[**Python Design Patterns** 4](#_Toc22309012)

[1. Định nghĩa Design pattern 4](#_Toc22309013)

[2. Tại sao cần sử dụng Design pattern 4](#_Toc22309014)

[3. Phân loại Design Patterns 4](#_Toc22309015)

[3.1. Nhóm Creational (nhóm khởi tạo) 7](#_Toc22309016)

[3.2. Nhóm Structural (nhóm cấu trúc) 8](#_Toc22309017)

[3.3. Nhóm Behavioral (nhóm hành vi/ tương tác) 9](#_Toc22309018)

[4. Singleton Pattern 11](#_Toc22309019)

[4.1. Eager initialization 12](#_Toc22309020)

[4.2. Static block initialization 12](#_Toc22309021)

[4.3. Lazy Initialization 13](#_Toc22309022)

[4.4. Thread Safe Singleton 13](#_Toc22309023)

[4.5. Double Check Locking Singleton 14](#_Toc22309024)

[4.6. Bill Pugh Singleton Implementation 15](#_Toc22309025)

[4.7. Phá vỡ cấu trúc Singleton Pattern bằng Reflection. 15](#_Toc22309026)

[4.8. Enum Singleton 16](#_Toc22309027)

[4.9. Serialization and Singleton 16](#_Toc22309028)

[4.10. Singleton trong Python 18](#_Toc22309029)

[5. Factory Pattern 19](#_Toc22309030)

[5.1. Factory pattern là gì? 19](#_Toc22309031)

[5.2. Khi nào sử dụng Factory Pattern: 19](#_Toc22309032)

[5.3. Cách Implements 19](#_Toc22309033)

[5.4. Factory Pattern trong Python 22](#_Toc22309034)

[6. Abstract Factory Pattern 23](#_Toc22309035)

[6.1. Abstract Factory pattern là gì? 23](#_Toc22309036)

[6.2. Cách cài đặt như thế nào? 23](#_Toc22309037)

[6.3. Lợi ích 27](#_Toc22309038)

[7. Builder Pattern 28](#_Toc22309039)

[7.1. Abstract Factory pattern là gì? 28](#_Toc22309040)

[7.2. Cài đặt Builder Pattern như thế nào 29](#_Toc22309041)

[7.3. Lợi ích của Builder Pattern là gì 34](#_Toc22309042)

[7.4. Nhược điểm của Builder Pattern là gì? 34](#_Toc22309043)

[7.5. Sử dụng Builder Pattern khi nào? 35](#_Toc22309044)

[7.6. So sánh Builder Pattern với Factory/ Abstract Factory Pattern 35](#_Toc22309045)

[7.7. Builder Pattern trong Python 35](#_Toc22309046)

[8. Prototype Pattern 37](#_Toc22309047)

[8.1. Prototype pattern là gì? 37](#_Toc22309048)

[8.2. Cách cài đặt 37](#_Toc22309049)

[8.3. Lợi ích 38](#_Toc22309050)

[8.4. Sử dụng khi nào 38](#_Toc22309051)

[8.5. Prototype Pattern trong Python 38](#_Toc22309052)

[9. Adapter Pattern 39](#_Toc22309053)

[9.1. Adapter pattern là gì? 40](#_Toc22309054)

[9.2. Adapter pattern cài đặt như thế nào 41](#_Toc22309055)

[9.3. Cách cài đặt 43](#_Toc22309056)

[9.4. Lợi ích của Adapter Pattern là gì? 45](#_Toc22309057)

[9.5. Sử dụng Adapter Pattern khi nào? 46](#_Toc22309058)

[9.6. Adapter Pattern trong Python 46](#_Toc22309059)

[10. Bridge Pattern 47](#_Toc22309060)

[10.1. Bridge pattern là gì? 47](#_Toc22309061)

[10.2. Cách cài đặt 48](#_Toc22309062)

[10.3. Bridge Pattern trong Python 51](#_Toc22309063)

[10.4. Lợi ích của Bridge Pattern là gì 52](#_Toc22309064)

[10.5. Sử dụng Bridge Pattern khi nào 53](#_Toc22309065)

[11. Composite Pattern 53](#_Toc22309066)

[11.1. Composite pattern là gì? 53](#_Toc22309067)

[11.2. Cài đặt Composite Pattern 54](#_Toc22309068)

[11.3. Lợi ích của Composite Pattern là gì 57](#_Toc22309069)

[11.4. Sử dụng Composite Pattern khi nào 57](#_Toc22309070)

[11.5. Composite Pattern trong python 57](#_Toc22309071)

[12. Decorator Pattern 58](#_Toc22309072)

[12.1. Decorator pattern là gì? 58](#_Toc22309073)

[12.2. Cách cài đặt 58](#_Toc22309074)

[12.3. Lợi ích của Decorator Pattern 64](#_Toc22309075)

[12.4. Sử dụng Decorator Pattern khi nào 65](#_Toc22309076)

[12.5. So sánh Decorator và Adapter 65](#_Toc22309077)

[12.6. Decorator Pattern trong Python 65](#_Toc22309078)

[13. Facade Pattern 67](#_Toc22309079)

[13.1. Facade pattern là gì? 67](#_Toc22309080)

[13.2. Cài đặt Facade Pattern 69](#_Toc22309081)

[14. Tham khảo 69](#_Toc22309082)

**Python Design Patterns**

# Định nghĩa Design pattern

* Design Patterns (mẫu thiết kế) là một kỹ thuật trong lập trình hướng đối tượng. Nó cung cấp 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.

