression extends Expression {        @Override      public String one() {          return "I";      }        @Override      public String four() {          return "IV";      }        @Override      public String five() {          return "V";      }        @Override      public String nine() {          return "IX";      }        @Override      public int multiplier() {          return 1;      }  } |

Client.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | package com.gpcoder.patterns.behavioral.interpreter.roman;    import java.util.ArrayList;  import java.util.List;    public class Client {        public static void main(String[] args) {          String[] romans = { "IV", "XII", "CLIX", "MMXVIII", "MMMDLIV" };          for (String roman : romans) {              convertRomanToNumber(roman);          }      }        private static void convertRomanToNumber(String roman) {          List<Expression> tree = new ArrayList<>();          tree.add(new ThousandExpression());          tree.add(new HundredExpression());          tree.add(new TenExpression());          tree.add(new OneExpression());            Context context = new Context(roman);          for (Expression exp : tree) {              exp.interpret(context);          }          System.out.println(roman + " = " + context.getOutput());      }  } |

Output của chương trình:

|  |  |
| --- | --- |
| 1  2  3  4  5 | IV = 4  XII = 12  CLIX = 159  MMXVIII = 2018  MMMDLIV = 3554 |

## Lợi ích của Interpreter Pattern là gì?

* Dễ dàng thay đổi và mở rộng ngữ pháp. Vì mẫu này sử dụng các lớp để biểu diễn các quy tắc ngữ pháp, chúng ta có thể sử dụng thừa kế để thay đổi hoặc mở rộng ngữ pháp. Các biểu thức hiện tại có thể được sửa đổi theo từng bước và các biểu thức mới có thể được định nghĩa lại các thay đổi trên các biểu thức cũ.
* Cài đặt và sử dụng ngữ pháp rất đơn giản. Các lớp xác định các nút trong cây cú pháp có các implement tương tự. Các lớp này dễ viết và các phân cấp con của chúng có thể được tự động hóa bằng trình biên dịch hoặc trình tạo trình phân tích cú pháp.

## Sử dụng Interpreter Pattern khi nào?

Interpreter Pattern được sử dụng hiệu quả khi:

* Bộ ngữ pháp đơn giản. Pattern này cần xác định ít nhất một lớp cho mỗi quy tắc trong ngữ pháp. Do đó ngữ pháp có chứa nhiều quy tắc có thể khó quản lý và bảo trì.
* Không quan tâm nhiều về hiệu suất. Do bộ ngữ pháp được phân tích trong cấu trúc phân cấp (cây) nên hiệu suất không được đảm bảo.

Interpreter Pattern thường được sử dụng trong trình biên dịch (compiler), định nghĩa các bộ ngữ pháp, rule, trình phân tích SQL, XML, …

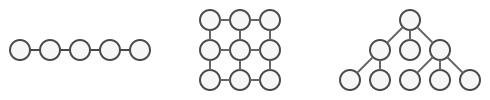
Tài liệu tham khảo:

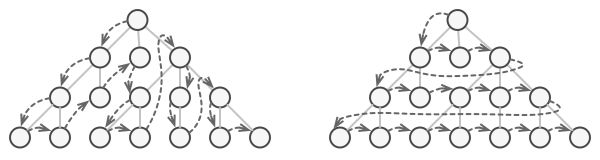
* <https://sourcemaking.com/design_patterns/interpreter>
* <https://www.javatpoint.com/interpreter-pattern>
* <https://www.tutorialspoint.com/design_pattern/interpreter_pattern.htm>
* <https://www.journaldev.com/1635/interpreter-design-pattern-java>
* <https://www.dineshonjava.com/interpreter-pattern-design-patterns-java/>
* <https://www.javacodegeeks.com/2015/09/interpreter-design-pattern.html>
* <https://www.opencodez.com/java/interpreter-design-pattern.htm>
* Design Patterns: Elements of Reusable Object-Oriented Software – GOF

# Hướng dẫn Java Design Pattern – Iterator

Đăng vào 17/12/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 1877 Lượt xem

Trong khi phát triển các ứng dụng, chúng ta làm việc với nhiều loại tập hợp như: cấu trúc cây, mảng, tập hợp, bảng băm, ngăn xếp, hàng đợi, … Cách thức mà tập hợp này lưu trữ đối tượng của nó rất khác nhau, và nếu bạn muốn truy cập dữ liệu của những đối tượng này, bạn phải học những kỹ thuật khác nhau cho từng loại tập hợp. Khi đó, mẫu Iterator là một giải pháp tốt. Chúng ta có thể sử dụng một interface được xác định phương thức cụ thể để truy cập tới từng phần tử của tập hợp. Sử dụng những phương thức này, chúng ta có thể truy xuất tới các phần tử trong tập hợp theo cùng cách dễ dàng nhất.





Trong phần tiếp theo của bài viết này chúng ta sẽ cùng tìm hiểu chi tiết về Iterator Pattern, cách cài đặt, lợi ích mà nó mang lại và khi nào chúng ta có thể áp dụng nó.

Nội dung [[Ẩn](https://gpcoder.com/4724-huong-dan-java-design-pattern-iterator/)]

* [1 Iterator Pattern là gì?](https://gpcoder.com/4724-huong-dan-java-design-pattern-iterator/" \l "Iterator_Pattern_la_gi)
* [2 Cài đặt Iterator Pattern như thế nào?](https://gpcoder.com/4724-huong-dan-java-design-pattern-iterator/" \l "Cai_dat_Iterator_Pattern_nhu_the_nao)
* [3 Lợi ích của Iterator Pattern là gì?](https://gpcoder.com/4724-huong-dan-java-design-pattern-iterator/" \l "Loi_ich_cua_Iterator_Pattern_la_gi)
* [4 Sử dụng Iterator Pattern khi nào?](https://gpcoder.com/4724-huong-dan-java-design-pattern-iterator/" \l "Su_dungIterator_Pattern_khi_nao)

## Iterator Pattern là gì?

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Iterator Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó được sử dụng để “Cung cấp một cách thức truy cập tuần tự tới các phần tử của một đối tượng tổng hợp, mà không cần phải tạo dựng riêng các phương pháp truy cập cho đối tượng tổng hợp này”.

Nói cách khác, một Iterator được thiết kế cho phép xử lý nhiều loại tập hợp khác nhau bằng cách truy cập những phần tử của tập hợp với cùng một phương pháp, cùng một cách thức định sẵn, mà không cần phải hiểu rõ về những chi tiết bên trong của những tập hợp này.

Iterator thường được viết trong Java như là những lớp độc lập. Ý tưởng thiết kế này là một trong những kỹ thuật được gọi là “đơn trách nhiệm – [Single responsibility principle (SRP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Single_responsibility_principle_SRP)” – một lớp chỉ có duy nhất một công việc để làm. Hãy suy nghĩ rằng tập hợp duy trì các phần tử, một iterator cung cấp cách thức làm việc với các phần tử đó. Đó cũng là lý do tại sao những Iterator có thể làm việc được trong các tập hợp khác nhau.

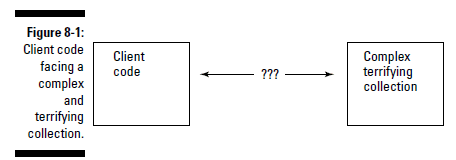
Tách biệt trách nhiệm giữa các lớp rất hữu dụng khi một lớp bị thay đổi. Nếu có quá nhiều thứ bên trong một lớp đơn lẻ, sẽ rất khó khăn để viết lại mã nguồn. Khi diễn ra sự thay đổi, một lớp “đơn trách nhiệm” sẽ chỉ có một lý do duy nhất để thay đổi.

Chúng ta có thể thấy Interator Pattern được áp dụng trong java với Interface iterator trong gói java.util.Iterator. Interface này định nghĩa các phương thức sau:

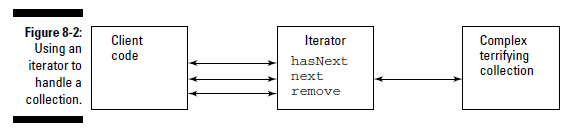
* Hàm next() : trả về phần tử kế tiếp trong tập hợp
* Hàm hasNext() : trả về giá trị True nếu vẫn còn phần tử trong tập hợp và trả về false trong trường hợp ngược lại.

Đó là cách Iterator làm việc. Nó cung cấp một giao diện đơn giản, nhất quán để làm việc với các tập hợp khác nhau.

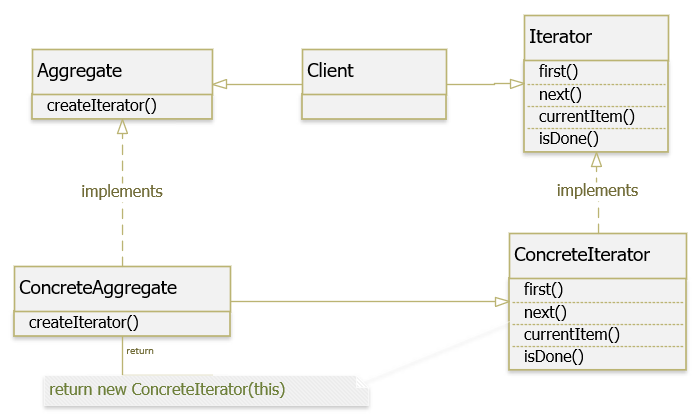
Giả sử rằng Client phải làm việc với một tập hợp phức tạp và rắc rối ( như hình sau) và không biết cách thức làm việc với nó như thế nào.



Client có thể sử dụng iterator để làm cầu nối với tập hợp, và client có thể sử dụng các phương thức cơ bản của Iterator để giao tiếp với tập hợp. Như hình sau:



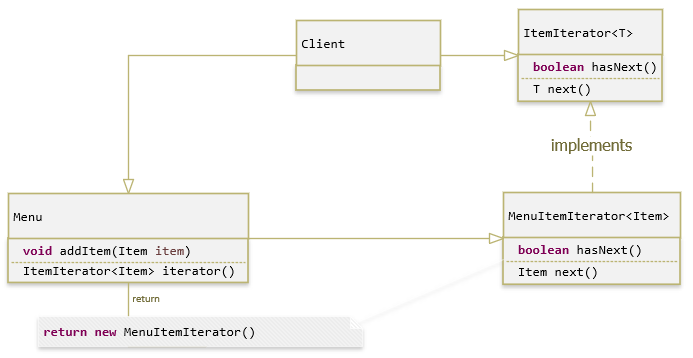
## Cài đặt Iterator Pattern như thế nào?



Các thành phần tham gia mẫu Iterator:

* Aggregate : là một interface định nghĩa định nghĩa các phương thức để tạo Iterator object.
* ConcreteAggregate : cài đặt các phương thức của Aggregate, nó cài đặt interface tạo Iterator để trả về một thể hiện của ConcreteIterator thích hợp.
* Iterator : là một interface hay abstract class, định nghĩa các phương thức để truy cập và duyệt qua các phần tử.
* ConcreteIterator : cài đặt các phương thức của Iterator, giữ index khi duyệt qua các phần tử.
* Client : đối tượng sử dụng Iterator Pattern, nó yêu cầu một iterator từ một đối tượng collection để duyệt qua các phần tử mà nó giữ. Các phương thức của iterator được sử dụng để truy xuất các phần tử từ collection theo một trình tự thích hợp.

Ví dụ: tạo Iterator để duyệt qua các item trong một menu.



Item.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | package com.gpcoder.patterns.behavioral.iterator;    public class Item {      private String title;      private String url;        public Item(String title, String url) {          super();          this.title = title;          this.url = url;      }        @Override      public String toString() {          return "Item [title=" + title + ", url=" + url + "]";      }  } |

ItemIterator.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | package com.gpcoder.patterns.behavioral.iterator;    public interface ItemIterator<T> {        boolean hasNext();        T next();  } |

Menu.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | package com.gpcoder.patterns.behavioral.iterator;    import java.util.ArrayList;  import java.util.List;    public class Menu {      private List<Item> menuItems = new ArrayList<>();        public void addItem(Item item) {          menuItems.add(item);      }        public ItemIterator<Item> iterator() {          return new MenuItemIterator();      }        class MenuItemIterator implements ItemIterator<Item> {          private int currentIndex = 0;            @Override          public boolean hasNext() {              return currentIndex < menuItems.size();          }            @Override          public Item next() {              return menuItems.get(currentIndex++);          }      }  } |

Client.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | package com.gpcoder.patterns.behavioral.iterator;    public class Client {      public static void main(String[] args) {            Menu menu = new Menu();          menu.addItem(new Item("Home", "/home"));          menu.addItem(new Item("Java", "/java"));          menu.addItem(new Item("Spring Boot", "/spring-boot"));            ItemIterator<Item> iterator = menu.iterator();          while (iterator.hasNext()) {              Item item = iterator.next();              System.out.println(item);          }      }  } |

Output của chương trình:

|  |  |
| --- | --- |
| 1  2  3 | Item [title=Home, url=/home]  Item [title=Java, url=/java]  Item [title=Spring Boot, url=/spring-boot] |

Các bạn có thể thêm về cách implement Iterator Pattern trong JDK như:

* Tất cả các cài đặt của java.util.Iterator
* Tất cả các cài đặt của java.util.Enumeration

## Lợi ích của Iterator Pattern là gì?

Một số lợi ích khi sử dụng Iterator Pattern:

* Đảm bảo nguyên tắc [Single responsibility principle (SRP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Single_responsibility_principle_SRP) : chúng ta có thể tách phần cài đặt các phương thức của tập hợp và phần duyệt qua các phần tử (iterator) theo từng class riêng lẻ.
* Đảm bảo nguyên tắc [Open/Closed Principle (OCP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Open-Closed_principle_OCP) : chúng ta có thể implement các loại collection mới và iterator mới, sau đó chuyển chúng vào code hiện có mà không vi phạm bất cứ nguyên tắc gì.
* Chúng ta có thể truy cập song song trên cùng một tập hợp vì mỗi đối tượng iterator có chứa trạng thái riêng của nó.

Một số điểm cần xem xét khi sử dụng Iterator:

* Sử dụng iterator có thể kém hiệu quả hơn so với việc duyệt qua các phần tử của bộ sưu tập một cách trực tiếp.
* Có thể không cần thiết nếu ứng dụng chỉ hoạt động với các collection đơn giản.

## Sử dụng Iterator Pattern khi nào?

* Cần truy cập nội dung của đối tượng trong tập hợp mà không cần biết nội dung cài đặt bên trong nó.
* Hỗ trợ truy xuất nhiều loại tập hợp khác nhau.
* Cung cấp một interface duy nhất để duyệt qua các phần tử của một tập hợp.

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/iterator>
* <https://refactoring.guru/design-patterns/iterator>
* <https://www.javatpoint.com/iterator-pattern>
* <https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm>
* Design Patterns: Elements of Reusable Object-Oriented Software – GOF
* Design Pattern for dummies

# Hướng dẫn Java Design Pattern – Mediator

Đăng vào 20/12/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 1739 Lượt xem

Để có một thiết kế hướng đối tượng tốt, chúng ta phải tạo ra nhiều class tương tác với nhau. Nếu một số nguyên tắc nhất định không được áp dụng, cuối cùng hệ thống trở thành một mớ hỗn độn trong đó mỗi đối tượng phụ thuộc (dependency) vào nhiều đối tượng khác để thực thi. Để tránh các class kết hợp chặt chẽ, chúng ta cần một cơ chế để tạo thuận lợi cho sự tương tác giữa các đối tượng theo cách mà các đối tượng không nhận thức được sự tồn tại của các đối tượng khác. Một trong những cách để giải quyết vấn đề này là áp dụng Mediator Pattern.

Nội dung [[Ẩn](https://gpcoder.com/4740-huong-dan-java-design-pattern-mediator/)]

* [1 Mediator Pattern là gì?](https://gpcoder.com/4740-huong-dan-java-design-pattern-mediator/" \l "Mediator_Pattern_la_gi)
* [2 Cài đặt Mediator Pattern như thế nào?](https://gpcoder.com/4740-huong-dan-java-design-pattern-mediator/" \l "Cai_dat_Mediator_Pattern_nhu_the_nao)
  + [2.1 Ví dụ Mediator Pattern trong hệ thống chat](https://gpcoder.com/4740-huong-dan-java-design-pattern-mediator/" \l "Vi_du_Mediator_Pattern_trong_he_thong_chat)
  + [2.2 Ví dụ Mediator Pattern trong ứng dụng điều khiển đèn giao thông](https://gpcoder.com/4740-huong-dan-java-design-pattern-mediator/" \l "Vi_du_Mediator_Pattern_trong_ung_dung_dieu_khien_den_giao_thong)
* [3 Lợi ích của Mediator Pattern là gì?](https://gpcoder.com/4740-huong-dan-java-design-pattern-mediator/" \l "Loi_ich_cua_Mediator_Pattern_la_gi)
* [4 Sử dụng Mediator Pattern khi nào?](https://gpcoder.com/4740-huong-dan-java-design-pattern-mediator/" \l "Su_dung_Mediator_Pattern_khi_nao)
* [5 So sánh Mediator Pattern với các Pattern khác](https://gpcoder.com/4740-huong-dan-java-design-pattern-mediator/" \l "So_sanh_Mediator_Pattern_voi_cac_Pattern_khac)

## Mediator Pattern là gì?

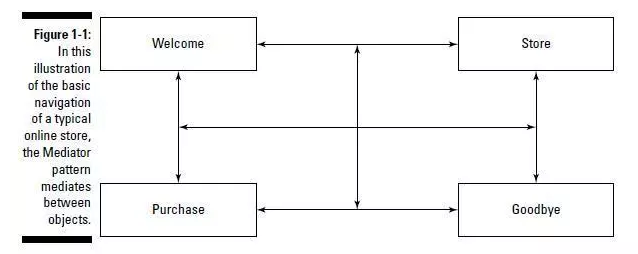
Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Mediator Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Mediator có nghĩa là người trung gian. Pattern này nói rằng “Định nghĩa một đối tượng gói gọn cách một tập hợp các đối tượng tương tác. Mediator thúc đẩy sự khớp nối lỏng lẻo bằng cách ngăn không cho các đối tượng đề cập đến nhau một cách rõ ràng và nó cho phép bạn thay đổi sự tương tác của họ một cách độc lập”.

Mediator Patern (mô hình trung gian) được sử dụng để giảm sự phức tạp trong “giao tiếp” giữa các lớp và các đối tượng. Mô hình này cung cấp một lớp trung gian có nhiệm vụ xử lý thông tin liên lạc giữa các tầng lớp, hỗ trợ bảo trì mã code dễ dàng bằng cách khớp nối lỏng lẻo.

Khớp nối lỏng lẻo ở đây được hiểu là các đối tượng tương đồng không “giao tiếp” trực tiếp với nhau mà giao tiếp thông qua người trung gian, và nó cho phép thay thay đổi cách tương tác giữa chúng một cách độc lập.

Mediator Patern thúc đẩy mối quan hệ nhiều – nhiều (many-to-many) giữa các đối tượng tượng với nhau để đạt đến được kết quả mong muốn.