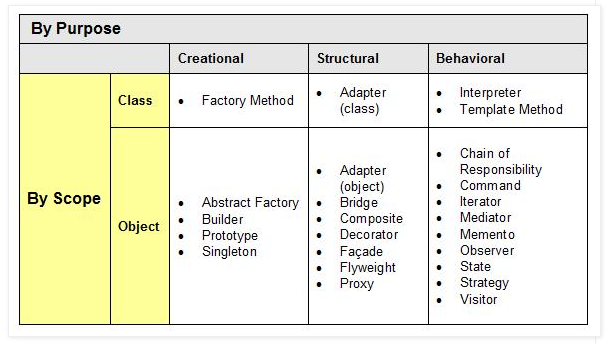
# Tại sao cần sử dụng Design pattern

* 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.

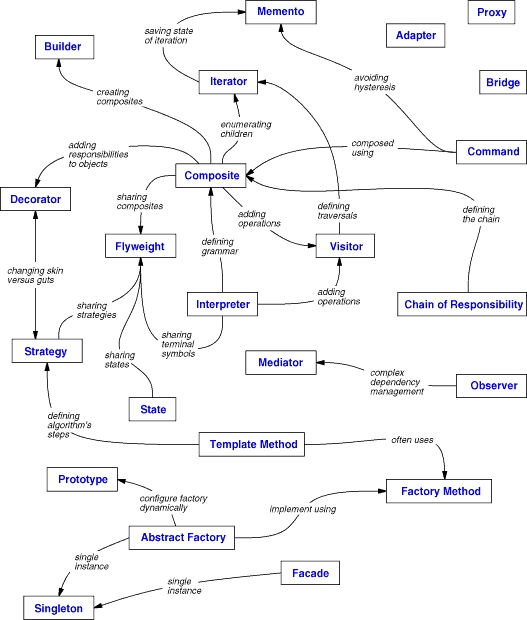
# Phân loại Design Patterns

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.

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** (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** (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** (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.

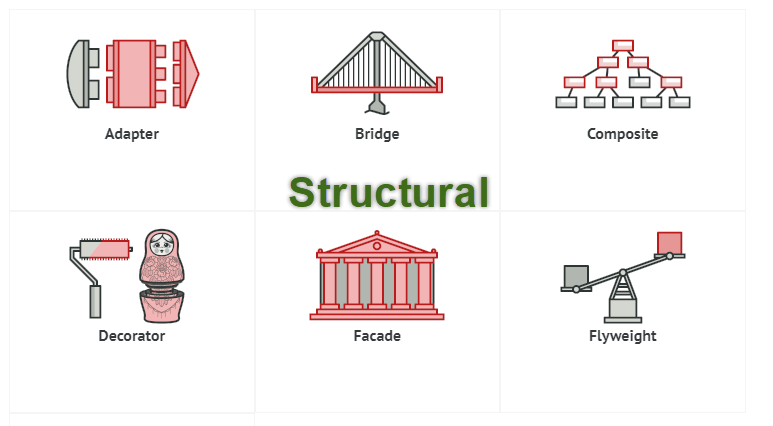


## Nhóm Creational (nhóm khởi tạo)



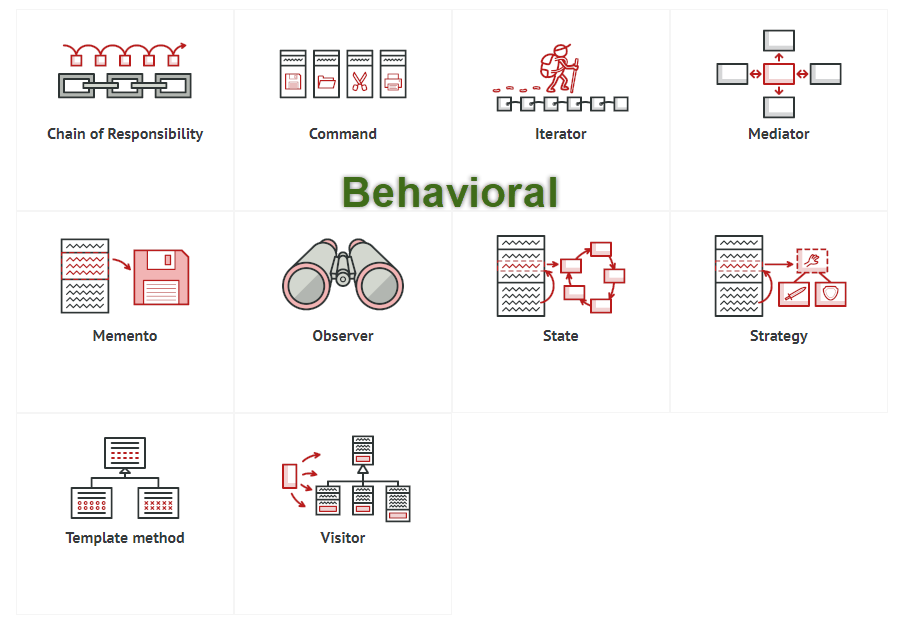
* **Singleton**:
  + Đảm bảo 1 class chỉ có 1 instance và cung cấp 1 điểm truy xuất toàn cục đến nó.
  + Tần suất sử dụng: cao trung bình.
* **Abstract Factory**:
  + Cung cấp một interface cho việc tạo lập các đối tượng (có liên hệ với nhau) mà không cần qui định lớp khi hay xác định lớp cụ thể (concrete) tạo mỗi đối tượng.
  + Tần suất sử dụng: cao.
* **Factory Method**:
  + 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**:
  + 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**:
  + 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**:
  + 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**:
  + 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**:
  + 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**:
  + Gán thêm trách nhiệm cho đối tượng (mở rộng chức năng) vào lúc chạy (dynamically).
  + Tần suất sử dụng: trung bình.
* **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:**
  + 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:**
  + 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)



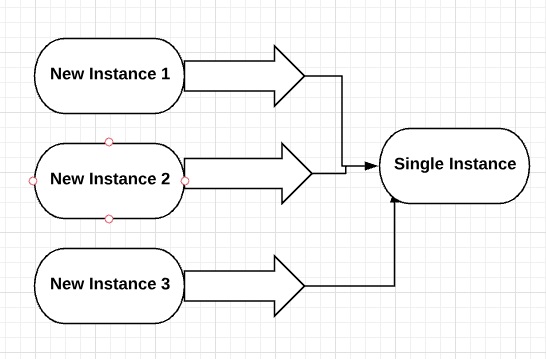
* **Chain of Responsibility:**
  + Khắc phục việc ghép cặp giữa bộ gởi và bộ nhận thông điệp. Các đối tượng nhận thông điệp được kết nối thành một chuỗi và thông điệp được chuyển dọc theo chuỗi nầy đến khi gặp được đối tượng xử lý nó. Tránh việc gắn kết cứng giữa phần tử gởi request với phần tử nhận 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:**
  + 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:**
  + Hỗ trợ việc định nghĩa biểu diễn văn phạm và bộ thông dịch cho một ngôn ngữ.
  + Tần suất sử dụng: thấp.
* **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:**
  + 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:**
  + 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:**
  + 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:**
  + 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:**
  + 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:**
  + 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:**
  + 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.

Ở 1 số bài viết có giới thiệu tới 32 mẫu design. Tuy nhiên những design đã liệt kê bên trên là các mẫu nguyên thủy nhất và hay được sử dụng nhiều nhất.

# Singleton Pattern

**Singleton pattern là gì?**

* Singleton đảm bảo chỉ duy nhất một thể hiện hay đối tượng (instance) được tạo ra.
* Lớp này cung cấp một method để truy xuất đến đối tượng duy nhất mọi lúc mọi nơi trong chương trình mà không cần khởi tạo đối tượng của lớp.



**Sử dụng khi nào?**

Khi chúng ta muốn:

* 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…
* 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 giới hạn chỉ định.

**Cách implement:** 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:

## 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.

|  |
| --- |
| **package** 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.

|  |
| --- |
| **package** 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).

|  |
| --- |
| **package** 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.

|  |
| --- |
| **package** 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*;  } } |

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** 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

|  |
| --- |
| **package** 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();  } } |

## 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** 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:

1 2018699554

2 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.

|  |
| --- |
| **package** 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.

|  |
| --- |
| **package** 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; // }* } |

|  |
| --- |
| **package** 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());  } } |

Kết quả:

|  |
| --- |
| 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.

## Singleton trong Python

|  |
| --- |
| **class** Singleton:  \_\_instance = **None** @staticmethod  **def** getInstance():  **if** Singleton.\_\_instance == **None**:  Singleton()  **return** Singleton.\_\_instance   **""" Python khong co private constructor"""  def** \_\_init\_\_(self):  **if** Singleton.\_\_instance != **None**:  **raise** Exception(**"This class is singleton"**)  **else**:  Singleton.\_\_instance = self  s2 = Singleton() print(s2)  s = Singleton.getInstance() print(s)  s2 = Singleton.getInstance() print(s2) |

# Factory Pattern

## Factory pattern là gì?

* 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.

## Khi nào sử dụng Factory Pattern:

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.