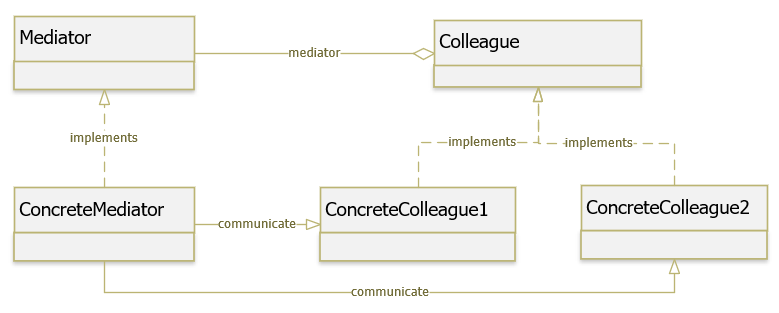
Hình bên trên là một ví dụ về mẫu thiết kế Mediator. Theo hình ta đang có một website với 4 trang. Website cho phép khách hàng duyệt qua kho hàng và đặt mua. Khách hàng có thể đi  từ trang này qua trang khác theo đường vẽ trên hình. Ở đây có một vấn đề phát sinh. Tại từng trang, bạn phải viết mã để nhận biết khi nào khách hàng muốn nhảy qua trang khác và kích hoạt trang đó. Tại một trang bạn có quá nhiều đường để đi tới trang khác, và vì vậy sẽ phát sinh nhiều đoạn code trùng lặp trên nhiều trang khác nhau.

Chúng ta có thể sử dụng mẫu Mediator để đóng gói tất cả các đường dẫn tới trang vào một module duy nhất, và đặt nó vào trong một đối tượng Mediator. Từ bây giờ, từng trang chỉ cần phải thông báo bất cứ sự thay đổi nào cho Mediator, và Mediator tự biết điều hướng trang cần thiết cho khách hàng, như trong hình bên dưới.



Chúng ta có thể tạo ra một Mediator với chức năng điều hướng trang. Tại đây chúng ta có thể chỉnh sửa và thay đổi dễ dàng. Đó chính là chức năng của Mediator (Người trung gian).

## Cài đặt Mediator Pattern như thế nào?



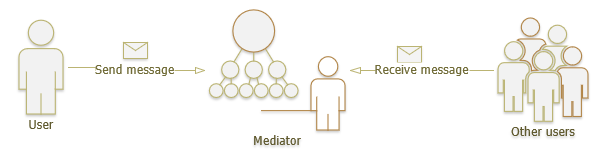
Các thành phần tham gia Mediator Pattern:

* Colleague : là một abstract class, giữ tham chiếu đến Mediator object.
* ConcreteColleague : cài đặt các phương thức của Colleague. Giao tiếp thông qua Mediator khi cần giao tiếp với Colleague khác.
* Mediator : là một interface, định nghĩa các phương thức để giao tiếp với các Colleague object.
* ConcreteMediator : cài đặt các phương thức của Mediator, biết và quản lý các Colleague object.

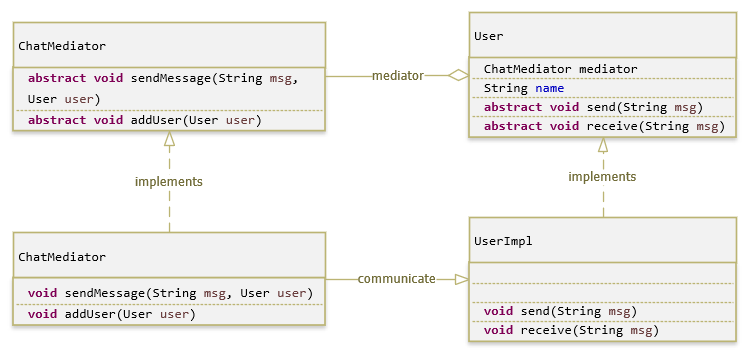
### Ví dụ Mediator Pattern trong hệ thống chat

Trong một ứng dụng chat, một user sẽ có thể send và recieve message. Khi một user muốn send message đến group thì user đó phải tìm xem tất cả những người đang online hoặc trong trạng thái có thể message để send. Nếu bình thường user gửi tin phải tự làm hết mọi thứ, phải tự kiểm tra từng member và thực hiện việc gửi tin nhắn.

Khi sử dụng Mediator: user không cần tự kiểm tra, không quan tâm ai có thể nhận message, user chỉ việc gửi thông tin đến Mediator của group. Mediator sẽ tự điều phối message này đến người nhận.



Chương trình của chúng ta như sau:



User.java

package com.gpcoder.patterns.behavioral.mediator.chat;

/\*\*

\* Colleague

\*/

public abstract class User {

protected ChatMediator mediator;

protected String name;

public User(ChatMediator med, String name) {

this.mediator = med;

this.name = name;

}

public abstract void send(String msg);

public abstract void receive(String msg);

@Override

public int hashCode() {

return name.hashCode();

}

@Override

public boolean equals(Object obj) {

if (obj == null) {

return false;

}

if (this.getClass() != obj.getClass()) {

return false;

}

User user = (User) obj;

return name.equals(user.name);

}

}

UserImpl.java

package com.gpcoder.patterns.behavioral.mediator.chat;

/\*\*

\* ConcreteColleague

\*/

public class UserImpl extends User {

public UserImpl(ChatMediator mediator, String name) {

super(mediator, name);

}

@Override

public void send(String msg) {

System.out.println("---");

System.out.println(this.name + " is sending the message: " + msg);

mediator.sendMessage(msg, this);

}

@Override

public void receive(String msg) {

System.out.println(this.name + " received the message: " + msg);

}

}

ChatMediator.java

package com.gpcoder.patterns.behavioral.mediator.chat;

public interface ChatMediator {

void sendMessage(String msg, User user);

void addUser(User user);

}

ChatMediatorImpl.java

package com.gpcoder.patterns.behavioral.mediator.chat;

import java.util.ArrayList;

import java.util.List;

public class ChatMediatorImpl implements ChatMediator {

public ChatMediatorImpl(String groupName) {

System.out.println(groupName + " group already created");

}

private List<User> users = new ArrayList<>();

@Override

public void addUser(User user) {

System.out.println(user.name + " joined this group");

this.users.add(user);

}

@Override

public void sendMessage(String msg, User user) {

for (User u : this.users) {

if (!u.equals(user)) {

u.receive(msg);

}

}

}

}

ChatClient.java

package com.gpcoder.patterns.behavioral.mediator.chat;

public class ChatClient {

public static void main(String[] args) {

ChatMediator mediator = new ChatMediatorImpl("Java design pattern");

User admin = new UserImpl(mediator, "GP Coder");

User user1 = new UserImpl(mediator, "User 1");

User user2 = new UserImpl(mediator, "User 2");

User user3 = new UserImpl(mediator, "User 3");

mediator.addUser(admin);

mediator.addUser(user1);

mediator.addUser(user2);

mediator.addUser(user3);

admin.send("Hi All");

user1.send("Hi Admin");

}

}

Output của chương trình:

Java design pattern group already created

GP Coder joined this group

User 1 joined this group

User 2 joined this group

User 3 joined this group

---

GP Coder is sending the message: Hi All

User 1 received the message: Hi All

User 2 received the message: Hi All

User 3 received the message: Hi All

---

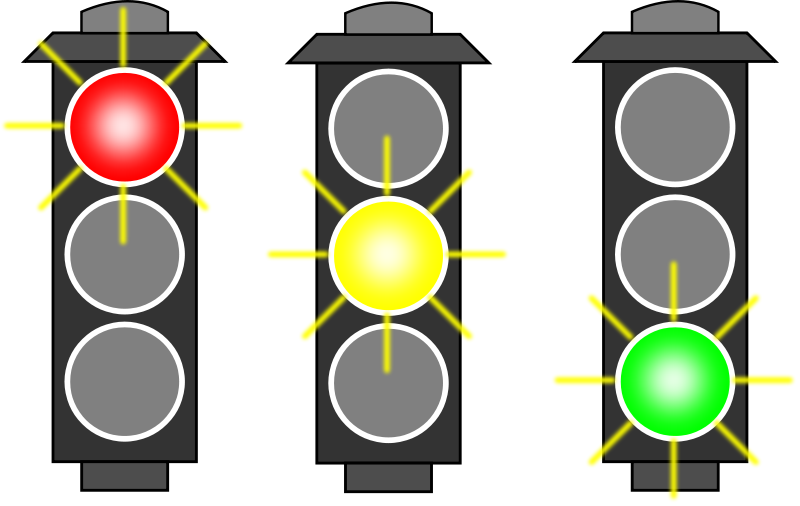
User 1 is sending the message: Hi Admin

GP Coder received the message: Hi Admin

User 2 received the message: Hi Admin

User 3 received the message: Hi Admin

### Ví dụ Mediator Pattern trong ứng dụng điều khiển đèn giao thông



Đèn giao thông có 3 màu: đỏ, vàng, xanh. Khi một đèn được mở thì những đèn còn lại sẽ tắt.

Chương trình của chúng ta như sau:

* Light: lớp chứa thông tin đèn giao thông, trạng thái (ON/OFF), giữ bộ thông tin bộ điều khiển tín hiệu đèn (Mediator).
* LightMediator : bộ điều khiển đèn thông thông, nhận thông báo khi một tín hiệu đèn được mở, thực hiện tắt những đèn còn lại.
* TrafficLightApp : khởi tạo Mediator, đăng ký các Light với Mediator. Thực hiện bật các đèn theo khoảng thời gian 3 giây.

Light.java

package com.gpcoder.patterns.behavioral.mediator.trafficlight;

/\*\*

\* Colleague

\*/

public class Light {

enum State {

ON, OFF

}

private String color;

private LightMediator lightMediator;

private State currentState;

public Light(String color, LightMediator lightMediator) {

this.color = color;

this.currentState = State.OFF;

this.lightMediator = lightMediator;

lightMediator.registerLight(this);

}

public void turnOn() {

this.currentState = State.ON;

lightMediator.notifyMediator(this);

}

public void turnOff() {

this.currentState = State.OFF;

lightMediator.notifyMediator(this);

}

public String getColor() {

return color;

}

public State getCurrentState() {

return currentState;

}

@Override

public int hashCode() {

return color.hashCode();

}

@Override

public boolean equals(Object obj) {

if (obj == null) {

return false;

}

if (this.getClass() != obj.getClass()) {

return false;

}

Light light = (Light) obj;

return color.equals(light.color);

}

}

LightMediator.java

package com.gpcoder.patterns.behavioral.mediator.trafficlight;

import java.util.HashSet;

import java.util.Set;

/\*\*

\* Mediator

\*/

public class LightMediator {

private Set<Light> trafficSignals = new HashSet<>();

public void registerLight(Light light) {

trafficSignals.add(light);

}

public void unRegisterLight(Light light) {

trafficSignals.remove(light);

}

public void notifyMediator(Light light) {

System.out.printf("%s is turned %s \n", light.getColor(), light.getCurrentState());

if (light.getCurrentState() == Light.State.ON) {

turnOffAllOtherLights(light);

}

}

private void turnOffAllOtherLights(Light light) {

for (Light otherLight : trafficSignals) {

if (!(light.equals(otherLight))) {

otherLight.turnOff();

}

}

System.out.println("---");

}

}

AutoRotateTrafficLightApp.java

package com.gpcoder.patterns.behavioral.mediator.trafficlight;

import java.util.concurrent.TimeUnit;

public class AutoRotateTrafficLightApp {

public static void main(String[] args) {

LightMediator lightMediator = new LightMediator();

Light[] lights = {

new Light("Red", lightMediator),

new Light("Green", lightMediator),

new Light("Yellow", lightMediator)

};

int currentLightIndex = 0;

Light light;

while (true) {

if (currentLightIndex >= lights.length) {

currentLightIndex = 0;

}

light = lights[currentLightIndex];

light.turnOn();

timer();

currentLightIndex++;

}

}

private static void timer() {

try {

TimeUnit.SECONDS.sleep(3);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

Output của chương trình như sau:

Red is turned ON

Yellow is turned OFF

Green is turned OFF

---

Green is turned ON

Red is turned OFF

Yellow is turned OFF

---

Yellow is turned ON

Red is turned OFF

Green is turned OFF

---

Red is turned ON

Yellow is turned OFF

Green is turned OFF

---

....

## Lợi ích của Mediator Pattern là gì?

* Đảm bảo nguyên tắc [Single responsibility principle (SRP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Single_responsibility_principle_SRP) : chúng ta có thể tách phần giao tiếp giữa các thành phần (component) ra một nơi khác.
* Đảm bảo nguyên tắc [Open/Closed Principle (OCP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Open-Closed_principle_OCP) : chúng ta có thể implement thêm một Mediator mới mà không ảnh hưởng đến các component hiện có.
* Giảm khớp nối giữa các component.
* Tái sử dụng các component dễ dàng hơn.
* Đơn giản hóa cách giao tiếp giữa các đối tượng. Một mediator sẽ thay thế mối quan hệ nhiều-nhiều (many-to-many) giữa các component bằng quan hệ một-nhiều (one-to-many) giữa một mediator với các component.
* Quản lý tập trung, giúp làm rõ các component tương tác trong hệ thống như thế nào trong hệ thống.

## Sử dụng Mediator Pattern khi nào?

* Khi tập hợp các đối tượng giao tiếp theo những cách thức được xác định rõ ràng nhưng cách thức đó quá phức tạp. Sự phụ thuộc lẫn nhau giữa các đối tượng tạo ra kết quả là cách tổ chức không có cấu trúc và khó hiểu.
* Khi cần tái sử dụng một đối tượng nhưng rất khó khăn vì nó tham chiếu và giao tiếp với nhiều đối tượng khác.
* Điều chỉnh hành vi giữa các lớp một cách dễ dàng, không cần chỉnh sửa ở nhiều lớp.
* Thường được sử dụng trong các hệ thống truyền thông điệp (message-based system), chẳng hạn như hệ thống chat.
* Khi giao tiếp giữa các object trong hệ thống quá phức tạp, có quá nhiều quan hệ giữa các object trong hệ thống. Một điểm chung để kiểm soát hoặc giao tiếp là cần thiết.

## So sánh Mediator Pattern với các Pattern khác

Các Pattern như [Chain of Responsibility](https://gpcoder.com/4665-huong-dan-java-design-pattern-chain-of-responsibility/), [Command](https://gpcoder.com/4686-huong-dan-java-design-pattern-command/), Mediator, Observer: giải quyết nhiều cách khác nhau để kết nối người gửi và người nhận yêu cầu.

* Chain of Responsibility : nhận yêu cầu của người gửi, và gửi yêu cầu đó (request) dọc theo một chuỗi những người nhận tiềm năng (processor) cho đến khi một trong số chúng xử lý nó.
* Command : thiết lập các kết nối đơn hướng giữa người nhận với người gửi với một phân lớp.
* Mediator : loại bỏ các kết nối trực tiếp giữa người gửi và người nhận, buộc chúng phải liên lạc gián tiếp thông qua một đối tượng Mediator.
* Observer : định nghĩa một interface tách biệt cho phép nhiều người nhận đăng ký và hủy đăng ký nhận yêu cầu tại thời điểm run-time.

Mediator tương tự như [Facade](https://gpcoder.com/4604-huong-dan-java-design-pattern-facade/), chúng cố gắng tổ chức sự hợp tác giữa nhiều class được liên kết chặt chẽ.

* Facade : định nghĩa một interface đơn giản hóa cho một hệ thống con của các đối tượng, nhưng nó không giới thiệu bất kỳ chức năng mới nào. Bản thân hệ thống con không biết về Facade. Các đối tượng trong hệ thống con có thể giao tiếp trực tiếp với nhau.
* Mediator : tập trung giao tiếp giữa các thành phần (component) của hệ thống. Các thành phần chỉ biết về đối tượng Mediator và không thể giao tiếp trực tiếp với nhau.

Sự khác biệt giữa Mediator và Observer là Observer phân phối giao tiếp bằng cách giới thiệu các đối tượng “quan sát viên (observer)” và “chủ thể (subject)”. Trong khi đó, một đối tượng Mediator đóng gói giao tiếp giữa các đối tượng khác. Thông thường, việc tạo ra các Observer và Subject có thể tái sử dụng dễ dàng hơn so với việc tạo các Mediator có thể tái sử dụng. Mặt khác, Mediator giải có thể tận dụng Observer để đăng ký động các thành phần và liên lạc với chúng.

Tuy nhiên, sự khác biệt giữa Mediator và Observer thường không rõ ràng. Trong hầu hết các trường hợp, chúng ta có thể thực hiện một trong hai mẫu này, nhưng đôi khi có thể áp dụng cả hai cùng một lúc.

* Mục tiêu chính của Mediator là loại bỏ sự phụ thuộc lẫn nhau giữa một tập hợp các thành phần hệ thống. Thay vào đó, các thành phần này trở nên phụ thuộc vào một đối tượng Mediator. Mục tiêu của Observer là thiết lập các kết nối một chiều động giữa các đối tượng, trong đó một số đối tượng đóng vai trò là cấp dưới của những người khác.
* Có một triển khai phổ biến của mẫu Mediator dựa trên Observer. Đối tượng Mediator đóng vai trò là nhà xuất bản (publisher) và các thành phần đóng vai trò là người đăng ký (subscriber) có thể đăng ký và hủy đăng ký khỏi các sự kiện của Mediator. Khi Mediator được triển khai theo cách này, nó có thể trông rất giống với Observer.

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/mediator>
* <https://refactoring.guru/design-patterns/mediator>
* <https://www.javatpoint.com/mediator-pattern>
* <https://www.tutorialspoint.com/design_pattern/mediator_pattern.htm>
* <https://codepumpkin.com/mediator-design-pattern/>
* Design Patterns: Elements of Reusable Object-Oriented Software – GOF
* Design Pattern for dummies

# Hướng dẫn Java Design Pattern – Memento

Đăng vào 25/12/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 1088 Lượt xem

Đôi khi chúng ta cần phải ghi lại trạng thái bên trong của một đối tượng. Điều này là bắt buộc khi thực hiện tại các điểm kiểm tra và cung cấp cơ chế hoàn tác cho phép người dùng có thể khôi phục từ các lỗi. Chúng ta phải lưu thông tin trạng thái ở đâu đó để có thể khôi phục các đối tượng về trạng thái trước đó của chúng. Nhưng các đối tượng thường đóng gói một phần hoặc tất cả trạng thái của chúng, khiến nó không thể truy cập được vào các đối tượng khác và không thể lưu ở bên ngoài. Public các trạng thái này sẽ vi phạm nguyên tắc đóng gói, có thể làm giảm độ tin cậy và khả năng mở rộng của ứng dụng. Trong những trường hợp như vậy chúng ta có thể nghĩ đến Memento Pattern, nó sẽ giúp chúng ta giải quyết vấn đề này.