## Cách Implements

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 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.



|  |
| --- |
| **package** creational.factory;  **public interface** Shape {  **void** draw(); }  **public class** Rectangle **implements** Shape {   @Override  **public void** draw() {  System.***out***.println(**"Inside Rectangle::draw() method."**);  } }  **public class** Square **implements** Shape {   @Override  **public void** draw() {  System.***out***.println(**"Inside Square::draw() method."**);  } }  **public class** Circle **implements** Shape{  @Override  **public void** draw() {  System.***out***.println(**"Inside Circle::draw() method."**);  } }  **public class** ShapeFactory {  *//use getShape method to get object of type shape* **public** Shape getShape(String shapeType){  **if**(shapeType == **null**){  **return null**;  }  **if**(shapeType.equalsIgnoreCase(**"CIRCLE"**)){  **return new** Circle();   } **else if**(shapeType.equalsIgnoreCase(**"RECTANGLE"**)){  **return new** Rectangle();   } **else if**(shapeType.equalsIgnoreCase(**"SQUARE"**)){  **return new** Square();  }   **return null**;  } }  **public class** FactoryPatternDemo {   **public static void** main(String[] args) {  ShapeFactory shapeFactory = **new** ShapeFactory();   *//get an object of Circle and call its draw method.* Shape shape1 = shapeFactory.getShape(**"CIRCLE"**);   *//call draw method of Circle* shape1.draw();   *//get an object of Rectangle and call its draw method.* Shape shape2 = shapeFactory.getShape(**"RECTANGLE"**);   *//call draw method of Rectangle* shape2.draw();   *//get an object of Square and call its draw method.* Shape shape3 = shapeFactory.getShape(**"SQUARE"**);   *//call draw method of square* shape3.draw();  } } |

## Factory Pattern trong Python

|  |
| --- |
| **from** abc **import** ABC, abstractclassmethod **class** Shape(object):  @abstractclassmethod  **def** draw(self):  **pass  class** Circle(Shape):  **def** draw(self):  print(**"Inside Circle::draw() method."**)  **class** Rectangle(Shape):  **def** draw(self):  print(**"Inside Rectangle::draw() method."**)  **class** Square(Shape):  **def** draw(self):  print(**"Inside Square::draw() method."**)  **class** ShapeFactory():  @staticmethod  **def** getShape(typ):  targetclass = typ.capitalize()  **return** globals()[targetclass]()  *# if targetclass == "Image":  # return Image()  # elif targetclass == "Input":  # return Input()  # elif targetclass == "Flash":  # return Flash()   # button\_obj = ShapeFactory()* shapes = [**'circle'**,**'rectangle'**, **'square'**] **for** b **in** shapes:  Shape = ShapeFactory.getShape(b)  Shape.draw() |

**Lợi ích:**

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.

# Abstract Factory Pattern

## Abstract Factory pattern là gì?

**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.

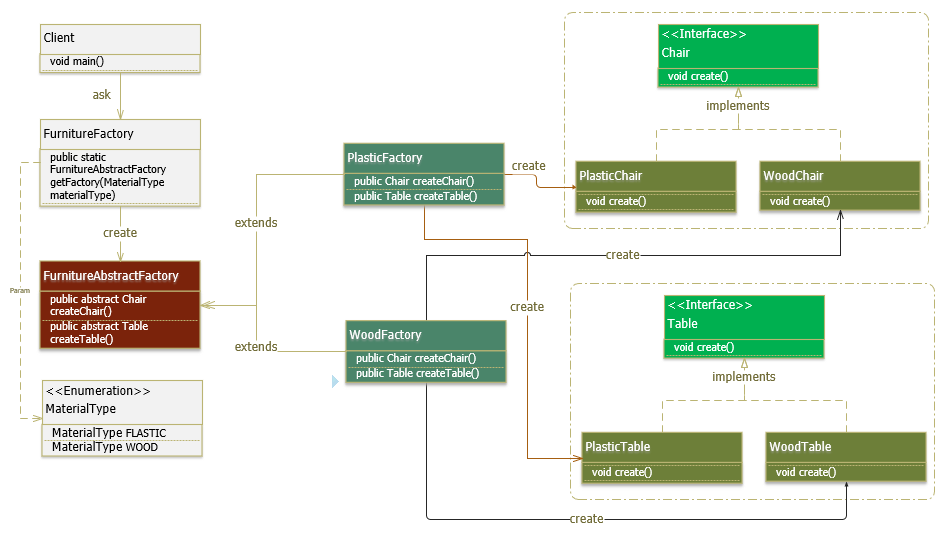
## Cách cài đặt như thế nào?