Nội dung [[Ẩn](https://gpcoder.com/4763-huong-dan-java-design-pattern-memento/)]

* [1 Memento Pattern là gì?](https://gpcoder.com/4763-huong-dan-java-design-pattern-memento/" \l "Memento_Pattern_la_gi)
* [2 Cài đặt Memento Pattern như thế nào?](https://gpcoder.com/4763-huong-dan-java-design-pattern-memento/" \l "Cai_dat_Memento_Pattern_nhu_the_nao)
  + [2.1 Ví dụ Memento Pattern quản lý trạng thái của một đối tượng](https://gpcoder.com/4763-huong-dan-java-design-pattern-memento/" \l "Vi_du_Memento_Pattern_quan_ly_trang_thai_cua_mot_doi_tuong)
  + [2.2 Ví dụ Memento Pattern với ứng dụng quản lý tọa độ các điểm ảnh](https://gpcoder.com/4763-huong-dan-java-design-pattern-memento/" \l "Vi_du_Memento_Pattern_voi_ung_dung_quan_ly_toa_do_cac_diem_anh)
* [3 Lợi ích của Memento Pattern là gì?](https://gpcoder.com/4763-huong-dan-java-design-pattern-memento/" \l "Loi_ich_cua_Memento_Pattern_la_gi)
* [4 Sử dụng Memento Pattern khi nào?](https://gpcoder.com/4763-huong-dan-java-design-pattern-memento/" \l "Su_dung_Memento_Pattern_khi_nao)

## Memento Pattern là gì?

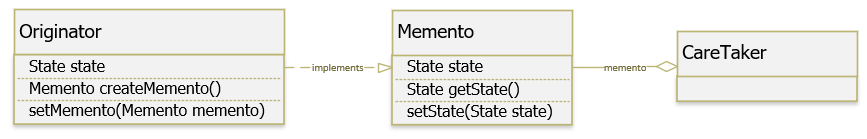
Without violating encapsulation, capture and externalize an object’s internal state so that the object can be returned to this state later.

Memento là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Memento là mẫu thiết kế có thể lưu lại trạng thái của một đối tượng để khôi phục lại sau này mà không vi phạm nguyên tắc đóng gói.

Dữ liệu trạng thái đã lưu trong đối tượng memento không thể truy cập bên ngoài đối tượng được lưu và khôi phục. Điều này bảo vệ tính toàn vẹn của dữ liệu trạng thái đã lưu.

Hoàn tác (Undo) hoặc ctrl + z là một trong những thao tác được sử dụng nhiều nhất trong trình soạn thảo văn bản (editor). Mẫu thiết kế Memento được sử dụng để thực hiện thao tác Undo. Điều này được thực hiện bằng cách lưu trạng thái hiện tại của đối tượng mỗi khi nó thay đổi trạng thái, từ đó chúng ta có thể khôi phục nó trong mọi trường hợp có lỗi.

## Cài đặt Memento Pattern như thế nào?



Các thành phần tham gia mẫu Memento:

* Originator : đại diện cho đối tượng mà chúng ta muốn lưu. Nó sử dụng memento để lưu và khôi phục trạng thái bên trong của nó.
* Caretaker : Nó không bao giờ thực hiện các thao tác trên nội dung của memento và thậm chí nó không kiểm tra nội dung. Nó giữ đối tượng memento và chịu trách nhiệm bảo vệ an toàn cho các đối tượng. Để khôi phục trạng thái trước đó, nó trả về đối tượng memento cho Originator.
* Memento : đại diện cho một đối tượng để lưu trữ trạng thái của Originator. Nó bảo vệ chống lại sự truy cập của các đối tượng khác ngoài Originator.
  + Lớp Memento cung cấp 2 interfaces: 1 interface cho Caretaker và 1 cho Originator. Interface Caretaker không được cho phép bất kỳ hoạt động hoặc bất kỳ quyền truy cập vào trạng thái nội bộ được lưu trữ bởi memento và do đó đảm bảo nguyên tắc đóng gói. Interface Originator cho phép nó truy cập bất kỳ biến trạng thái nào cần thiết để có thể khôi phục trạng thái trước đó.
  + Lớp Memento thường là một lớp bên trong của Originator. Vì vậy, originator có quyền truy cập vào các trường của memento, nhưng các lớp bên ngoài không có quyền truy cập vào các trường này.

### Ví dụ Memento Pattern quản lý trạng thái của một đối tượng

Ví dụ đơn giản bên dưới cho phép chúng ta lưu trữ trạng thái của một đối tượng và có thể phục hồi lại trạng thái của nó tại một thời điểm đã được lưu trữ.

package com.gpcoder.patterns.behavioral.memento.state;

import java.util.ArrayList;

import java.util.List;

class Originator {

private String state;

public void set(String state) {

System.out.println("Originator: Setting state to " + state);

this.state = state;

}

public Memento saveToMemento() {

System.out.println("Originator: Saving to Memento.");

return new Memento(this.state);

}

public void restoreFromMemento(Memento memento) {

this.state = memento.getSavedState();

System.out.println("Originator: State after restoring from Memento: " + state);

}

public static class Memento {

private final String state;

public Memento(String stateToSave) {

state = stateToSave;

}

public String getSavedState() {

return state;

}

}

}

class MementoExample {

public static void main(String[] args) {

List<Originator.Memento> savedStates = new ArrayList<>(); // caretaker

Originator originator = new Originator();

originator.set("State #1");

originator.set("State #2");

savedStates.add(originator.saveToMemento());

originator.set("State #3");

savedStates.add(originator.saveToMemento());

originator.set("State #4");

originator.restoreFromMemento(savedStates.get(1)); // This point need roll back

}

}

Output của chương trình:

Originator: Setting state to State #1

Originator: Setting state to State #2

Originator: Saving to Memento.

Originator: Setting state to State #3

Originator: Saving to Memento.

Originator: Setting state to State #4

Originator: State after restoring from Memento: State #3

### Ví dụ Memento Pattern với ứng dụng quản lý tọa độ các điểm ảnh

Trong ví dụ bên dưới chúng ta sẽ tách biệt các thành phần của Memento Pattern ra từng class riêng lẻ để tiện quản lý. Chương trình cho phép chúng ta có thể khôi phục lại dữ liệu tại một thời điểm đã lưu trữ trước đó.

Originator.java

package com.gpcoder.patterns.behavioral.memento.point;

public class Originator {

private double x;

private double y;

public Originator(double x, double y) {

this.x = x;

this.y = y;

}

public double getX() {

return x;

}

public double getY() {

return y;

}

public void setX(double x) {

this.x = x;

}

public void setY(double y) {

this.y = y;

}

public Memento save() {

return new Memento(this.x, this.y);

}

public void undo(Memento mem) {

this.x = mem.getX();

this.y = mem.getY();

}

@Override

public String toString() {

return "X: " + x + ", Y: " + y;

}

}

Memento.java

package com.gpcoder.patterns.behavioral.memento.point;

public class Memento {

private double x;

private double y;

public Memento(double x, double y) {

this.x = x;

this.y = y;

}

public double getX() {

return x;

}

public double getY() {

return y;

}

}

CareTaker.java

package com.gpcoder.patterns.behavioral.memento.point;

import java.util.HashMap;

import java.util.Map;

public class CareTaker {

private final Map<String, Memento> savepointStorage = new HashMap<>();

public void saveMemento(Memento memento, String savedPointName) {

System.out.println("Saving state..." + savedPointName);

savepointStorage.put(savedPointName, memento);

}

public Memento getMemento(String savedPointName) {

System.out.println("Undo at ..." + savedPointName);

return savepointStorage.get(savedPointName);

}

public void clearSavepoints() {

System.out.println("Clearing all save points...");

savepointStorage.clear();

}

}

MementoPatternExample.java

package com.gpcoder.patterns.behavioral.memento.point;

public class MementoPatternExample {

public static void main(String[] args) {

CareTaker careTaker = new CareTaker();

Originator originator = new Originator(5, 10);

originator.setX(originator.getY() \* 51);

System.out.println("State initial: " + originator);

careTaker.saveMemento(originator.save(), "SAVE #1");

originator.setY(originator.getX() / 22);

System.out.println("State changed: " + originator);

originator.undo(careTaker.getMemento("SAVE #1"));

System.out.println("State after undo: " + originator);

originator.setX(Math.pow(originator.getX(), 3));

careTaker.saveMemento(originator.save(), "SAVE #2");

System.out.println("State changed: " + originator);

originator.setY(originator.getX() - 30);

careTaker.saveMemento(originator.save(), "SAVE #3");

System.out.println("State saved #3: " + originator);

originator.setY(originator.getX() / 22);

careTaker.saveMemento(originator.save(), "SAVE #4");

System.out.println("State saved #4: " + originator);

originator.undo(careTaker.getMemento("SAVE #2"));

System.out.println("Retrieving at saved #2: " + originator);

}

}

Output của chương trình:

State initial: X: 510.0, Y: 10.0

Saving state...SAVE #1

State changed: X: 510.0, Y: 23.181818181818183

Undo at ...SAVE #1

State after undo: X: 510.0, Y: 10.0

Saving state...SAVE #2

State changed: X: 1.32651E8, Y: 10.0

Saving state...SAVE #3

State saved #3: X: 1.32651E8, Y: 1.3265097E8

Saving state...SAVE #4

State saved #4: X: 1.32651E8, Y: 6029590.909090909

Undo at ...SAVE #2

Retrieving at saved #2: X: 1.32651E8, Y: 10.0

## Lợi ích của Memento Pattern là gì?

Lợi ích:

* Bảo bảo nguyên tắc đóng gói: sử dụng trực tiếp trạng thái của đối tượng có thể làm lộ thông tin chi tiết bên trong đối tượng và vi phạm nguyên tắc đóng gói.
* Đơn giản code của Originator bằng cách để Memento lưu giữ trạng thái của Originator và Caretaker quản lý lịch sử thay đổi của Originator.

Một số vấn đề cần xem xét khi sử dụng Memento Pattern:

* Khi có một số lượng lớn Memento được tạo ra có thể gặp vấn đề về bộ nhớ, performance của ứng dụng.
* Khó đảm bảo trạng thái bên trong của Memento không bị thay đổi.

## Sử dụng Memento Pattern khi nào?

* Các ứng dụng cần chức năng cần Undo/ Redo: lưu trạng thái của một đối tượng bên ngoài và có thể restore/ rollback sau này.
* Thích hợp với các ứng dụng cần quản lý transaction.

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/memento>
* <https://refactoring.guru/design-patterns/memento>
* <https://www.javatpoint.com/memento-pattern>
* <https://www.javacodegeeks.com/2015/09/memento-design-pattern.html>
* Design Patterns: Elements of Reusable Object-Oriented Software – GOF
* Design Pattern for dummies

# Hướng dẫn Java Design Pattern – Observer

Đăng vào 28/12/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 2912 Lượt xem

Chúng ta không thể nói về Lập trình hướng đối tượng mà không xem xét trạng thái của các đối tượng. Tất cả các chương trình hướng đối tượng là về các đối tượng và sự tương tác của chúng. Trong trường hợp khi một số đối tượng nhất định cần được thông báo thường xuyên về những thay đổi xảy ra trong các đối tượng khác. Để có một thiết kế tốt có nghĩa là tách rời càng nhiều càng tốt và giảm sự phụ thuộc. Mẫu thiết kế Observer (quan sát) có thể được sử dụng bất cứ khi nào mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.

Nội dung [[Ẩn](https://gpcoder.com/4747-huong-dan-java-design-pattern-observer/)]

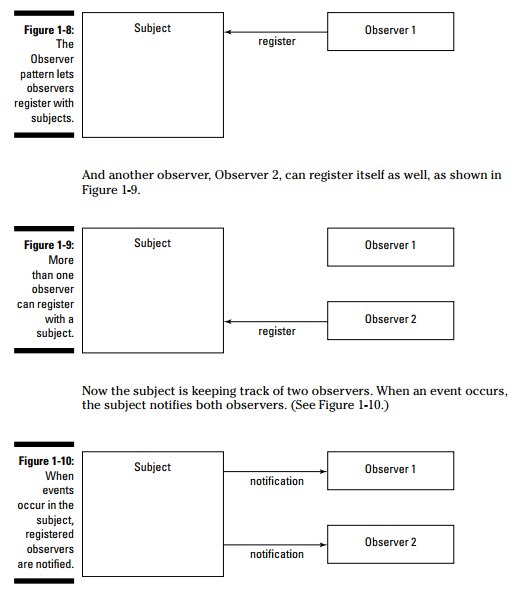
* [1 Observer Pattern là gì?](https://gpcoder.com/4747-huong-dan-java-design-pattern-observer/" \l "Observer_Pattern_la_gi)
* [2 Cài đặt Observer Pattern như thế nào?](https://gpcoder.com/4747-huong-dan-java-design-pattern-observer/" \l "Cai_dat_Observer_Pattern_nhu_the_nao)
  + [2.1 Ví dụ Observer Pattern với ứng dụng Tracking thao tác một Account](https://gpcoder.com/4747-huong-dan-java-design-pattern-observer/" \l "Vi_du_Observer_Pattern_voi_ung_dung_Tracking_thao_tac_mot_Account)
* [3 Lợi ích của Observer Pattern là gì?](https://gpcoder.com/4747-huong-dan-java-design-pattern-observer/" \l "Loi_ich_cua_Observer_Pattern_la_gi)
* [4 Sử dụng Observer Pattern khi nào?](https://gpcoder.com/4747-huong-dan-java-design-pattern-observer/" \l "Su_dung_Observer_Pattern_khi_nao)

## Observer Pattern là gì?

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Observer Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó định nghĩa mối phụ thuộc một – nhiều giữa các đối tượng để khi mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.

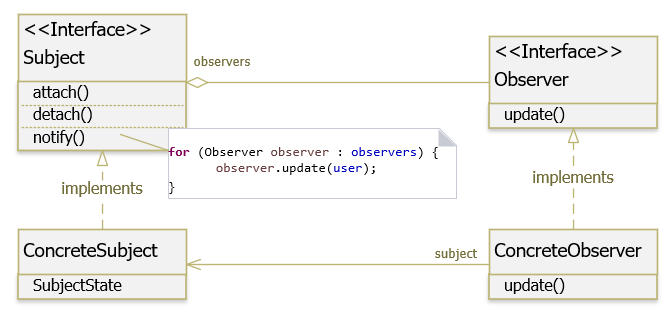
Observer có thể đăng ký với hệ thống. Khi hệ thống có sự thay đổi, hệ thống sẽ thông báo cho Observer biết. Khi không cần nữa, mẫu Observer sẽ được gỡ khỏi hệ thống.



* Hình 1-8, cho phép observer thứ 1 đăng ký với hệ thống.
* Hình 1-9, cho phép observer thứ 2 đăng ký với hệ thống.
* Hiện tại hệ thống đang liên lạc với 2 observer: Observer 1 và Observer 2. Khi hệ thống phát sinh một sự kiện cụ thể nào đó, nó sẽ thông báo (notification) với cả 2 observer như hình số 1-10.

Observer Pattern còn gọi là Dependents, Publish/Subcribe hoặc Source/Listener.

## Cài đặt Observer Pattern như thế nào?



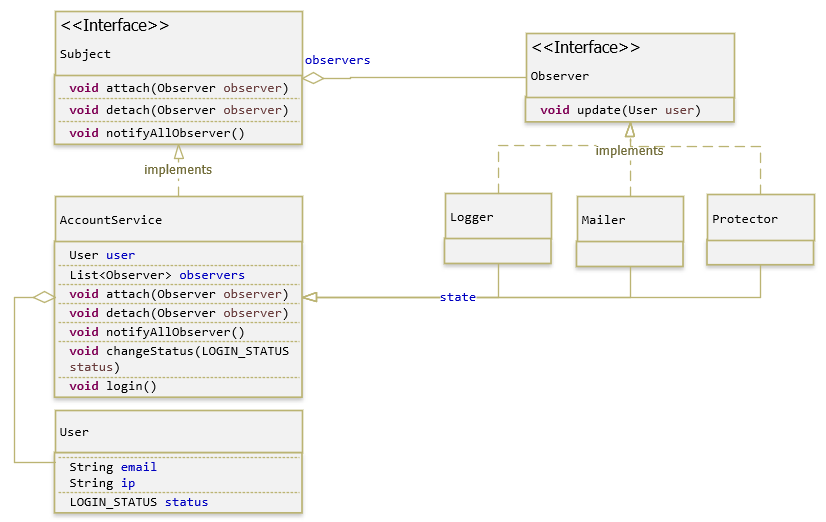
Các thành phần tham gia Observer Pattern:

* Subject : chứa danh sách các observer,  cung cấp phương thức để có thể thêm và loại bỏ observer.
* Observer : định nghĩa một phương thức update() cho các đối tượng sẽ được subject thông báo đến khi có sự thay đổi trạng thái.
* ConcreteSubject : cài đặt các phương thức của Subject, lưu trữ trạng thái danh sách các ConcreateObserver, gửi thông báo đến các observer của nó khi có sự thay đổi trạng thái.
* ConcreteObserver : cài đặt các phương thức của Observer, lưu trữ trạng thái của subject, thực thi việc cập nhật để giữ cho trạng thái đồng nhất với subject gửi thông báo đến.

Sự tương tác giữa subject và các observer như sau: mỗi khi subject có sự thay đổi trạng thái, nó sẽ duyệt qua danh sách các observer của nó và gọi phương thức cập nhật trạng thái ở từng observer, có thể truyền chính nó vào phương thức để các observer có thể lấy ra trạng thái của nó và xử lý.

### Ví dụ Observer Pattern với ứng dụng Tracking thao tác một Account

Giả sử hệ thống của chúng ta cần theo dõi về tài khoản của người dùng. Mọi thao tác của người dùng đều cần được ghi log lại, sẽ thực hiện gửi mail thông báo khi tài khoản hết hạn, thực hiện chặn người dùng nếu truy cập không hợp lệ, …



Chương trình của chúng ta như sau:

* Subject : cung cấp các phương thức để thêm, loại bỏ, thông báo observer.
* AccountService : đóng vai trò là ConcreteSubject, sẽ thông báo tới tất cả các observers bất cứ khi nào có thao tác của người dùng liên quan đến đăng nhập, tài khoản hết hạn.
* Observer : định nghĩa một phương thức update() cho các đối tượng sẽ được subject thông báo đến khi có sự thay đổi trạng thái. Phương thức này chấp nhận đối số là SubjectState, cho phép các ConcreteObserver sử dụng dữ liệu của nó.
* Logger, Mailer và Protector là các ConcreteObserver. Sau khi nhận được thông báo rằng có thao tác với user và gọi tới phương thức update(), các ConcreteObserver sẽ sử dụng dữ liệu SubjectState để xử lý.

Subject.java

package com.gpcoder.patterns.behavioral.observer.account;

public interface Subject {

void attach(Observer observer);

void detach(Observer observer);

void notifyAllObserver();

}

AccountService.java

package com.gpcoder.patterns.behavioral.observer.account;

import java.util.ArrayList;

import java.util.List;

import lombok.Data;

enum LoginStatus {

SUCCESS, FAILURE, INVALID, EXPIRED

}

@Data

class User {

private String email;

private String ip;

private LoginStatus status;

}

public class AccountService implements Subject {

private User user;

private List<Observer> observers = new ArrayList<>();

public AccountService(String email, String ip) {

user = new User();

user.setEmail(email);

user.setIp(ip);

}

@Override

public void attach(Observer observer) {

if (!observers.contains(observer))

observers.add(observer);

}

@Override

public void detach(Observer observer) {

if (observers.contains(observer)) {

observers.remove(observer);

}

}

@Override

public void notifyAllObserver() {

for (Observer observer : observers) {

observer.update(user);

}

}

public void changeStatus(LoginStatus status) {

user.setStatus(status);

System.out.println("Status is changed");

this.notifyAllObserver();

}

public void login() {

if (!this.isValidIP()) {

user.setStatus(LoginStatus.INVALID);

} else if (this.isValidEmail()) {

user.setStatus(LoginStatus.SUCCESS);

} else {

user.setStatus(LoginStatus.FAILURE);

}

System.out.println("Login is handled");

this.notifyAllObserver();

}

private boolean isValidIP() {

return "127.0.0.1".equals(user.getIp());

}

private boolean isValidEmail() {

return "contact@gpcoder.com".equalsIgnoreCase(user.getEmail());

}

}

Observer.java

package com.gpcoder.patterns.behavioral.observer.account;

public interface Observer {

void update(User user);

}

Logger.java

package com.gpcoder.patterns.behavioral.observer.account;

public class Logger implements Observer {

@Override

public void update(User user) {

System.out.println("Logger: " + user);

}

}

Mailer.java

package com.gpcoder.patterns.behavioral.observer.account;

public class Mailer implements Observer {

@Override

public void update(User user) {

if (user.getStatus() == LoginStatus.EXPIRED) {

System.out.println("Mailer: User " + user.getEmail() + " is expired. An email was sent!");

}

}

}

Protector.java

package com.gpcoder.patterns.behavioral.observer.account;

public class Protector implements Observer {

@Override

public void update(User user) {

if (user.getStatus() == LoginStatus.INVALID) {

System.out.println("Protector: User " + user.getEmail() + " is invalid. "

+ "IP " + user.getIp() + " is blocked");

}

}

}

ObserverPatternExample.java

package com.gpcoder.patterns.behavioral.observer.account;

public class ObserverPatternExample {

public static void main(String[] args) {

AccountService account1 = createAccount("contact@gpcoder.com", "127.0.0.1");

account1.login();

account1.changeStatus(LoginStatus.EXPIRED);

System.out.println("---");

AccountService account2 = createAccount("contact@gpcoder.com", "116.108.77.231");

account2.login();

}

private static AccountService createAccount(String email, String ip) {

AccountService account = new AccountService(email, ip);

account.attach(new Logger());

account.attach(new Mailer());

account.attach(new Protector());

return account;

}

}

Output của chương trình:

Login is handled

Logger: User(email=contact@gpcoder.com, ip=127.0.0.1, status=SUCCESS)

Status is changed

Logger: User(email=contact@gpcoder.com, ip=127.0.0.1, status=EXPIRED)

Mailer: User contact@gpcoder.com is expired. An email was sent!

---

Login is handled

Logger: User(email=contact@gpcoder.com, ip=116.108.77.231, status=INVALID)

Protector: User contact@gpcoder.com is invalid. IP 116.108.77.231 is blocked

## Lợi ích của Observer Pattern là gì?

Lợi ích:

* Dễ dàng mở rộng với ít sự thay đổi : mẫu này cho phép thay đổi Subject và Observer một cách độc lập. Chúng ta có thể tái sử dụng các Subject mà không cần tái sử dụng các Observer và ngược lại. Nó cho phép thêm Observer mà không sửa đổi Subject hoặc Observer khác. Vì vậy, nó đảm bảo nguyên tắc [Open/Closed Principle (OCP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Open-Closed_principle_OCP).
* Sự thay đổi trạng thái ở 1 đối tượng có thể được thông báo đến các đối tượng khác mà không phải giữ chúng liên kết quá chặt chẽ.
* Một đối tượng có thể thông báo đến một số lượng không giới hạn các đối tượng khác.

Bên cạnh những lợi ích, chúng ta cần xem xét đến trường hợp cập nhật không mong muốn (Unexpected update) của Subject. Bởi vì các Observer không biết về sự hiện diện của nhau, nó có thể gây tốn nhiều chi phí của việc thay đổi Subject.

## Sử dụng Observer Pattern khi nào?

* Thường được sử dụng trong mối quan hệ 1-n giữa các object với nhau. Trong đó một đối tượng thay đổi và muốn thông báo cho tất cả các object liên quan biết về sự thay đổi đó.
* Khi thay đổi một đối tượng, yêu cầu thay đổi đối tượng khác và chúng ta không biết có bao nhiêu đối tượng cần thay đổi, những đối tượng này là ai.
* Sử dụng trong ứng dụng broadcast-type communication.
* Sử dụng để quản lý sự kiện (Event management).
* Sử dụng trong mẫu mô hình MVC (Model View Controller Pattern) : trong MVC, mẫu này được sử dụng để tách Model khỏi View. View đại diện cho Observer và Model là đối tượng Observable.

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/observer>
* <https://refactoring.guru/design-patterns/observer>
* <https://www.baeldung.com/java-observer-pattern>
* <https://pawelgrzybek.com/the-observer-pattern-in-javascript-explained/>
* Design Patterns: Elements of Reusable Object-Oriented Software – GOF
* Design Pattern for dummies

# Hướng dẫn Java Design Pattern – State

Đăng vào 31/12/2018 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 1806 Lượt xem

Trong các ứng dụng, một số đối tượng có thông tin về trạng thái. Hành vi của đối tượng phụ thuộc vào trạng thái của nó tại thời điểm thực thi (run-time) và các phương thức xử lý nghiệp vụ có thể thay đổi trạng thái buộc đối tượng có hành vi xử lý khác đi. Trong trường hợp như vậy, chúng ta có thể sử dụng State Pattern.

Nội dung [[Ẩn](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/)]

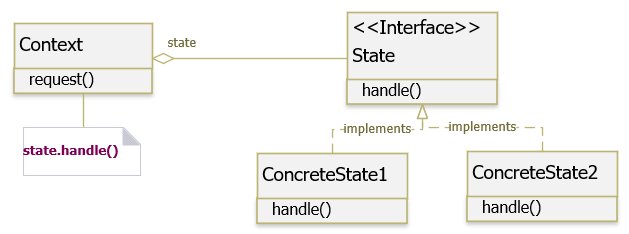
* [1 State Pattern là gì?](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/" \l "State_Pattern_la_gi)
* [2 Cài đặt State Pattern như thế nào?](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/" \l "Cai_dat_State_Pattern_nhu_the_nao)
  + [2.1 Ví dụ State Pattern](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/" \l "Vi_du_State_Pattern)
* [3 Lợi ích của State Pattern là gì?](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/" \l "Loi_ich_cua_State_Pattern_la_gi)
* [4 Sử dụng State Pattern khi nào?](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/" \l "Su_dung_State_Pattern_khi_nao)

## State Pattern là gì?

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

State Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó cho phép một đối tượng thay đổi hành vi của nó khi trạng thái nội bộ của nó thay đổi. Đối tượng sẽ xuất hiện để thay đổi lớp của nó.

## Cài đặt State Pattern như thế nào?



Các thành phần tham gia State Pattern:

* Context : được sử dụng bởi Client. Client không truy cập trực tiếp đến State của đối tượng. Lớp Context này chứa thông tin của ConcreteState object, cho hành vi nào tương ứng với trạng thái nào hiện đang được thực hiện.
* State : là một interface hoặc abstract class xác định các đặc tính cơ bản của tất cả các đối tượng ConcreteState. Chúng sẽ được sử dụng bởi đối tượng Context để truy cập chức năng có thể thay đổi.
* ConcreteState : cài đặt các phương thức của State. Mỗi ConcreteState có thể thực hiện logic và hành vi của riêng nó tùy thuộc vào Context.

Một vài điểm cần ghi nhớ khi áp dụng pattern này:

* Một đối tượng nên thay đổi hành vi của nó khi trạng thái bên trong của nó thay đổi.
* Mỗi State nên được xác định độc lập.
* Thêm các trạng thái mới sẽ không làm ảnh hưởng đến các trạng thái hoặc chức năng khác.

### Ví dụ State Pattern

Giả sử chúng ta cần xây dựng một ứng dụng quản lý Document. Một Document có thể bao gồm các trạng thái: tạo mới (New), trình phê duyệt (Submitted), phê duyệt (Approved) và từ chối (Rejected).

Với yêu cầu trên, chương trình của chúng ta như sau:

package com.gpcoder.patterns.behavioral.state.document.bad;

enum DocumentState {

NEW, SUBMITTED, APPROVED, REJECTED

}

class DocumentService {

private DocumentState state;

public void setState(DocumentState state) {

this.state = state;

}

public void handleRequest() {

switch (state) {

case NEW:

System.out.println("Create a new document");

break;

case SUBMITTED:

System.out.println("Submitted");

break;

case APPROVED:

System.out.println("Approved");

break;

case REJECTED:

System.out.println("Rejected");

break;

default:

break;

}

}

}

public class DocumentApp {

public static void main(String[] args) {

DocumentService service = new DocumentService();

service.setState(DocumentState.NEW);

service.handleRequest();

service.setState(DocumentState.SUBMITTED);

service.handleRequest();

service.setState(DocumentState.APPROVED);

service.handleRequest();

}

}

Output của chương trình:

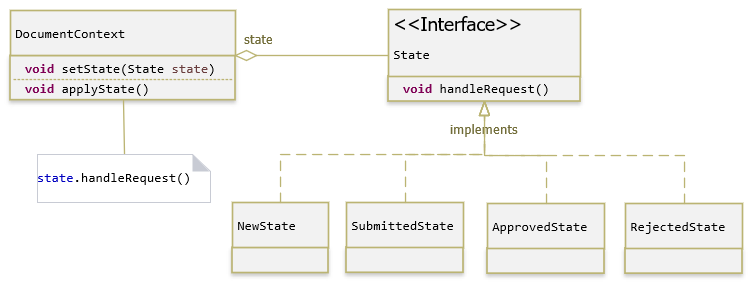
Create a new document

Submitted

Approved

Như bạn thấy chương trình trên chạy ok, không vấn đề gì. Nhưng bây giờ chúng ta muốn thêm một trạng thái mới như lưu nháp (Save Draft). Đơn giản chúng ta chỉ cần thêm vào enum một giá trị mới và thêm điều kiện xử lý trong switch-case. Tuy nhiên, nếu làm như vậy thì chúng ta đã vi phạm nguyên tắc Open/ Close. Mỗi khi có thêm một trạng thái mới chúng ta phải sửa nhiều nơi, code trong phương thức handleRequest() ngày càng nhiều và cần phải test lại toàn bộ app.

Bây giờ chúng ta hãy áp dụng State Pattern cho chương trình trên:



* Đầu tiên chúng ta sẽ tạo 1 base inteface để nhận yêu cầu xử lý. Lớp này gọi là State.
* Tiếp theo, ứng với mỗi giá trị trong enum, chúng ta sẽ tạo một class mới và implement các phương thức của State.
* Cuối cùng, chúng ta tạo một class Context. Class này chứa thông tin State hiện tại và nhận yêu cầu xử lý trực tiếp từ Client.

State.java

package com.gpcoder.patterns.behavioral.state.document.improve;

public interface State {

void handleRequest();

}

NewState.java

package com.gpcoder.patterns.behavioral.state.document.improve;

public class NewState implements State {

@Override

public void handleRequest() {

System.out.println("Create a new document");

}

}

SubmittedState.java

package com.gpcoder.patterns.behavioral.state.document.improve;

public class SubmittedState implements State {

@Override

public void handleRequest() {

System.out.println("Submitted");

}

}

ApprovedState.java

package com.gpcoder.patterns.behavioral.state.document.improve;

public class ApprovedState implements State {

@Override

public void handleRequest() {

System.out.println("Approved");

}

}

RejectedState.java

package com.gpcoder.patterns.behavioral.state.document.improve;

public class RejectedState implements State {

@Override

public void handleRequest() {

System.out.println("Rejected");

}

}

DocumentContext.java

package com.gpcoder.patterns.behavioral.state.document.improve;

public class DocumentContext {

private State state;

public void setState(State state) {

this.state = state;

}

public void applyState() {

this.state.handleRequest();

}

}

StatePatternExample.java

package com.gpcoder.patterns.behavioral.state.document.improve;

public class StatePatternExample {

public static void main(String[] args) {

DocumentContext context = new DocumentContext();

context.setState(new NewState());

context.applyState();

context.setState(new SubmittedState());

context.applyState();

context.setState(new ApprovedState());

context.applyState();

}

}

Output của chương trình:

Create a new document

Submitted

Approved

Như bạn thấy, kết quả cũng không đổi. Tuy nhiên, chúng ta rất dễ dàng mở rộng. Nếu muốn thêm một trạng thái mới như lưu tạm (Save Draft), đơn giản tạo một class mới implements từ State mà không làm ảnh hưởng đến các State khác và Context.

## Lợi ích của State Pattern là gì?

Lợi ích:

* Đảm bảo nguyên tắc [Single responsibility principle (SRP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Single_responsibility_principle_SRP) : tách biệt mỗi State tương ứng với 1 class riêng biệt.
* Đảm bảo nguyên tắc [Open/Closed Principle (OCP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Open-Closed_principle_OCP) : chúng ta có thể thêm một State mới mà không ảnh hưởng đến State khác hay Context hiện có.
* Giữ hành vi cụ thể tương ứng với trạng thái.
* Giúp chuyển trạng thái một cách rõ ràng.

## Sử dụng State Pattern khi nào?

* Khi hành vi của đối tượng phụ thuộc vào trạng thái của nó và nó phải có khả năng thay đổi hành vi của nó lúc run-time theo trạng thái mới.
* Khi nhiều điều kiện phức tạp buộc đối tượng phụ thuộc vào trạng thái của nó.

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/state>
* <https://refactoring.guru/design-patterns/state>
* <https://www.javatpoint.com/state-pattern>
* <https://www.tutorialspoint.com/design_pattern/state_pattern.htm>
* Design Patterns: Elements of Reusable Object-Oriented Software – GOF
* Design Pattern for dummies

# Hướng dẫn Java Design Pattern – Strategy

Đăng vào 03/01/2019 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 2744 Lượt xem

Có một vài trường hợp, các lớp chỉ khác nhau về hành vi của chúng. Trong trường hợp như vậy, ý tưởng tốt là chúng ta sẽ tách biệt các thuật toán trong các lớp riêng biệt để có khả năng lựa chọn các thuật toán khác nhau trong thời gian chạy (run-time). Ý tưởng này được gọi là Strategy Pattern, một pattern giúp chúng ta giải quyết vấn đề về sự thay đổi, tương tự như [State Design Pattern](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/).

Nội dung [[Ẩn](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/)]

* [1 Strategy Pattern là gì?](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/" \l "Strategy_Pattern_la_gi)
* [2 Cài đặt Strategy Pattern như thế nào?](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/" \l "Cai_dat_Strategy_Pattern_nhu_the_nao)
  + [2.1 Ví dụ Strategy Pattern với ứng dụng sắp xếp](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/" \l "Vi_du_Strategy_Pattern_voi_ung_dung_sap_xep)
  + [2.2 Ví dụ Strategy Pattern với ứng dụng thanh toán online](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/" \l "Vi_du_Strategy_Pattern_voi_ung_dung_thanh_toan_online)
* [3 Lợi ích của Strategy Pattern là gì?](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/" \l "Loi_ich_cua_Strategy_Pattern_la_gi)
* [4 Sử dụng Strategy Pattern khi nào?](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/" \l "Su_dung_Strategy_Pattern_khi_nao)
* [5 So sánh Strategy Pattern vs State Pattern](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/" \l "So_sanh_Strategy_Pattern_vs_State_Pattern)

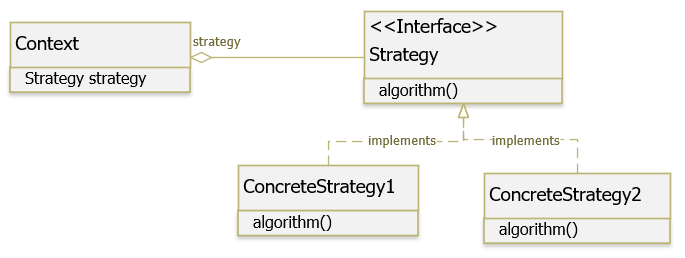
## Strategy Pattern là gì?

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

Strategy Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó cho phép định nghĩa tập hợp các thuật toán, đóng gói từng thuật toán lại, và dễ dàng thay đổi linh hoạt các thuật toán bên trong object. Strategy cho phép thuật toán biến đổi độc lập khi người dùng sử dụng chúng.

Ý nghĩa thực sự của Strategy Pattern là giúp tách rời phần xử lý một chức năng cụ thể ra khỏi đối tượng. Sau đó tạo ra một tập hợp các thuật toán để xử lý chức năng đó và lựa chọn thuật toán nào mà chúng ta thấy đúng đắn nhất khi thực thi chương trình. Mẫu thiết kế này thường được sử dụng để thay thế cho sự kế thừa, khi muốn chấm dứt việc theo dõi và chỉnh sửa một chức năng qua nhiều lớp con.

## Cài đặt Strategy Pattern như thế nào?

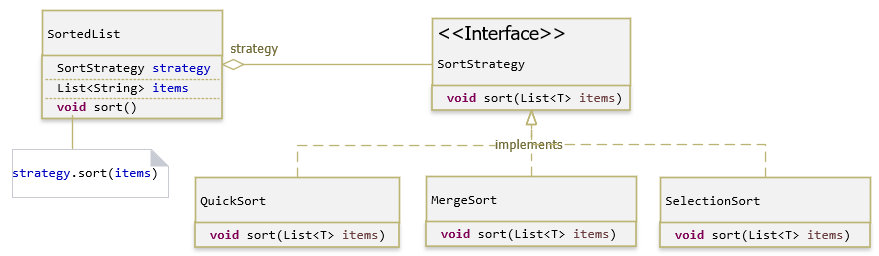


Các thành phần tham gia Strategy Pattern:

* Strategy : định nghĩa các hành vi có thể có của một Strategy.
* ConcreteStrategy : cài đặt các hành vi cụ thể của Strategy.
* Context : chứa một tham chiếu đến đối tượng Strategy và nhận các yêu cầu từ Client, các yêu cầu này sau đó được ủy quyền cho Strategy thực hiện.

### Ví dụ Strategy Pattern với ứng dụng sắp xếp

Chương trình của chúng ta cung cấp nhiều giải thuật sắp xếp khác nhau: quick sort, merge sort, selection sort, heap sort, tim sort, …. Tùy theo loại dữ liệu, số lượng phần tử, … mà người dùng có thể chọn một giải thuật sắp xếp phù hợp.