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:

****

**Super Factory Class:**

|  |
| --- |
| **package** creational.abstractFactory;  **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** creational.abstractFactory;  **public abstract class** FurnitureAbstractFactory {   **public abstract** Chair createChair();   **public abstract** Table createTable();  } |

**ConcreteFactory:**

FlasticFactory:

|  |
| --- |
| **package** creational.abstractFactory;  **public class** FlasticFactory **extends** FurnitureAbstractFactory {   @Override  **public** Chair createChair() {  **return new** PlasticChair();  }   @Override  **public** Table createTable() {  **return new** PlasticTable();  }  } |

WoodFactory

|  |
| --- |
| **package** creational.abstractFactory;  **public class** WoodFactory **extends** FurnitureAbstractFactory {   @Override  **public** Chair createChair() {  **return new** WoodChair();  }   @Override  **public** Table createTable() {  **return new** WoodTable();  } } |

**AbstractProduct và Product:**

Chair:

|  |
| --- |
| **package** creational.abstractFactory;  **public interface** Chair {  **void** create(); } |

|  |
| --- |
| **package** creational.abstractFactory;   **public class** PlasticChair **implements** Chair {  @Override  **public void** create() {  System.***out***.println(**"Create plastic chair"**);  } } |

|  |
| --- |
| **package** creational.abstractFactory;  **public class** WoodChair **implements** Chair {  @Override  **public void** create() {  System.***out***.println(**"Create wood chair"**);  } } |

Table

|  |
| --- |
| **package** creational.abstractFactory;  **public interface** Table {  **void** create(); } |

|  |
| --- |
| **package** creational.abstractFactory;  **public class** PlasticTable **implements** Table {  @Override  **public void** create() {  System.***out***.println(**"Create plastic table"**);  } } |

|  |
| --- |
| **package** creational.abstractFactory;  **public class** WoodTable **implements** Table {  @Override  **public void** create() {  System.***out***.println(**"Create wood table"**);  } } |

Material type:

|  |
| --- |
| **package** creational.abstractFactory;  **public enum** MaterialType {  ***FLASTIC***, ***WOOD*** } |

**Client:**

|  |
| --- |
| **package** creational.abstractFactory;  **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* } } |

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

* 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

# Builder Pattern

## Abstract Factory pattern là gì?

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** 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.

**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.

**Product:**

|  |
| --- |
| **package** creational.builder;  **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**;  }  }  **enum** BreadType {  ***SIMPLE***, ***OMELETTE***, ***FRIED\_EGG***, ***GRILLED\_FISH***, ***PORK***, ***BEEF***, }  **enum** OrderType {  ***ON\_SITE***, ***TAKE\_AWAY***; }  **enum** SauceType {  ***SOY\_SAUCE***, ***FISH\_SAUCE***, ***OLIVE\_OIL***, ***KETCHUP***, ***MUSTARD***; }  **enum** VegetableType {  ***SALAD***, ***CUCUMBER***, ***TOMATO*** } |

**Builder:**

|  |
| --- |
| **package** creational.builder;   **public interface** OrderBuilder {   OrderBuilder orderType(OrderType orderType);   OrderBuilder orderBread(BreadType breadType);   OrderBuilder orderSauce(SauceType sauceType);   OrderBuilder orderVegetable(VegetableType vegetableType);   Order build(); } |

ConcreteBuilder

|  |
| --- |
| **package** creational.builder;  **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** creational.builder;  **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);  } } |

**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** creational.builder;  **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** creational.builder;  **public class** ClientBank {  **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);  }  } |

## 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.

## 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.

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.

## Builder Pattern trong Python

|  |
| --- |
| **from** abc **import** abstractmethod  **class** BankAccount:  **def** \_\_init\_\_(self, name, accountNumber, address, email, newsletter, mobileBanking):  self.\_\_name = name  self.\_\_accountNumber = accountNumber  self.\_\_address = address  self.\_\_email = email  self.\_\_newsletter = newsletter  self.\_\_mobileBanking = mobileBanking   **def** getName(self):  **return** self.\_\_name   **def** getAccountNumber(self):  **return** self.\_\_accountNumber   **def** getAddress(self):  **return** self.\_\_address   **def** getNewsletter(self):  **return** self.\_\_newsletter   **def** getMobileBanking(self):  **return** self.\_\_mobileBanking   **class** AccountBuilder:  @abstractmethod  **def** withName(self, name):  **pass** @abstractmethod  **def** withAccountNumber(self, accountNumber):  **pass** @abstractmethod  **def** withAddress(self, address):  **pass** @abstractmethod  **def** build(self):  **pass    class** BankAccountBuilder(AccountBuilder):  **def** \_\_init\_\_(self):  self.\_\_name = **''** self.\_\_accountNumber = **''** self.\_\_address = **''** self.\_\_email = **''** self.\_\_newsletter = **False** self.\_\_mobileBanking = **False   def** withName(self, name):  self.\_\_name = name  **return** self   **def** withAccountNumber(self, accountNumber):  self.\_\_accountNumber = accountNumber  **return** self  **def** withAddress(self, address):  self.\_\_address = address  **return** self   **def** build(self):  bankAccount = BankAccount(self.\_\_name, self.\_\_accountNumber, self.\_\_address, self.\_\_email, self.\_\_newsletter, self.\_\_mobileBanking)  **return** bankAccount  **if** \_\_name\_\_ == **"\_\_main\_\_"**:  bObject = BankAccount.BankAccountBuilder()\  .withAccountNumber(**"00842"**)\  .withName(**"namdx"**)\  .withAddress(**"Nam Dinh City"**)\  .build()   print(bObject.getName())  print(bObject.getAccountNumber())  print(bObject.getAddress()) |