SortStrategy.java

package com.gpcoder.patterns.behavioral.strategy.sort;

import java.util.List;

public interface SortStrategy {

<T> void sort(List<T> items);

}

QuickSort.java

package com.gpcoder.patterns.behavioral.strategy.sort;

import java.util.List;

public class QuickSort implements SortStrategy {

@Override

public <T> void sort(List<T> items) {

System.out.println("Quick sort");

}

}

MergeSort.java

package com.gpcoder.patterns.behavioral.strategy.sort;

import java.util.List;

public class MergeSort implements SortStrategy {

@Override

public <T> void sort(List<T> items) {

System.out.println("Merge sort");

}

}

SelectionSort.java

package com.gpcoder.patterns.behavioral.strategy.sort;

import java.util.List;

public class SelectionSort implements SortStrategy {

@Override

public <T> void sort(List<T> items) {

System.out.println("Selection sort");

}

}

SortedList.java

SortStrategy.java

package com.gpcoder.patterns.behavioral.strategy.sort;

import java.util.ArrayList;

import java.util.List;

public class SortedList {

private SortStrategy strategy;

private List<String> items = new ArrayList<>();

public void setSortStrategy(SortStrategy strategy) {

this.strategy = strategy;

}

public void add(String name) {

items.add(name);

}

public void sort() {

strategy.sort(items);

}

}

StrategyPatternExample.java

package com.gpcoder.patterns.behavioral.strategy.sort;

public class StrategyPatternExample {

public static void main(String[] args) {

SortedList sortedList = new SortedList();

sortedList.add("Java Core");

sortedList.add("Java Design Pattern");

sortedList.add("Java Library");

sortedList.add("Java Framework");

sortedList.setSortStrategy(new QuickSort());

sortedList.sort();

sortedList.setSortStrategy(new MergeSort());

sortedList.sort();

}

}

Output của chương trình:

Quick sort

Merge sort

### Ví dụ Strategy Pattern với ứng dụng thanh toán online

Code cũng tương tự như ví dụ trên, các vui lòng xem trên [Github](https://github.com/gpcodervn/Design-Pattern-Tutorial/tree/master/DesignPatternTutorial/src/com/gpcoder/patterns/behavioral/strategy).

## Lợi ích của Strategy Pattern là gì?

* Đảm bảo nguyên tắc [Single responsibility principle (SRP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Single_responsibility_principle_SRP) : một lớp định nghĩa nhiều hành vi và chúng xuất hiện dưới dạng với nhiều câu lệnh có điều kiện. Thay vì nhiều điều kiện, chúng ta sẽ chuyển các nhánh có điều kiện liên quan vào lớp Strategy riêng lẻ của nó.
* Đảm bảo nguyên tắc [Open/Closed Principle (OCP)](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Open-Closed_principle_OCP) : chúng ta dễ dàng mở rộng và kết hợp hành vi mới mà không thay đổi ứng dụng.
* Cung cấp một sự thay thế cho kế thừa.

## Sử dụng Strategy Pattern khi nào?

* Khi muốn có thể thay đổi các thuật toán được sử dụng bên trong một đối tượng tại thời điểm run-time.
* Khi có một đoạn mã dễ thay đổi, và muốn tách chúng ra khỏi chương trình chính để dễ dàng bảo trì.
* Tránh sự rắc rối, khi phải hiện thực một chức năng nào đó qua quá nhiều lớp con.
* Cần che dấu sự phức tạp, cấu trúc bên trong của thuật toán.

## So sánh Strategy Pattern vs State Pattern

Sự phụ thuộc (dependency):

* [State Pattern](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/) đi kèm với một chút phụ thuộc trong các lớp con, chẳng hạn một State biết về các trạng thái khác xuất hiện trước/ sau nó. Ví dụ: t[ín hiệu đèn giao thông](https://github.com/gpcodervn/Design-Pattern-Tutorial/tree/master/DesignPatternTutorial/src/com/gpcoder/patterns/behavioral/state/trafficlight) như đèn đỏ -> đèn xanh -> đèn vàng, ứng dụng workflow nơi mà chúng có thể biết được bước trước và sau nó làm gì.
* Strategy Pattern không có sự phụ thuộc như State. Bất kỳ loại trạng thái nào cũng có thể được khởi tạo một cách độc lập, chúng không biết được sự tồn tại của các Strategy khác. Ví dụ: người dùng có thể lựa chọn giải thuật sắp xếp, phương thức thanh toán, nén file rar/zip, xuất báo cáo ra file excel/ csv, …

Mục đích sử dụng (intent) :

* Strategy Pattern quyết định cách thực hiện một số hành động, sử dụng Strategy khi chúng ta cần  trả lời how.
* Trong khi State Pattern quyết định khi nào để thực hiện chúng, sử dụng Strategy khi chúng ta cần  trả lời when/ what (state or type).

Thời điểm khởi tạo (binding time):

* Strategy Pattern là một mô hình khởi tạo một lần, không có sự luân chuyển trạng thái.
* Trong khi State  Pattern thì năng động hơn (có thể nhiều lần), có sự luân chuyển từ trạng thái này sang trạng thái khác. State Pattern có thể coi là trường hợp mở rộng của Strategy Pattern.

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/strategy>
* <https://refactoring.guru/design-patterns/strategy>
* <https://www.javatpoint.com/strategy-pattern>
* <https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm>
* <https://stackoverflow.com/questions/1658192/what-is-the-difference-between-strategy-design-pattern-and-state-design-pattern/>
* Design Patterns: Elements of Reusable Object-Oriented Software – GOF
* Design Pattern for dummies

# Hướng dẫn Java Design Pattern – Template Method

Đăng vào 07/01/2019 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 1750 Lượt xem

Trong quá trình phát triển ứng dụng, chúng ta có các component khác nhau có sự tương đồng đáng kể, nhưng chúng không sử dụng interface/ abstract class chung, dẫn đến code duplicate ở nhiều nơi. Nếu muốn thay đổi chung cho tất cả component, chúng ta phải đi sửa ở từng nơi trong component, làm tốn nhiều chi phí không cần thiết. Một trong những cách để giải quyết vấn đề này là sử dụng Template Method Pattern.

Nội dung [[Ẩn](https://gpcoder.com/4810-huong-dan-java-design-pattern-template-method/)]

* [1 Template Method Pattern là gì?](https://gpcoder.com/4810-huong-dan-java-design-pattern-template-method/" \l "Template_Method_Pattern_la_gi)
* [2 Cài đặt Template Method Pattern như thế nào?](https://gpcoder.com/4810-huong-dan-java-design-pattern-template-method/" \l "Cai_dat_Template_Method_Pattern_nhu_the_nao)
  + [2.1 Ví dụ Template Method Pattern với thiết kế template của website](https://gpcoder.com/4810-huong-dan-java-design-pattern-template-method/" \l "Vi_duTemplate_Method_Pattern_voi_thiet_ke_template_cua_website)
* [3 Lợi ích của Template Method Pattern là gì?](https://gpcoder.com/4810-huong-dan-java-design-pattern-template-method/" \l "Loi_ich_cua_Template_Method_Pattern_la_gi)
* [4 Sử dụng Template Method Pattern khi nào?](https://gpcoder.com/4810-huong-dan-java-design-pattern-template-method/" \l "Su_dung_Template_Method_Pattern_khi_nao)

## Template Method Pattern là gì?

Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’s structure.

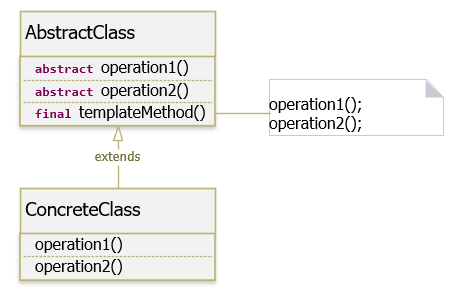
Template Method Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Pattern này nói rằng “Định nghĩa một bộ khung của một thuật toán trong một chức năng, chuyển giao việc thực hiện nó cho các lớp con. Mẫu Template Method cho phép lớp con định nghĩa lại cách thực hiện của một thuật toán, mà không phải thay đổi cấu trúc thuật toán“.

Điều này có nghĩa là Template method giúp cho chúng ta tạo nên một bộ khung (template) cho một vấn đề đang cần giải quyết. Trong đó các đối tượng cụ thể sẽ có cùng các bước thực hiện, nhưng trong mỗi bước thực hiện đó có thể khác nhau. Điều này sẽ tạo nên một cách thức truy cập giống nhau nhưng có hành động và kết quả khác nhau.

Template Method Pattern được sử dụng khá nhiều trong mô hình Abstract – Concrete Class. Khi chúng ta muốn các Concrete class tự thực thi xử lí theo cách của nó, nhưng đồng thời vẫn đảm bảo tuận theo những ràng buộc nhất định từ Abstract class. Ví dụ như ràng buộc về thứ tự các bước thực hiện, hay ràng buộc về dữ liệu đầu vào, đầu ra, …

Trong Template method pattern, Abstract class định nghĩa ra một template method để thực hiện một chức năng nào đó. Template method này sẽ gọi đến các method khác bên trong Abstract class để tạo dựng nên bộ khung. Nhưng có thể các method đó sẽ không được thực thi bên trong Abstract class, mà sẽ được override và thực thi lại bên trong các Concrete class.

## Cài đặt Template Method Pattern như thế nào?

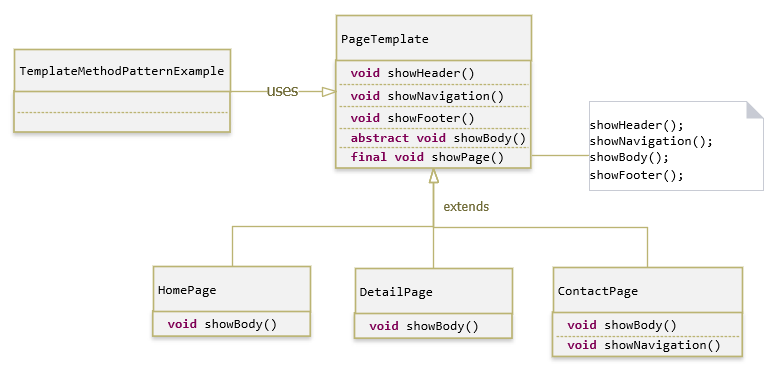


Các thành phần tham gia Template Method Pattern:

* AbstractClass :
  + Định nghĩa các phương thức trừu tượng cho từng bước có thể được điều chỉnh bởi các lớp con.
  + Cài đặt một phương thức duy nhất điều khiển thuật toán và gọi các bước riêng lẻ đã được cài đặt ở các lớp con.
* ConcreteClass : là một thuật toán cụ thể, cài đặt các phương thức của AbstractClass. Các thuật toán này ghi đè lên các phương thức trừu tượng để cung cấp các triển khai thực sự. Nó không thể ghi đè phương thức duy nhất đã được cài đặt ở AbstractClass (templateMethod).

### Ví dụ Template Method Pattern với thiết kế template của website

Cấu trúc của một website thông thường gồm các phần header, footer, navigation (menu), body. Riêng phần body thường xuyên thay đổi, sẽ hiển thị riêng theo từng trang. Những phần khác ít khi thay đổi, trừ khi có yêu cầu đặt biệt. Thay vì phải viết tất cả các phần ở mỗi trang, chúng ta có thể gom chúng lại và đặt trong một template để tái sử dụng mà không duplicate code ở nhiều nơi.



PageTemplate.java

package com.gpcoder.patterns.behavioral.templatemethod.template;

public abstract class PageTemplate {

protected void showHeader() {

System.out.println("

<header />");

}

protected void showNavigation() {

System.out.println("

<nav />");

}

protected void showFooter() {

System.out.println("

<footer />");

}

protected abstract void showBody();

public final void showPage() {

showHeader();

showNavigation();

showBody();

showFooter();

}

}

HomePage.java

package com.gpcoder.patterns.behavioral.templatemethod.template;

public class HomePage extends PageTemplate {

@Override

protected void showBody() {

System.out.println("Content of home page page");

}

}

DetailPage.java

package com.gpcoder.patterns.behavioral.templatemethod.template;

public class DetailPage extends PageTemplate {

@Override

protected void showBody() {

System.out.println("Content of detail");

}

}

ContactPage.java

package com.gpcoder.patterns.behavioral.templatemethod.template;

public class ContactPage extends PageTemplate {

@Override

protected void showNavigation() {

// Just do nothing

// Because we don't want to show navigation bar on contact page

}

@Override

protected void showBody() {

System.out.println("Content of contact page");

}

}

TemplateMethodPatternExample.java

package com.gpcoder.patterns.behavioral.templatemethod.template;

public class TemplateMethodPatternExample {

public static void main(String[] args) {

PageTemplate homePage = new HomePage();

homePage.showPage();

System.out.println();

PageTemplate detailPage = new DetailPage();

detailPage.showPage();

System.out.println();

PageTemplate contactPage = new ContactPage();

contactPage.showPage();

}

}

Output của chương trình:

<header />

<nav />

Content of home page page

<footer />

<header />

<nav />

Content of detail

<footer />

<header />

Content of contact page

<footer />

## Lợi ích của Template Method Pattern là gì?

* Tái sử dụng code (reuse), tránh trùng lặp code (duplicate): đưa những phần trùng lặp vào lớp cha (abstract class).
* Cho phép người dùng override chỉ một số phần nhất định của thuật toán lớn, làm cho chúng ít bị ảnh hưởng hơn bởi những thay đổi xảy ra với các phần khác của thuật toán.

## Sử dụng Template Method Pattern khi nào?

* Khi có một thuật toán với nhiều bước và mong muốn cho phép tùy chỉnh chúng trong lớp con.
* Mong muốn chỉ có một triển khai phương thức trừu tượng duy nhất của một thuật toán.
* Mong muốn hành vi chung giữa các lớp con nên được đặt ở một lớp chung.
* Các lớp cha có thể gọi các hành vi trong các lớp con của chúng một cách thống nhất (step by step).

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/template_method>
* <https://refactoring.guru/design-patterns/template-method>
* <https://www.javatpoint.com/template-pattern>
* Design Patterns: Elements of Reusable Object-Oriented Software – GOF
* Design Pattern for dummies

# Hướng dẫn Java Design Pattern – Visitor

Đăng vào 10/01/2019 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 1594 Lượt xem

Ở các bài viết trước chúng ta đã cùng tìm hiểu về các Pattern thuộc nhóm hành vi (Behavior Pattern): [Chain of Responsibility](https://gpcoder.com/4665-huong-dan-java-design-pattern-chain-of-responsibility/), [Command](https://gpcoder.com/4686-huong-dan-java-design-pattern-command/), [Interpreter](https://gpcoder.com/4702-huong-dan-java-design-pattern-interpreter/), [Iterator](https://gpcoder.com/4724-huong-dan-java-design-pattern-iterator/), [Mediator](https://gpcoder.com/4740-huong-dan-java-design-pattern-mediator/), [Memento](https://gpcoder.com/4763-huong-dan-java-design-pattern-memento/), [Observer](https://gpcoder.com/4747-huong-dan-java-design-pattern-observer/), [State](https://gpcoder.com/4785-huong-dan-java-design-pattern-state/), [Strategy](https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/), [Template Method](https://gpcoder.com/4810-huong-dan-java-design-pattern-template-method/). Trong bài viết này, chúng ta sẽ cùng tìm hiểu về một Pattern cuối cùng thuộc nhóm Behavior Pattern được giới thiệu trong cuốn “Design Patterns: Elements of Reusable Object-Oriented Software – GOF” là Visitor Design Pattern.

Nội dung [[Ẩn](https://gpcoder.com/4813-huong-dan-java-design-pattern-visitor/)]

* [1 Single Dispatch và Double Dispatch là gì?](https://gpcoder.com/4813-huong-dan-java-design-pattern-visitor/" \l "Single_Dispatch_va_Double_Dispatch_la_gi)
  + [1.1 Single Dispatch là gì?](https://gpcoder.com/4813-huong-dan-java-design-pattern-visitor/" \l "Single_Dispatch_la_gi)
  + [1.2 Double Dispatch là gì?](https://gpcoder.com/4813-huong-dan-java-design-pattern-visitor/" \l "Double_Dispatch_la_gi)
* [2 Visitor Pattern là gì?](https://gpcoder.com/4813-huong-dan-java-design-pattern-visitor/" \l "Visitor_Pattern_la_gi)
* [3 Cài đặt Visitor Pattern như thế nào?](https://gpcoder.com/4813-huong-dan-java-design-pattern-visitor/" \l "Cai_dat_Visitor_Pattern_nhu_the_nao)
  + [3.1 Ví dụ sử dụng Visitor Pattern](https://gpcoder.com/4813-huong-dan-java-design-pattern-visitor/" \l "Vi_du_su_dungVisitor_Pattern)
* [4 Lợi ích của Visitor Pattern là gì?](https://gpcoder.com/4813-huong-dan-java-design-pattern-visitor/" \l "Loi_ich_cua_Visitor_Pattern_la_gi)
* [5 Sử dụng Visitor Pattern khi nào?](https://gpcoder.com/4813-huong-dan-java-design-pattern-visitor/" \l "Su_dung_Visitor_Pattern_khi_nao)
* [6 Giới hạn của Visitor Pattern và Áp dụng Reflection trong Visitor Pattern](https://gpcoder.com/4813-huong-dan-java-design-pattern-visitor/" \l "Gioi_han_cua_Visitor_Pattern_va_Ap_dung_Reflection_trong_Visitor_Pattern)

## Single Dispatch và Double Dispatch là gì?

Trước khi đi vào chi tiết về Visitor Pattern, chúng ta hãy cùng tìm hiểu về Single Dispatch và Double Dispatch.

### Single Dispatch là gì?

Hãy xem đoạn code bên dưới:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | package com.gpcoder.patterns.behavioral.visitor.singledispatch;    import lombok.Data;    @Data  public class Book {      private String name;      private int price;  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public class ProgramingBook extends Book {    }    public class BusinessBook extends Book {    } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | package com.gpcoder.patterns.behavioral.visitor.singledispatch.example1;    public interface Customer {        void buy(Book book);        void buy(ProgramingBook book);        void buy(BusinessBook book);  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | package com.gpcoder.patterns.behavioral.visitor.singledispatch.example1;    public class Developer implements Customer {        @Override      public void buy(Book book) {          System.out.println("Developer buy a Book");      }        @Override      public void buy(ProgramingBook book) {          System.out.println("Developer buy a Programing Book");        }        @Override      public void buy(BusinessBook book) {          System.out.println("Developer buy a Business Book");      }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | package com.gpcoder.patterns.behavioral.visitor.singledispatch.example1;    public class SingleDispatchExample {        public static void main(String[] args) {          Book book = new ProgramingBook(); // (1)          Customer gpcoder = new Developer();          gpcoder.buy(book);            ProgramingBook programingBook = new ProgramingBook(); // (2)          gpcoder.buy(programingBook); // Developer buy a Programing Book      }  } |

Output của chương trình:

|  |  |
| --- | --- |
| 1  2 | Developer buy a Book  Developer buy a Programing Book |

Như bạn thấy, kết quả ở (1) không như chúng ta mong đợi, cái chúng ta cần là kết quả như (2).

Tại sao sử dụng như ở (1) lại không đúng?

=> Trong Java, chúng ta có thể định nghĩa các phương thức cùng tên nhưng khác nhau về tham số (tính đa hình). Một phương thức được gọi sẽ dựa trên 2 yếu tố: tên của phương thức và kiểu đối tượng gọi nó. Cơ chế này được gọi là Single Dispatch. Trong ví dụ trên, do ở (1) chúng ta đã cast nó về Book nên phương thức buy(Book book) được gọi.

Để giải quyết vấn đề trên, chúng ta có thể sử dụng từ khóa instanceof để kiểm tra kiểu của đối tượng gọi như sau:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | package com.gpcoder.patterns.behavioral.visitor.singledispatch.example2;    import com.gpcoder.patterns.behavioral.visitor.singledispatch.Book;  import com.gpcoder.patterns.behavioral.visitor.singledispatch.BusinessBook;  import com.gpcoder.patterns.behavioral.visitor.singledispatch.ProgramingBook;  import com.gpcoder.patterns.behavioral.visitor.singledispatch.example1.Customer;    public class Developer2 implements Customer {        @Override      public void buy(Book book) {          if (book instanceof ProgramingBook) {              ProgramingBook programingBook = (ProgramingBook) book;              buy(programingBook);          } else if (book instanceof BusinessBook) {              BusinessBook businessBook = (BusinessBook) book;              buy(businessBook);          } else {              System.out.println("Developer buy a Book");          }      }        @Override      public void buy(ProgramingBook book) {          System.out.println("Developer buy a Programing Book");        }        @Override      public void buy(BusinessBook book) {          System.out.println("Developer buy a Business Book");      }  } |

Chạy lại chương trình trên với implement mới, chúng ta có kết quả như sau:

|  |  |
| --- | --- |
| 1  2 | Developer buy a Programing Book  Developer buy a Programing Book |

### Double Dispatch là gì?

Ngoài cách sử dụng instanceof như trong ví dụ trên, chúng ta có thể sử dụng kỹ thuật Double Dispatch. Trong Double Dispatch, một phương thức sẽ được gọi dựa trên 3 yếu tố: tên của phương thức, kiểu của cả đối tượng gọi và kiểu của đối số truyền vào.

Vậy áp dụng Double Dispatch như thế nào, chúng ta hãy cùng theo dõi trong phần tiếp theo với Visitor Design pattern.

## Visitor Pattern là gì?

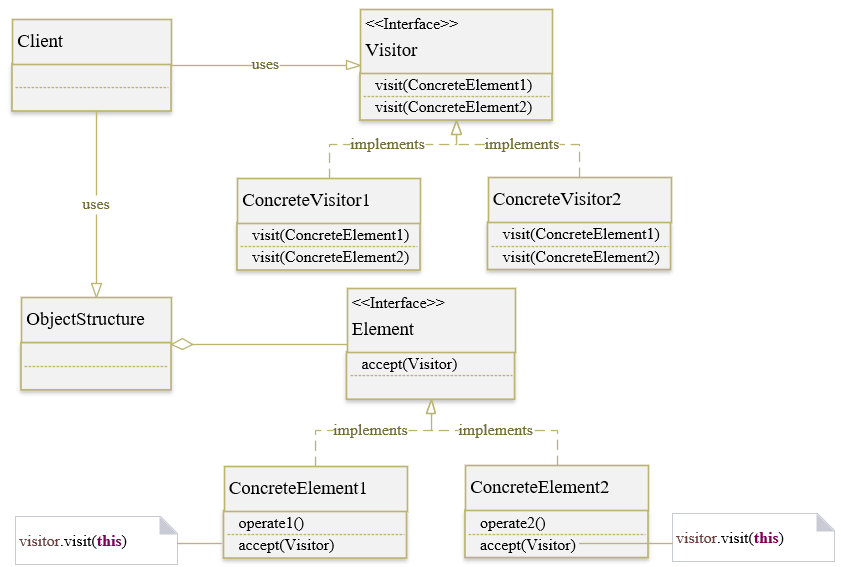
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Visitor Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Visitor cho phép định nghĩa các thao tác (operations) trên một tập hợp các đối tượng (objects) không đồng nhất (về kiểu) mà không làm thay đổi định nghĩa về lớp (classes) của các đối tượng đó. Để đạt được điều này, trong mẫu thiết kế visitor ta định nghĩa các thao tác trên các lớp tách biệt gọi các lớp visitors, các lớp này cho phép tách rời các thao tác với các đối tượng mà nó tác động đến. Với mỗi thao tác được thêm vào, một lớp visitor tương ứng được tạo ra.

Đây là một kỹ thuật giúp chúng ta phục hồi lại kiểu dữ liệu bị mất (thay vì dùng instanceof). Nó thực hiện đúng thao tác dựa trên tên của phương thức, kiểu của cả đối tượng gọi và kiểu của đối số truyền vào.

Visitor còn được biết đến như là Double dispatch.

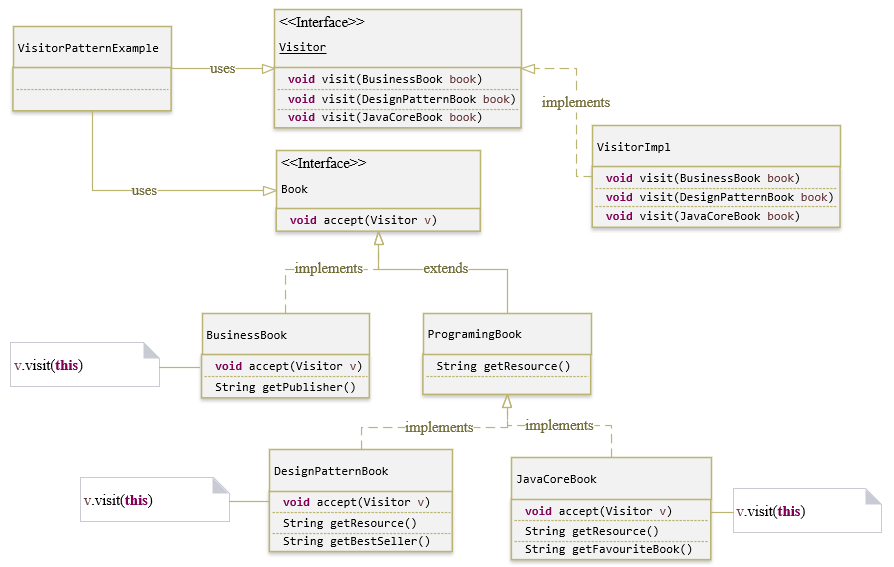
## Cài đặt Visitor Pattern như thế nào?



Các thành phần tham gia Visitor Pattern:

* Visitor :
  + Là một interface hoặc một abstract class được sử dụng để khai báo các hành vi cho tất cả các loại visitor.
  + Class này định nghĩa một loạt các các phương thức truy cập chấp nhận các ConcreteElement cụ thể khác nhau làm tham số. Điều này sẽ hơi giống với cơ chế nạp chồng (overloading) nhưng các loại tham số nên khác nhau do đó các hành vi hoàn toàn khác nhau. Các hành vi truy cập sẽ được thực hiện trên từng phần tử cụ thể trong cấu trúc đối tượng thông qua phương thức visit(). Loại phần tử cụ thể đầu vào sẽ quyết định phương thức được gọi.
* ConcreteVisitor : cài đặt tất cả các phương thức abstract đã khai báo trong Visitor. Mỗi visitor sẽ chịu trách nhiệm cho các hành vi khác nhau của đối tượng.
* Element (Visitable): là một thành phần trừu tượng, nó khai báo phương thức accept() và chấp nhận đối số là Visitor.
* ConcreteElement (ConcreteVisitable): cài đặt phương thức đã được khai báo trong Element dựa vào đối số visitor được cung cấp.
* ObjectStructure : là một lớp chứa tất cả các đối tượng Element, cung cấp một cơ chế để duyệt qua tất cả các phần tử. Cấu trúc đối tượng này có thể là một tập hợp (collection) hoặc một cấu trúc phức tạp giống như một đối tượng tổng hợp (composite).
* Client : không biết về ConcreteElement và chỉ gọi phương thức accept() của Element.

### Ví dụ sử dụng Visitor Pattern



Visitor.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | package com.gpcoder.patterns.behavioral.visitor.doubledispatch;    public interface Visitor {        void visit(BusinessBook book);        void visit(DesignPatternBook book);        void visit(JavaCoreBook book);  } |

VisitorImpl.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package com.gpcoder.patterns.behavioral.visitor.doubledispatch;    public class VisitorImpl implements Visitor {        @Override      public void visit(BusinessBook a) {          System.out.println(a.getPublisher());      }        @Override      public void visit(DesignPatternBook w) {          System.out.println(w.getBestSeller());      }        @Override      public void visit(JavaCoreBook g) {          System.out.println(g.getFavouriteBook());      }  } |

Book.java

|  |  |
| --- | --- |
| 1  2  3  4  5 | package com.gpcoder.patterns.behavioral.visitor.doubledispatch;    public interface Book {      void accept(Visitor v);  } |

BusinessBook.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | package com.gpcoder.patterns.behavioral.visitor.doubledispatch;    public class BusinessBook implements Book {      public void accept(Visitor v) {          v.visit(this);      }        public String getPublisher() {          return "The publisher of business book";      }  } |

ProgramingBook.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | package com.gpcoder.patterns.behavioral.visitor.doubledispatch;    public interface ProgramingBook extends Book {        String getResource();  } |

DesignPatternBook.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | package com.gpcoder.patterns.behavioral.visitor.doubledispatch;    public class DesignPatternBook implements ProgramingBook {        @Override      public void accept(Visitor v) {          v.visit(this);      }        @Override      public String getResource() {          return "<https://github.com/gpcodervn/Design-Pattern-Tutorial/>";      }        public String getBestSeller() {          return "The best Seller of design pattern book";      }  } |

JavaCoreBook.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | package com.gpcoder.patterns.behavioral.visitor.doubledispatch;    public class JavaCoreBook implements ProgramingBook {        @Override      public void accept(Visitor v) {          v.visit(this);      }        @Override      public String getResource() {          return "<https://github.com/gpcodervn/Java-Tutorial/>";      }        public String getFavouriteBook() {          return "The most favourite book of java core";      }  } |

VisitorPatternExample.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package com.gpcoder.patterns.behavioral.visitor.doubledispatch;    public class VisitorPatternExample {        public static void main(String[] args) throws Exception {          Book book1 = new BusinessBook();          Book book2 = new JavaCoreBook();          Book book3 = new DesignPatternBook();            Visitor v = new VisitorImpl();          book1.accept(v);          book2.accept(v);          book3.accept(v);      }  } |

Output của chương trình:

|  |  |
| --- | --- |
| 1  2  3 | The publisher of business book  The most favourite book of java core  The best Seller of design pattern book |

## Lợi ích của Visitor Pattern là gì?

* Cho phép một hoặc nhiều hành vi được áp dụng cho một tập hợp các đối tượng tại thời điểm run-time, tách rời các hành vi khỏi cấu trúc đối tượng.
* Đảm bảo nguyên tắc [Open/ Close](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Open-Closed_principle_OCP): đối tượng gốc không bị thay đổi, dễ dàng thêm hành vi mới cho đối tượng thông qua visitor.

## Sử dụng Visitor Pattern khi nào?

* Khi có một cấu trúc đối tượng phức tạp với nhiều class và interface. Người dùng cần thực hiện một số hành vi cụ thể của riêng đối tượng, tùy thuộc vào concrete class của chúng.
* Khi chúng ta phải thực hiện một thao tác trên một nhóm các loại đối tượng tương tự. Chúng ta có thể di chuyển logic hành vi từ các đối tượng sang một lớp khác.
* Khi cấu trúc dữ liệu của đối tượng ít khi thay đổi nhưng hành vi của chúng được thay đổi thường xuyên.
* Khi muốn tránh sử dụng toán tử instanceof.

## Giới hạn của Visitor Pattern và Áp dụng Reflection trong Visitor Pattern

Hạn chế lớn nhất của Visitor Pattern là chúng ta cần phải biết kiểu trả về của phương thức visit() tại thời điểm thiết kế nếu không chúng ta phải thay đổi interface và tất cả các cài đặt của nó. Như trong ví dụ trên, nếu chúng ta muốn thêm một loại sách khác (chẳng hạn OthersBook), chúng ta phải thêm phương thức visit(OthersBook) trong interface và sửa đổi tất cả các cài đặt tương ứng của Visitor.

Để giải quyết vấn đề này, chúng ta có thể sử dụng kỹ thuật [Reflection](https://gpcoder.com/2883-huong-dan-su-dung-java-reflection/) để xác định chính xác concrete class được gọi tại thời điểm run-time thay vì compile-time. Bằng cách này, chúng ta có thể sử dụng tham số là super class thay vì concrete class. Hãy xem ví dụ bên dưới, các lớp mới nếu không có phương thức cài đặt riêng visit(), thì nó sẽ sử dụng hàm mặc định, chúng ta không cần thêm phương thức visit() mới và tránh sửa đổi tất cả các concrete visitor đã tồn tại.

Thêm một loại sách mới OthersBook.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | package com.gpcoder.patterns.behavioral.visitor.doubledispatch.reflection;    public class OthersBook implements ProgramingBook {        @Override      public void accept(Visitor v) {          v.visit(this);      }        @Override      public String getResource() {          return "Undefined";      }  } |

Thay đổi Visitor.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | package com.gpcoder.patterns.behavioral.visitor.doubledispatch.reflection;    import java.lang.reflect.Method;    public abstract class Visitor {      public abstract void visit(Book book);        protected Method getMethod(Class<?> clazz) {          while (!clazz.equals(Object.class)) { // Check superclasses              try {                  return this.getClass().getDeclaredMethod("visit", clazz);              } catch (NoSuchMethodException ex) {                  clazz = clazz.getSuperclass();              }          }          Class<?>[] interfaces = clazz.getInterfaces(); // Check interfaces          for (Class<?> anInterface : interfaces) {              try {                  return this.getClass().getDeclaredMethod("visit", anInterface);              } catch (NoSuchMethodException ex) {                  ex.printStackTrace();              }          }          return null;      }        protected void defaultVisit(Book book) {          System.out.println("A book");      }  } |

Thay đổi VisitorImpl.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | package com.gpcoder.patterns.behavioral.visitor.doubledispatch.reflection;    import java.lang.reflect.InvocationTargetException;  import java.lang.reflect.Method;    public class VisitorImpl extends Visitor {        public void visit(Book book) {          Method downPolymorphic = getMethod(book.getClass());          if (downPolymorphic == null) {              defaultVisit(book);          } else {              try {                  downPolymorphic.invoke(this, book);              } catch (IllegalAccessException | IllegalArgumentException | InvocationTargetException e) {                  e.printStackTrace();              }          }      }        public void visit(BusinessBook a) {          System.out.println(a.getPublisher());      }        public void visit(DesignPatternBook w) {          System.out.println(w.getBestSeller());      }        public void visit(JavaCoreBook g) {          System.out.println(g.getFavouriteBook());      }  } |

ReflectiveVisitorPatternExample.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | package com.gpcoder.patterns.behavioral.visitor.doubledispatch.reflection;    public class ReflectiveVisitorPatternExample {        public static void main(String[] args) throws Exception {          Book book1 = new BusinessBook();          Book book2 = new JavaCoreBook();          Book book3 = new DesignPatternBook();          Book book4 = new OthersBook();            Visitor v = new VisitorImpl();          book1.accept(v);          book2.accept(v);          book3.accept(v);          book4.accept(v);      }  } |

Output của chương trình:

|  |  |
| --- | --- |
| 1  2  3  4 | The publisher of business book  Effective Java  Head First Design Patterns  A book |

Như bạn thấy, nếu chúng ta không xác định kiểu trả về của phương thức accept(OthersBook), nó sẽ gọi hàm mặc định defaultVisit(Book).

Tài liệu tham khảo:

* <https://sourcemaking.com/design_patterns/visitor>
* <https://refactoring.guru/design-patterns/visitor>
* <https://vi.wikipedia.org/wiki/Visitor_pattern>
* <https://www.journaldev.com/1769/visitor-design-pattern-java>
* <https://howtodoinjava.com/design-patterns/behavioral/visitor-design-pattern-example-tutorial/>
* <https://www.baeldung.com/java-visitor-pattern>
* Design Patterns: Elements of Reusable Object-Oriented Software – GOF

# Hướng dẫn Java Design Pattern – DAO

Đăng vào 17/01/2019 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 3045 Lượt xem

Một trong những khía cạnh quan trọng của lớp nghiệp vụ (business layer) là lớp truy cập dữ liệu (data access layer) để kết nối các dịch vụ (service) với cơ sở dữ liệu (database). Việc truy cập dữ liệu tùy thuộc vào nguồn dữ liệu, loại lưu trữ như database, text file, xml file, json file, …Thậm chí nó khác với cách triển khai của nó, ví dụ: cú pháp truy vấn SQL khác nhau giữa MySQL, SQL Server, Oracle, … Với mong muốn sẽ không có gì khác biệt khi truy cập cơ sở dữ liệu quan hệ, phân tích xml file hay bất kỳ nguồn dữ liệu nào khác, chúng ta có thể áp dụng mẫu thiết kế đối tượng truy cập dữ liệu (Data Access Object Pattern– DAO Pattern).

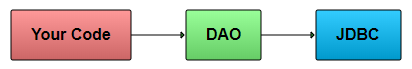
Nội dung [[Ẩn](https://gpcoder.com/4935-huong-dan-java-design-pattern-dao/)]

* [1 DAO Pattern là gì?](https://gpcoder.com/4935-huong-dan-java-design-pattern-dao/" \l "DAO_Pattern_la_gi)
* [2 Cài đặt DAO Pattern như thế nào?](https://gpcoder.com/4935-huong-dan-java-design-pattern-dao/" \l "Cai_dat_DAO_Pattern_nhu_the_nao)
  + [2.1 Ví dụ sử dụng DAO Pattern](https://gpcoder.com/4935-huong-dan-java-design-pattern-dao/" \l "Vi_du_su_dung_DAO_Pattern)
* [3 Lợi ích của DAO Pattern là gì?](https://gpcoder.com/4935-huong-dan-java-design-pattern-dao/" \l "Loi_ich_cua_DAO_Pattern_la_gi)
* [4 Sử dụng DAO Pattern khi nào?](https://gpcoder.com/4935-huong-dan-java-design-pattern-dao/" \l "Su_dung_DAO_Pattern_khi_nao)

## DAO Pattern là gì?

Data Access Object (DAO) Pattern là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern).  Mẫu thiết kế DAO được sử dụng để phân tách logic lưu trữ dữ liệu trong một lớp riêng biệt. Theo cách này, các service được che dấu về cách các hoạt động cấp thấp để truy cập cơ sở dữ liệu được thực hiện. Nó còn được gọi là nguyên tắc Tách logic (Separation of Logic).

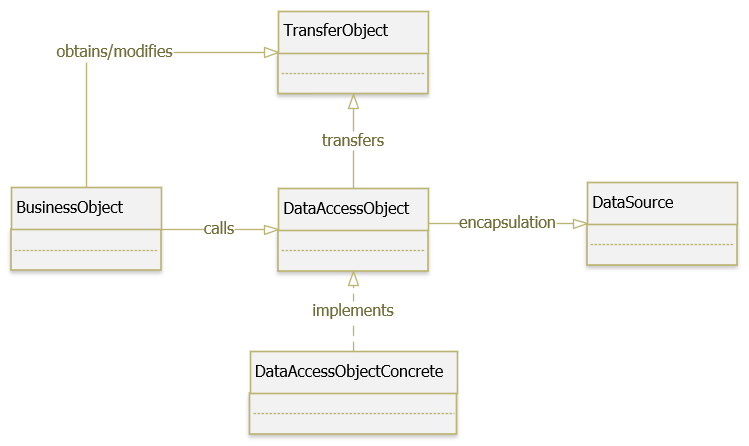
Ý tưởng là thay vì có logic giao tiếp trực tiếp với cơ sở dữ liệu, hệ thống file, dịch vụ web hoặc bất kỳ cơ chế lưu trữ nào mà ứng dụng cần sử dụng, chúng ta sẽ để logic này sẽ giao tiếp với lớp trung gian DAO. Lớp DAO này sau đó giao tiếp với hệ thống lưu trữ, hệ quản trị CSDL như thực hiện các công việc liên quan đến lưu trữ và truy vấn dữ liệu (tìm kiếm, thêm, xóa, sửa,…).