# Prototype Pattern

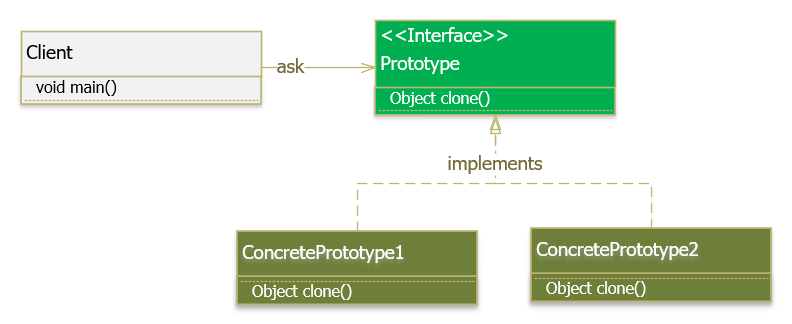
## Prototype pattern là gì?

**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** và sử dụng method **clone** để tạo object có đầy đủ thuộc tính của đối tượng ban đầu

## Cách cài đặt

****

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.

## Lợi ích

* 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.
* 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 bạn sẽ thấy rõ điều này.

## Sử dụng khi nào

* Chúng ta có một object và cần phải tạo 1 object 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.

## Prototype Pattern trong Python

|  |
| --- |
| **from** abc **import** abstractmethod **import** copy **class** Shape:  \_\_id = **None** \_\_type = **None    def** clone(self):  **pass   def** setId(self,id):  self.\_\_id = id   **def** getId(self):  **return** self.\_\_id   **def** setType(self,type):  self.\_\_type = type   **def** getType(self):  **return** self.\_\_type  **class** Cycle(Shape):  **def** \_\_init\_\_(self,id):  self.setType(**"Cycle"**)  self.setId(id)   **def** clone(self):  **return** copy.copy(self)   **class** Rectangle(Shape):  **def** \_\_init\_\_(self, id):  self.setType(**"Rectangle"**)  self.setId(id)   **def** clone(self):  **return** copy.copy(self)  **if** \_\_name\_\_ == **"\_\_main\_\_"**:  cycle1 = Cycle(1);  cycle2 = cycle1.clone();  print(**"Doi tuong cycle1"**)  print(cycle1)  print(**"id: %s , type: %s"**%(cycle1.getId(),cycle1.getType()))  print(**"Doi tuong cycle2"**)  print(cycle2)  print(**"id: %s , type: %s"** % (cycle2.getId(), cycle2.getType()))   rectangle1 = Rectangle(1);  rectangle2 = rectangle1.clone();  print(**"Doi tuong rectangle1"**)  print(rectangle1)  print(**"id: %s , type: %s"** % (rectangle1.getId(), rectangle1.getType()))  print(**"Doi tuong rectangle2"**)  print(rectangle2)  print(**"id: %s , type: %s"** % (rectangle2.getId(), rectangle2.getType())) |

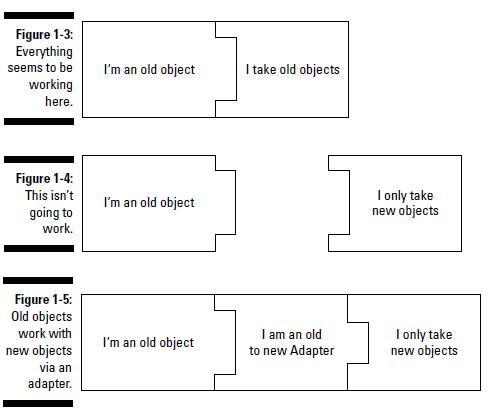
# Adapter Pattern

## Adapter pattern là gì?

**Adapter pattern** làm việc như một cầu nối giữa 2 interface không tương thích với nhau. Pattern này thuộc mẫu cấu trúc vì mẫu này kết hợp các phương thức của 2 interface độc lập. Mẫu này liên quan đến 1 class duy nhất chịu trách nhiệm kết nối các chức năng của các giao diện với nhau.

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



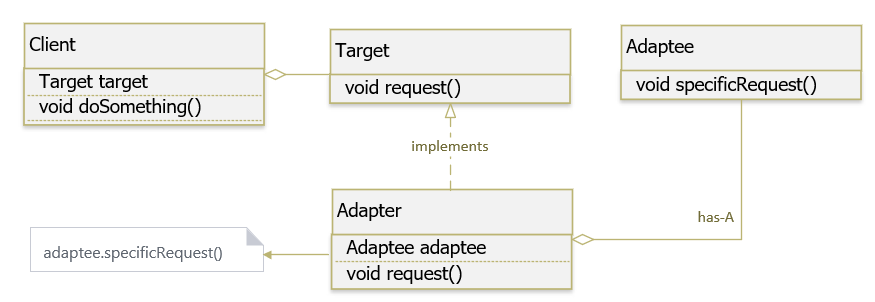
## Adapter pattern cài đặt như thế nào

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

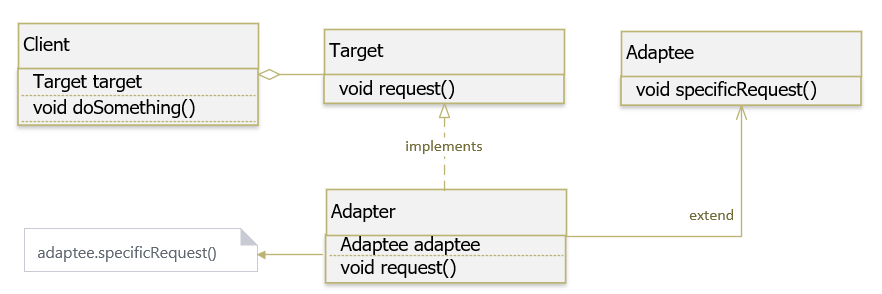
* **Adaptee**: định nghĩa interface không tương thích, cần được tích hợp vào.
* **Adapter**: lớp tích hợp, giúp interface không tương thích tích hợp được với interface đang làm việc. Thực hiện việc chuyển đổi interface cho Adaptee và kết nối Adaptee với Client.
* **Target**: một interface chứa các chức 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.

## Cách cài đặt



Java

|  |
| --- |
| **package** structural.adapter;  **public interface** MediaPlayer {  **public void** play(String audioType, String fileName); } |

|  |
| --- |
| **package** structural.adapter;  **public interface** AdvancedMediaPlayer {  **public void** playVlc(String fileName);  **public void** playMp4(String fileName); } |

|  |
| --- |
| **package** structural.adapter;  **public class** Mp4Player **implements** AdvancedMediaPlayer {   @Override  **public void** playVlc(String fileName) {  *//do nothing* }   @Override  **public void** playMp4(String fileName) {  System.***out***.println(**"Playing mp4 file. Name: "** + fileName);  } } |

|  |
| --- |
| **package** structural.adapter;  **public class** VlcPlayer **implements** AdvancedMediaPlayer {  @Override  **public void** playVlc(String fileName) {  System.***out***.println(**"Playing vlc file. Name: "** + fileName);  }   @Override  **public void** playMp4(String fileName) {  *//do nothing* } } |

|  |
| --- |
| **package** structural.adapter;  **public class** MediaAdapter **implements** MediaPlayer {   AdvancedMediaPlayer **advancedMusicPlayer**;   **public** MediaAdapter(String audioType) {   **if** (audioType.equalsIgnoreCase(**"vlc"**)) {  **advancedMusicPlayer** = **new** VlcPlayer();   } **else if** (audioType.equalsIgnoreCase(**"mp4"**)) {  **advancedMusicPlayer** = **new** Mp4Player();  }  }   @Override  **public void** play(String audioType, String fileName) {   **if** (audioType.equalsIgnoreCase(**"vlc"**)) {  **advancedMusicPlayer**.playVlc(fileName);  } **else if** (audioType.equalsIgnoreCase(**"mp4"**)) {  **advancedMusicPlayer**.playMp4(fileName);  }  } } |

|  |
| --- |
| **package** structural.adapter;  **public class** AudioPlayer **implements** MediaPlayer {  MediaAdapter **mediaAdapter**;   @Override  **public void** play(String audioType, String fileName) {   *//inbuilt support to play mp3 music files* **if** (audioType.equalsIgnoreCase(**"mp3"**)) {  System.***out***.println(**"Playing mp3 file. Name: "** + fileName);  }   *//mediaAdapter is providing support to play other file formats* **else if** (audioType.equalsIgnoreCase(**"vlc"**) || audioType.equalsIgnoreCase(**"mp4"**)) {  **mediaAdapter** = **new** MediaAdapter(audioType);  **mediaAdapter**.play(audioType, fileName);  } **else** {  System.***out***.println(**"Invalid media. "** + audioType + **" format not supported"**);  }  } } |

|  |
| --- |
| **package** structural.adapter;  **public class** AdapterPatternDemo {  **public static void** main(String[] args) {  AudioPlayer audioPlayer = **new** AudioPlayer();   audioPlayer.play(**"mp3"**, **"beyond the horizon.mp3"**);  audioPlayer.play(**"mp4"**, **"alone.mp4"**);  audioPlayer.play(**"vlc"**, **"far far away.vlc"**);  audioPlayer.play(**"avi"**, **"mind me.avi"**);  } } |