DAO Pattern dựa trên các nguyên tắc thiết kế abstraction và encapsulation. Nó bảo vệ phần còn lại của ứng dụng khỏi mọi thay đổi trong lớp lưu trữ, ví dụ: thay đổi database từ Oracle sang MySQL, thay đổi công nghệ lưu trữ từ file sang database.

Trong Java, DAO được triển khai theo nhiều cách khác nhau như [Java Persistence API](https://en.wikipedia.org/wiki/Java_Persistence_API),  [Enterprise Java Bean](https://en.wikipedia.org/wiki/Enterprise_JavaBeans) (EJP), [Object-relational mapping](https://en.wikipedia.org/wiki/Object-relational_mapping) (ORM) với các implement cụ thể như [Hibernate](http://hibernate.org/), [iBATIS](http://ibatis.apache.org/), [Spring JPA](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/), … Chi tiết về các phần này chúng ta sẽ cùng thảo luận ở một bài viết khác.

## Cài đặt DAO Pattern như thế nào?

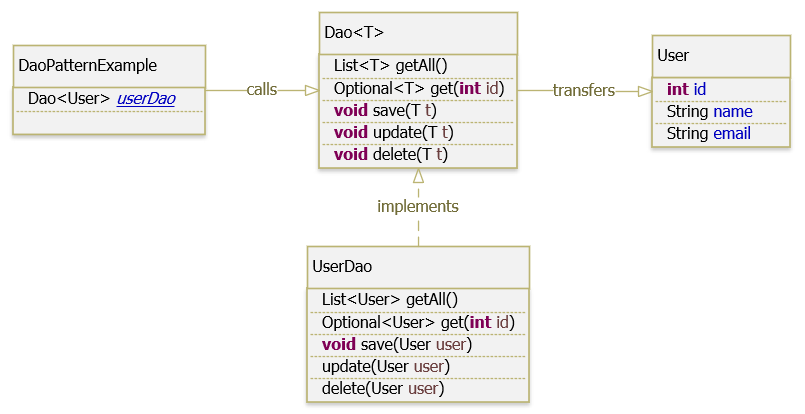


Các thành phần tham gia mẫu Data Access Object (DAO) Pattern:

* BusinessObject : đại diện cho Client, yêu cầu truy cập vào nguồn dữ liệu để lấy và lưu trữ dữ liệu.
* DataAccessObject (DAO): là một interface định nghĩa các phương thức trừu tượng việc triển khai truy cập dữ liệu cơ bản cho BusinessObject để cho phép truy cập vào nguồn dữ liệu (DataSource).
* DataAccessObjectConcrete : cài đặt các phương thức được định nghĩa trong DAO, lớp này sẽ thao tác trực tiếp với nguồn dữ liệu (DataSource).
* DataSource : là nơi chứa dữ liệu, nó có thể là database, xml, json, text file, webservice, …
* TransferObject : là một POJO (Plain old Java object) object, chứa các phương thức get/set được sử dụng để lưu trữ dữ liệu và được sử dụng trong DAO class.

### Ví dụ sử dụng DAO Pattern

Ví dụ bên dưới cho chúng ta thấy cách áp dụng DAO Pattern trong việc cung cấp interface chung để thao tác với cơ sở dữ liệu (CRUD). Các thao tác này bao gồm: Create/ Save, Read/ Get, Update, Delete.



User.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package com.gpcoder.patterns.structural.dao;    import lombok.AllArgsConstructor;  import lombok.Data;    @Data  @AllArgsConstructor  public class User {        private int id;        private String name;        private String email;  } |

Dao.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | package com.gpcoder.patterns.structural.dao;    import java.util.List;  import java.util.Optional;    public interface Dao<T> {        List<T> getAll();        Optional<T> get(int id);        void save(T t);        void update(T t);        void delete(T t);  } |

UserDao.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | package com.gpcoder.patterns.structural.dao;    import java.util.ArrayList;  import java.util.List;  import java.util.Optional;    public class UserDao implements Dao<User> {        private List<User> users = new ArrayList<>();        public UserDao() {          users.add(new User(1, "GP Coder", "contact@gpcoder.com"));          users.add(new User(2, "Giang Phan", "gpcodervn@gmail.com"));      }        @Override      public List<User> getAll() {          return users;      }        @Override      public Optional<User> get(int id) {          return users.stream().filter(u -> u.getId() == id).findFirst();      }        @Override      public void save(User user) {          users.add(user);      }        @Override      public void update(User user) {          get(user.getId()).ifPresent(existUser -> {              existUser.setName(user.getName());              existUser.setEmail(user.getEmail());          });      }        @Override      public void delete(User user) {          get(user.getId()).ifPresent(existUser -> users.remove(existUser));      }  } |

DaoPatternExample.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package com.gpcoder.patterns.structural.dao;    public class DaoPatternExample {        private static Dao<User> userDao;        public static void main(String[] args) {          userDao = new UserDao();            User user1 = userDao.get(1).get();          System.out.println("user1: " + user1);            user1.setName("updated." + user1.getName());          userDao.update(user1);            System.out.println("All users: ");          userDao.getAll().forEach(user -> System.out.println(user));      }  } |

User.java

|  |  |
| --- | --- |
| 1  2  3  4 | user1: User(id=1, name=GP Coder, email=contact@gpcoder.com)  All users:  User(id=1, name=updated.GP Coder, email=contact@gpcoder.com)  User(id=2, name=Giang Phan, email=gpcodervn@gmail.com) |

## Lợi ích của DAO Pattern là gì?

* Giảm sự kết nối (loose coupling) giữa logic nghiệp vụ (Business) và logic lưu trữ (Persistence).
* Mẫu DAO cho phép đóng gói code để thực hiện thao tác CRUD, ngăn chặn việc implement riêng lẻ trong từng phần khác nhau của ứng dụng.
* Dễ mở rộng, bảo trì: tất cả các chi tiết lưu trữ được ẩn khỏi phần còn lại của ứng dụng. Do đó, những thay đổi có thể được thực hiện bằng cách chỉ sửa đổi một implement của DAO trong khi phần còn lại của ứng dụng không bị ảnh hưởng. DAO hoạt động như một trung gian giữa ứng dụng và cơ sở dữ liệu.
* Dễ hiểu: mọi người đều theo một quy chuẩn đã được định sẵn, nên dễ hiểu hơn, tiết kiệm được nhiều thời gian hơn.
* Trong một dự án lớn hơn, các nhóm khác nhau làm việc trên các phần khác nhau của ứng dụng, mẫu DAO cho phép phân tách rõ ràng các thành phần này.

## Sử dụng DAO Pattern khi nào?

* Khi muốn thay đổi nguồn dữ liệu sau này, như chuyển từ cơ dữ liệu MySQL sang Oracle, SQL Server, …
* Khi muốn phân tách rõ ràng các thành phần của ứng dụng.

Tài liệu tham khảo:

* <https://en.wikipedia.org/wiki/Data_access_object>
* <https://www.tutorialspoint.com/design_pattern/data_access_object_pattern.htm>
* <https://www.baeldung.com/java-dao-pattern>
* <https://javarevisited.blogspot.com/2013/01/data-access-object-dao-design-pattern-java-tutorial-example.html>
* <https://www.oracle.com/technetwork/java/dataaccessobject-138824.html>

# Hướng dẫn Java Design Pattern – MVC

Đăng vào 19/02/2019 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 1252 Lượt xem

Nội dung [[Ẩn](https://gpcoder.com/5160-huong-dan-java-design-pattern-mvc/)]

* [1 MVC Pattern là gì?](https://gpcoder.com/5160-huong-dan-java-design-pattern-mvc/" \l "MVC_Pattern_la_gi)
* [2 Cài đặt MVC Pattern như thế nào?](https://gpcoder.com/5160-huong-dan-java-design-pattern-mvc/" \l "Cai_dat_MVC_Pattern_nhu_the_nao)
  + [2.1 Ví dụ sử dụng MVC Pattern](https://gpcoder.com/5160-huong-dan-java-design-pattern-mvc/" \l "Vi_du_su_dung_MVC_Pattern)
* [3 Lợi ích của MVC Pattern là gì?](https://gpcoder.com/5160-huong-dan-java-design-pattern-mvc/" \l "Loi_ich_cua_MVC_Pattern_la_gi)

## MVC Pattern là gì?

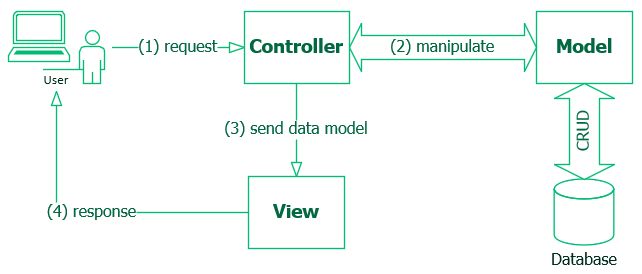
Model – View – Controller (MVC) Pattern là một mẫu thiết kế nhằm mục tiêu chia tách phần giao diện và code để dễ quản lý, phát triển và bảo trì.

MVC Pattern  là một dạng Architectural Design Pattern được áp dụng để xử lý các vấn đề liên quan đến kiến trúc ứng dụng.

MVC Pattern tuân thủ nguyên tắc thiết kế Separation of Concern, giúp phân tách logic của các tầng (layer) khác nhau trong một chương trình thành các đơn vị độc lập.

MVC chia ứng dụng phần mềm ra làm 3 phần có tương tác với nhau: Model (dữ liệu), View (giao diện), Controller (điều khiển tương tác giữa Model và View).

## Cài đặt MVC Pattern như thế nào?



Các thành phần tham gia MVC Pattern:

* Model : là nơi lưu trữ dữ liệu người dùng, chứa business logic. Nó cho phép truy xuất dữ liệu để hiển thị hoặc thu thập dữ liệu. Model là cầu nối giữa thành phần View và Controller. Mục đích quan trọng nhất của nó là kết nối cơ sở dữ liệu (database), xử lý dữ liệu và chuẩn bị dữ liệu để chuyển đến các thành phần khác.
* View : là giao diện của hệ thống, nơi dữ liệu (Model) được hiển thị, nhận tương tác trực tiếp với người dùng. Trong ứng dụng web, View là một phần của hệ thống, nơi mà các mã HTML được sinh ra và hiển thị. Một vấn đề quan trọng là View không được lấy dữ liệu trực tiếp từ Controller mà phải thông qua Model.
* Controller : nhận yêu cầu, dữ liệu từ người dùng, sau đó cập nhật sang Model và cuối cùng trả kết quả lại View để show kết quả cho người dùng. Controller không chứa bất kỳ logic nghiệp vụ nào.

### Ví dụ sử dụng MVC Pattern

Giả sử chúng ta có một ứng dụng quản lý cho phép hiển thị thông tin chi tiết, chỉnh sửa thông tin của Sinh viên. Chương trình của chúng ta như sau:

StudentModel.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | package com.gpcoder.patterns.other.mvc;    import java.time.LocalDate;    import lombok.AllArgsConstructor;  import lombok.Data;    @Data  @AllArgsConstructor  public class StudentModel {        private Integer id;      private String name;      private LocalDate updatedDate;        public boolean save() {          if (name.length() <= 10) {              this.updatedDate = LocalDate.now();              return true;          }          return false;      }  } |

StudentView.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | package com.gpcoder.patterns.other.mvc;    import java.time.format.DateTimeFormatter;    public class StudentView {        public void showHomePage() {          System.out.println("Home page of Student");      }        public void showDetailPage(StudentModel student) {          System.out.println("Student: ");          System.out.println("+ Id: " + student.getId());          System.out.println("+ Name: " + student.getName());          System.out.println("+ Updated date: " +                  student.getUpdatedDate().format(DateTimeFormatter.ofPattern("dd/MM/yyyy")));          System.out.println("---------------------------");      }        public void showUpdatedPage(StudentModel student) {          System.out.println("Saved successfully");          showDetailPage(student);      }        public void showErrorMessage() {          System.out.println("ERROR!!!");      }  } |

StudentController.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | package com.gpcoder.patterns.other.mvc;    import java.time.LocalDate;    public class StudentController {        private StudentModel student;      private StudentView view;        public StudentController() {          student = new StudentModel(1, "No Name", LocalDate.now());          view = new StudentView();      }        public void showDetail() {          view.showDetailPage(student);      }        public void changeName(String newName) {          student.setName(newName);      }        public void save() {          boolean isSuccess = student.save();          if (isSuccess) {              view.showUpdatedPage(student);          } else {              view.showErrorMessage();          }      }  } |

MVCPatternExample.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | package com.gpcoder.patterns.other.mvc;    public class MVCPatternExample {        public static void main(String[] args) {          StudentController controller = new StudentController();          controller.showDetail();            controller.changeName("gpcoder");          controller.save();            controller.changeName("Java Design Pattern – MVC");          controller.save();      }  } |

Output của chương trình:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Student:  + Id: 1  + Name: No Name  + Updated date: 17/02/2019  ---------------------------  Saved successfully  Student:  + Id: 1  + Name: gpcoder  + Updated date: 17/02/2019  ---------------------------  ERROR!!! |

Trong ví dụ trên, chúng ta thấy:

* StudentModel : đóng vai trò là một Model, chứa thông tin của sinh viên (id, name) và logic xử lý lưu thông tin sinh viên (save).
* StudentView : đóng vai trò là một View, dùng để show thông tin của sinh viên (Model).
* StudentController : đóng vai trò là một Controller, nhận yêu cầu từ User (showDetail, changeName, save), cập nhật thông tin vào Model (setName), chứa flow điều khiển gọi xử lý lưu thông tin từ Model (save), xác định View nào sẽ được hiển thị (showDetailPage, showUpdatedPage, showErrorMessage).
* MVCPatternExample : đóng vai trò là một User, chỉ có thể thao tác với Controller để xem thông tin Model từ View.

## Lợi ích của MVC Pattern là gì?

* MVC cho phép phát triển ứng dụng nhanh chóng. Chúng ta có thể tạo một Model trước và chuyển thông tin này cho nhóm phát triển giao diện người dùng. Họ có thể bắt đầu implement phần View song song với đội Backend implement Controller và logic để phát triển nhanh hơn.
* Giảm độ phức tạp của code.
* Tăng khả năng tái sử dụng.
* Giảm bớt sự phụ thuộc trong code, dễ bảo trì, dễ nâng cấp hơn.

Tài liệu tham khảo:

* <https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm>
* <https://www.oracle.com/technetwork/articles/javase/mvc-136693.html>
* <https://www.journaldev.com/16974/mvc-design-pattern>

# Giới thiệu Aspect Oriented Programming (AOP)

Đăng vào 09/02/2019 Được đăng bởi [GP Coder](https://gpcoder.com/author/gpadmin/) 1352 Lượt xem

Nội dung [[Ẩn](https://gpcoder.com/5112-gioi-thieu-aspect-oriented-programming-aop/)]

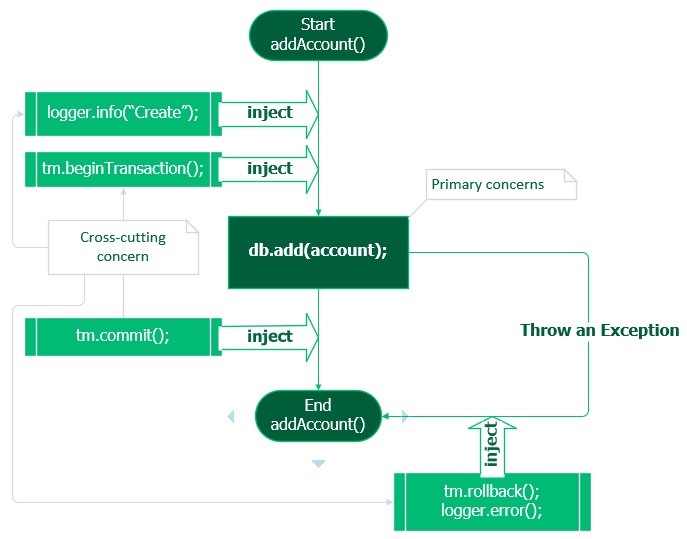
* [1 AOP là gì?](https://gpcoder.com/5112-gioi-thieu-aspect-oriented-programming-aop/" \l "AOP_la_gi)
* [2 Weaving là gì?](https://gpcoder.com/5112-gioi-thieu-aspect-oriented-programming-aop/" \l "Weaving_la_gi)
  + [2.1 Static weaving](https://gpcoder.com/5112-gioi-thieu-aspect-oriented-programming-aop/" \l "Static_weaving)
  + [2.2 Dynamic weaving](https://gpcoder.com/5112-gioi-thieu-aspect-oriented-programming-aop/" \l "Dynamic_weaving)
* [3 Cài đặt AOP trong Java như thế nào?](https://gpcoder.com/5112-gioi-thieu-aspect-oriented-programming-aop/" \l "Cai_dat_AOP_trong_Java_nhu_the_nao)
* [4 Ví dụ tự xây dựng một AOP framework](https://gpcoder.com/5112-gioi-thieu-aspect-oriented-programming-aop/" \l "Vi_dutu_xay_dung_mot_AOPframework)
* [5 Lợi ích của AOP là gì?](https://gpcoder.com/5112-gioi-thieu-aspect-oriented-programming-aop/" \l "Loi_ich_cua_AOP_la_gi)

## AOP là gì?

Trong khi xây dựng các chương trình ứng dụng, có rất nhiều những vấn đề liên quan đến phần mềm mà chúng ta cần quan tâm. Chẳng hạn, chúng ta xây dựng một hệ thống đăng ký tạo tài khoản cho một ngân hàng. Ngoài công việc chính cho phép người dùng có thể tạo tài khoản (core concern), hệ thống còn phải đảm bảo các vấn đề khác (cross-cutting concern) như chứng thực người dùng, kiểm tra ràng buộc, quản lý transaction, xử lý ngoại lệ, ghi log, debug, đo hiệu năng của ứng dụng, …

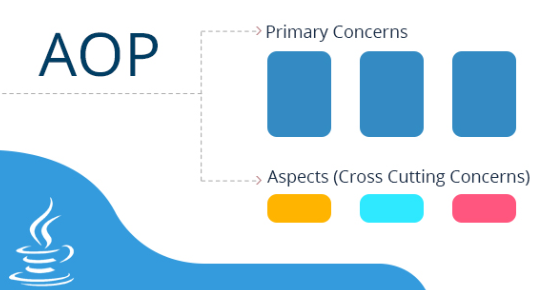
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | Logger logger = Logger.getLogger(...);  TransactionManager tm = getTransactionManager();  public void addAccount(Account account) {      logger.info("Creating (" + account + ") Account");      try {          tm.beginTransaction();          db.add(account);          tm.commit();      } catch (Exception) {          tm.rollback();          logger.error("Account creation failed");      }  } |

Như bạn thấy, logic của chương trình của chúng ta phải làm rất nhiều việc như ghi log, mở/ đóng transaction, xử lý ngoại lệ, … Khi có nhiều phương thức tương tự vậy trong ứng dụng, code chúng ta bị liên kết chặt vào nhau, duplicate code, phân mảnh nhiều nơi, khó khăn khi sửa đổi thay thêm logic mới, … Để giải quyết vấn đề này, chúng ta có thể sử dụng kỹ thuật AOP.



AOP là từ viết tắt của Aspect Oriented Programming, dịch ra tiếng Việt là “phương pháp lập trình hướng khía cạnh”. AOP là một kỹ thuật lập trình cho phép phân tách chương trình thành cách module riêng rẽ, không phụ thuộc nhau. Khi hoạt động, chương trình sẽ kết hợp các module lại để thực hiện các chức năng nhưng khi sửa đổi chức năng thì chỉ cần sửa đổi trên một module cụ thể.

AOP còn được gọi là Aspect Oriented Software Development (AOSD) là một nguyên tắc thiết kế, giúp tách rời các yêu cầu hay các vấn đề được quan tâm (separation of concerns) trong chương trình thành các thành phần độc lập và từ đó tăng tính uyển chuyển cho chương trình. Trong Separation of concerns, người ta cho rằng những vấn đề tương tự nhau nên được giải quyết trong một unit of code. Khi lập trình thủ tục (functional programming), một unit of code là một function/ method, còn trong lập trình hướng đối tượng (OOP) thì unit of code là một class.



Trong AOP, chương trình của chúng ta được chia thành 2 loại concern:

* Core concern/ Primary concern: là requirement, logic xử lý chính của chương trình.
* Cross-cutting concern: là những logic xử lý phụ cần được thực hiện của chương trình khi core concern được gọi như security, logging, tracing, monitoring, …

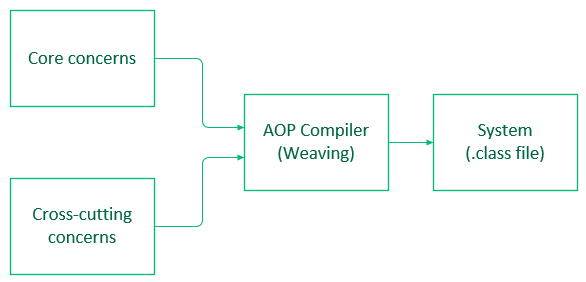
Một số thuật ngữ khác trong AOP:

* Joinpoint: là một điểm trong chương trình, là những nơi có thể được chèn những cross-cutting concern. Chẳng hạn chúng ta cần ghi log lại sau khi chạy method nào đó thì điểm ngay sau method đó được thực thi gọi là một Jointpoint. Một Jointpoint có thể là một phương thức được gọi, một ngoại lệ được throw ra, hay một field được thay đổi.
* Pointcut: có nhiều cách để xác định Joinpoint, những cách như thế được gọi là Pointcut. Nó là các biểu thức được sử dụng để kiểm tra nó có khớp với các Jointpoint để xác định xem Advice có cần được thực hiện hay không.
* Advice: những xử lý phụ (crosscutting concern) được thêm vào xử lý chính (core concern), code để thực hiện các xử lý đó được gọi Advice. Advice được chia thành các loại sau:
  + Before: được thực hiện trước join point.
  + After: được thực hiện sau join point.
  + Around: được thực hiện trước và sau join point.
  + After returning : được thực hiện sau join point hoàn thành một cách bình thường.
  + After throwing : được thực hiện sau khi join point được kết thúc bằng một Exception.
* Aspect: tương tự như một Java class. Một Aspect đóng gói toàn bộ cross-cutting concern và có thể chứa các JointPoint, PointCut, Advice.
* Target Object : là những đối tượng mà advice được áp dụng.

Một vài cross-cutting concern thường thấy trong ứng dụng:

* Logging
* Monitor
* Access control
* Error handling
* Transaction management
* Session management
* Input/output validation

## Weaving là gì?



Về cơ bản Weaving (đan/ dệt) là quá trình liên kết các thành phần aspect và non-aspect của một chương trình để tạo ra đầu ra mong muốn.

Có một vài cách khác nhau giữa các hệ thống AOP về cách tạo ra Weaving. Có thể chia làm các loại Weaving: Compile-time weaving (static weaving), Load-Time Weaving và Run-time weaving (dynamic weaving).

* Compile-time weaving :
  + Pre-Compile Weaving : sử dụng bộ tiền xử lý (pre-processor) để combine code của aspect và code non-aspect lại với nhau trước khi code được biên dịch thành byte code Java (.class).
  + Post-Compile Weaving / Binary weaving : cách này dùng để inject code của aspect vào những tập tin .class của Java đã được compile.
* Load-Time Weaving : cách này dùng để inject code của aspect khi class cần sử dụng aspect được load vào JVM, nghĩa là trong khi ứng dụng đang chạy.
* Run-time weaving : thực hiện weaving và unweaving code của aspect và non-aspect tại run-time.

### Static weaving

Static weaving là quá trình combine code của aspect và code non-aspect lại với nhau trước khi code được biên dịch thành Java byte code (.class) bằng cách sử dụng bộ tiền xử lý (pre-processor). Do đó, code gốc chỉ được thay đổi một lần tại thời gian biên dịch (compile). Hiệu suất của code được combine này tương đương với code được viết theo truyền thống.

Hạn chế của phương pháp Static weaving là khó khăn trong việc xác định code của aspect sau này hoặc thực hiện thay đổi đối với code của aspect. Mỗi khi code của aspect bị thay đổi, tất cả tất cả code sử dụng aspect phải được biên dịch lại.

### Dynamic weaving

Dynamic weaving khắc phục một số hạn chế gặp phải khi weaving được thực hiện tại thời gian biên dịch (Compile-time weaving/ Static weaving). Có thể tránh được yêu cầu biên dịch lại (recompilation), triển khai lại (redeployment) và khởi động lại (restart) bằng cách thực hiện quy trình weaving trong thời gian chạy (run-time weaving).

Có một chút khác biệt giữa load-time và run-time weaving.

* Load-time weaving chỉ đơn giản là trì hoãn quá trình weaving cho đến khi các lớp được nạp bởi class loader. Cách tiếp cận này yêu cầu sử dụng một weaving class loader hoặc thay thế class loader bằng một loader khác. Hạn chế là load-time tăng lên và thiếu quyền truy cập vào các aspect trong khi chạy.
* Run-time weaving là quá trình weaving và unweaving tại run-time. Cách tiếp cận này yêu cầu các cơ chế mới để can thiệp vào việc tính toán chạy.

Các AOP Framework khác nhau có cách thực hiện dynamic weaving khác nhau. Trong khi AspectWerkz sử dụng sửa đổi byte code thông qua chức năng cấp JVM và kiến ​​trúc “hotswap” để thực hiện weaving các lớp tại run-time, thì Spring AOP Framework dựa trên các proxy thay vì các class loader hoặc dựa vào các đối số JVM.

Dynamic weaving cho phép tăng tốc các giai đoạn thiết kế và kiểm thử trong phát triển phần mềm, vì các aspect mới có thể được thêm vào hoặc các aspect hiện tại có thể được thay đổi mà không cần biên dịch lại và triển khai lại các ứng dụng. Tuy nhiên, một nhược điểm lớn là hiệu suất giảm, vì weaving xảy ra tại run-time.

## Cài đặt AOP trong Java như thế nào?

Một vài ý tưởng để implement AOP trong chương trình của chúng ta:

* Class-weaving : như đã đề cập ở phần trên.
* Proxy-based : bạn có thể tưởng tượng nó như là một ví dụ sử dụng [Decorator Pattern](https://gpcoder.com/4574-huong-dan-java-design-pattern-decorator/). Sử dụng công cụ mã hóa byte code của một số thư viện như JDK proxy, CGLib proxy, chúng ta có thể chặn các lệnh gọi hàm và thêm code của riêng mình để được thực thi trước.
  + JDK proxy : đây là cách đơn giản nhất, nhưng chỉ có thể xử lý với các phương thức public được gọi, không thể xử lý các cuộc gọi nội bộ (các cuộc gọi bắt nguồn từ chính lớp đó).
  + CGLib proxy : yêu cầu chỉnh sửa byte code bị giới hạn và có thể xử lý các lời gọi phương thức private, nhưng vẫn không thể xử lý truy cập thuộc tính trực tiếp.
* Sử dụng các thư viện sau: Google Guice, AspectJ, Spring AOP.

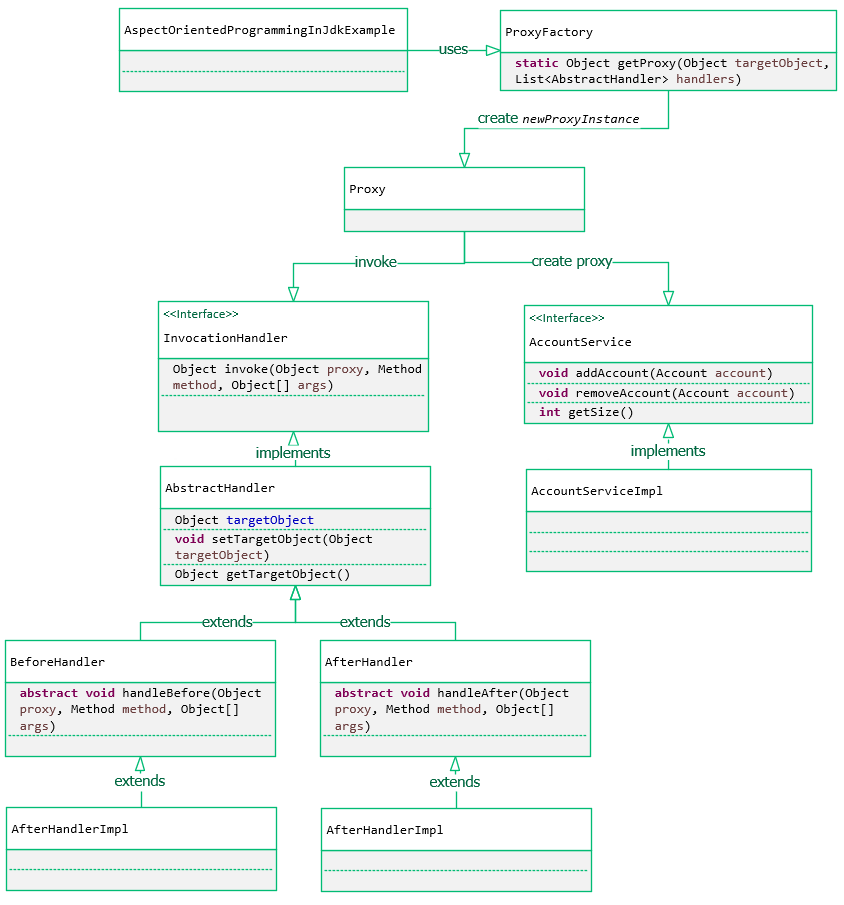
## Ví dụ tự xây dựng một AOP framework

Trong ví dụ bên dưới chúng ta sẽ tự viết code để tạo AOP Framework sử dụng JDK Proxy mà không cần sử dụng bất cứ một thư viện thứ ba nào (third-party libraries).

Để xử dụng JDK Proxy chúng ta thực hiện các bước sau:

* Tạo Invocation Handler: class này phải implemenet java.lang.reflect.InvocationHandler. InvocationHandler là interface được thực hiện bởi trình xử lý lời gọi (invocation handler) của một proxy instance. Khi một phương thức được gọi trên một proxy instance, lời gọi phương thức được mã hóa và gửi đến phương thức gọi của invocation handler của nó.
* Tạo Proxy Instance : sử dụng phương thức Proxy.newProxyInstance() được cung cấp bởi factory method java.lang.reflect.Proxy.

Ví dụ: chúng ta cần thêm một vài xử lý trước và sau khi các phương thức trong class AccountService được gọi.



Account.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | package com.gpcoder.aop.account;    import lombok.AllArgsConstructor;  import lombok.Data;    @Data  @AllArgsConstructor  public class Account {        private String owner;      private String currency;      private int balance;  } |

AccountService.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | package com.gpcoder.aop.account;    public interface AccountService {        void addAccount(Account account);        void removeAccount(Account account);        int getSize();  } |

AccountServiceImpl.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | package com.gpcoder.aop.account;    import java.util.ArrayList;  import java.util.List;    public class AccountServiceImpl implements AccountService {        private List<Account> accounts = new ArrayList<>();        @Override      public void addAccount(Account account) {          System.out.println("addAccount: " + account);          accounts.add(account);      }        @Override      public void removeAccount(Account account) {          System.out.println("removeAccount: " + account);          accounts.remove(account);      }        @Override      public int getSize() {          System.out.println("getSize: " + accounts.size());          return accounts.size();      }  } |

AbstractHandler.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | package com.gpcoder.aop.handler;    import java.lang.reflect.InvocationHandler;    public abstract class AbstractHandler implements InvocationHandler {        private Object targetObject;        public void setTargetObject(Object targetObject) {          this.targetObject = targetObject;      }        public Object getTargetObject() {          return targetObject;      }  } |

BeforeHandler.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | package com.gpcoder.aop.handler;    import java.lang.reflect.Method;    /\*\*   \* The class BeforeHandler provides a template for the before execution   \*/  public abstract class BeforeHandler extends AbstractHandler {        /\*\*       \* Handles before execution of actual method.       \*/      public abstract void handleBefore(Object proxy, Method method, Object[] args);        /\*       \* (non-Javadoc)       \*       \* @see java.lang.reflect.InvocationHandler#invoke(java.lang.Object,       \* java.lang.reflect.Method, java.lang.Object[])       \*/      @Override      public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {          handleBefore(proxy, method, args);          return method.invoke(getTargetObject(), args);      }  } |

BeforeHandlerImpl.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | package com.gpcoder.aop.handler.impl;    import java.lang.reflect.Method;    import com.gpcoder.aop.handler.BeforeHandler;    /\*\*   \* This class provides implementation before actual execution of method.   \*/  public class BeforeHandlerImpl extends BeforeHandler {        @Override      public void handleBefore(Object proxy, Method method, Object[] args) {          // Provide your own cross cutting concern          System.out.println("Handling before actual method execution");      }  } |

AfterHandler.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | package com.gpcoder.aop.handler;    import java.lang.reflect.Method;    public abstract class AfterHandler extends AbstractHandler {        /\*\*       \* Handles after the execution of method.       \*/      public abstract void handleAfter(Object proxy, Method method, Object[] args);        @Override      public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {          Object result = method.invoke(getTargetObject(), args);          handleAfter(proxy, method, args);          return result;      }  } |

AfterHandlerImpl.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | package com.gpcoder.aop.handler.impl;    import java.lang.reflect.Method;    import com.gpcoder.aop.handler.AfterHandler;    /\*\*   \* This class provides an implementation of business logic which will be   \* executed after the actual method execution.   \*/  public class AfterHandlerImpl extends AfterHandler {        @Override      public void handleAfter(Object proxy, Method method, Object[] args) {          // Provide your own cross cutting concern          System.out.println("Handling after actual method execution");          System.out.println("---");      }  } |

ProxyFactory.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | package com.gpcoder.aop.handler;    import java.lang.reflect.Proxy;  import java.util.List;    /\*\*   \* A factory for creating Proxy objects.   \*/  public class ProxyFactory {        private ProxyFactory() {          throw new UnsupportedOperationException();      }        public static Object getProxy(Object targetObject, List<AbstractHandler> handlers) {          if (handlers != null && !handlers.isEmpty()) {              Object proxyObject = targetObject;              for (AbstractHandler handler : handlers) {                  handler.setTargetObject(proxyObject);                  proxyObject = Proxy.newProxyInstance(targetObject.getClass().getClassLoader(),                          targetObject.getClass().getInterfaces(), handler);              }              return proxyObject;          }            return targetObject;      }  } |

AspectOrientedProgrammingInJdkExample.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | package com.gpcoder.aop;    import java.util.ArrayList;  import java.util.List;    import com.gpcoder.aop.account.Account;  import com.gpcoder.aop.account.AccountService;  import com.gpcoder.aop.account.AccountServiceImpl;  import com.gpcoder.aop.handler.AbstractHandler;  import com.gpcoder.aop.handler.ProxyFactory;  import com.gpcoder.aop.handler.impl.AfterHandlerImpl;  import com.gpcoder.aop.handler.impl.BeforeHandlerImpl;    /\*\*   \* This class to verify an AOP example using JDK proxy.   \*/  public class AspectOrientedProgrammingInJdkExample {        public static void main(String[] args) {            List<AbstractHandler> handlers = new ArrayList<>();          handlers.add(new BeforeHandlerImpl());          handlers.add(new AfterHandlerImpl());            AccountService proxy = (AccountService) ProxyFactory.getProxy(new AccountServiceImpl(), handlers);          Account account = new Account("gpcoder", "USD", 100);          proxy.addAccount(account);          proxy.getSize();          proxy.removeAccount(account);          proxy.getSize();      }  } |

Output của chương trình:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | Handling before actual method execution  addAccount: Account(owner=gpcoder, currency=USD, balance=100)  Handling after actual method execution  ---  Handling before actual method execution  getSize: 1  Handling after actual method execution  ---  Handling before actual method execution  removeAccount: Account(owner=gpcoder, currency=USD, balance=100)  Handling after actual method execution  ---  Handling before actual method execution  getSize: 0  Handling after actual method execution  --- |

## Lợi ích của AOP là gì?

* Tăng hiệu quả của Object-orented programming (OOP).
* AOP không phải dùng để thay thế OOP mà để bổ sung cho OOP, nơi mà OOP còn thiếu sót trong việc tạo những ứng dụng thuộc loại phức tạp.
* Tăng cường tối đa khả năng tái sử dụng của mã nguồn.
* Đảm bảo [Single responsibility principle](https://gpcoder.com/4200-cac-nguyen-ly-thiet-ke-huong-doi-tuong/" \l "Single_responsibility_principle_SRP_8211Nguyen_ly_don_chuc_nang): mỗi một module chỉ làm cái mà nó cần phải làm.
* Tuân thủ nguyên tắc “You aren’t gonna need it – [YAGNI](https://gpcoder.com/4299-mot-so-nguyen-tac-dinh-luat-trong-lap-trinh/" \l "Nguyen_tac_YAGNI)” – chúng ta chỉ cài đặt những thứ chúng ta thực sự cần, không bao giờ làm trước.
* Module hóa ở mức tiến trình/ chức năng.
* Code gọn gàng hơn do tách biệt phần xử lý chính và phần xử lý liên quan.
* Chức năng chính của chương trình không cần biết đến các chức năng phụ khác.
* Các chức năng phụ có thể được thêm thắt, bật tắt tại thời điểm run-time tùy theo yêu cầu.
* Các thay đổi nếu có đối với các chức năng phụ sẽ không ảnh hưởng đến chương trình chính.
* Hệ thống sẽ uyển chuyển và giảm thiểu tính phụ thuộc lẫn nhau của các module.

Tài liệu tham khảo:

* + <http://best-practice-software-engineering.ifs.tuwien.ac.at/technology/aop.html>
  + <https://en.wikipedia.org/wiki/Aspect-oriented_programming>
  + <https://en.wikipedia.org/wiki/Separation_of_concerns>
  + <https://www.eclipse.org/aspectj/doc/next/devguide/ltw.html>
  + <https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html>
  + <https://www.baeldung.com/java-dynamic-proxies>
  + <https://www.baeldung.com/cglib>