## 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** trong một ứng dụng.

## Adapter Pattern trong Python

|  |
| --- |
| **from** abc **import** abstractmethod **class** MediaPlayer:  @abstractmethod  **def** play(self, audioType, fileName):  **pass  class** AdvancedMediaPlayer:  @abstractmethod  **def** playVlc(self, fileName):  **pass** @abstractmethod  **def** playMp4(self, fileName):  **pass  class** Mp4Player(AdvancedMediaPlayer):  **def** playVlc(self, fileName):  **pass   def** playMp4(self, fileName):  print(**"Playing mp4 file. Name %s"** %(fileName))  **class** VlcPlayer(AdvancedMediaPlayer):  **def** playVlc(self, fileName):  print(**"Playing vlc file. Name: %s "**%(fileName))   **def** playMp4(self, fileName):  **pass  class** MediaAdapter(MediaPlayer):  **def** \_\_init\_\_(self, audioType):  **if** audioType == **"vlc"**:  self.\_\_advancedMusicPlay = VlcPlayer()  **elif** audioType == **"mp4"**:  self.\_\_advancedMusicPlay = Mp4Player()   **def** play(self, audioType, fileName):  **if** audioType == **"vlc"**:  self.\_\_advancedMusicPlay.playVlc(fileName)  **elif** audioType == **"mp4"**:  self.\_\_advancedMusicPlay.playMp4(fileName)  **class** AudioPlayer(MediaPlayer):  **def** play(self, audioType, fileName):  **if** audioType == **"mp3"**:  print(**"Playing mp3 file. Name: %s"** %(fileName))  **elif** audioType == **"vlc" or** audioType == **"mp4"**:  self.\_\_mediaAdapter = MediaAdapter(audioType)  self.\_\_mediaAdapter.play(audioType, fileName)  **else**:  print(**"Invalid media. %s format not supported"** % (audioType))   **if** \_\_name\_\_ == **"\_\_main\_\_"**:  audioPlayer = AudioPlayer()  audioPlayer.play(**"mp3"**, **"beyond the horizon.mp3"**)  audioPlayer.play(**"mp4"**, **"alone.mp4"**)  audioPlayer.play(**"vlc"**, **"far far away.vlc"**)  audioPlayer.play(**"avi"**, **"mind me.avi"**) |

# Bridge Pattern

## Bridge pattern là gì?

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).

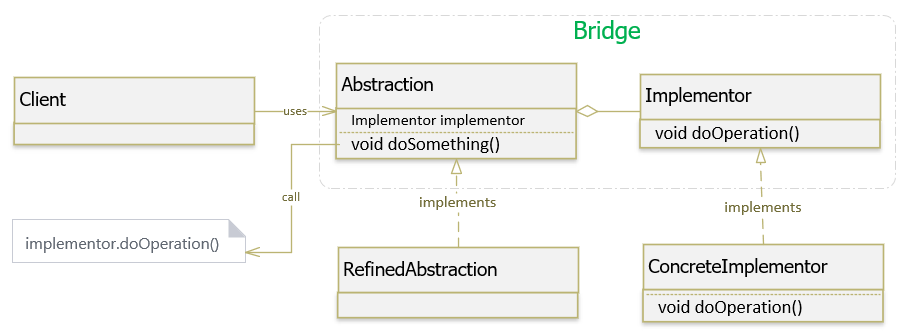
**Bridge Pattern** là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Ý tưởng của nó là tách tính trừu tượng (abstraction) ra khỏi tính hiện thực (implementation) của nó. Từ đó có thể dễ dàng chỉnh sửa hoặc thay thế mà không làm ảnh hưởng đến những nơi có sử dụng lớp ban đầu.

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** 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.

## Cách cài đặt

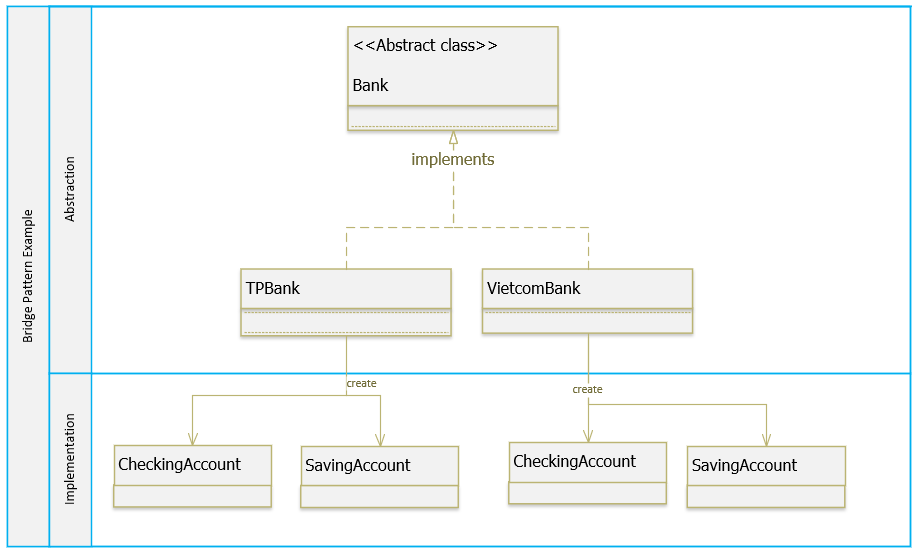


Một Bridge Pattern bao gồm các thành phần sau:

* **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.

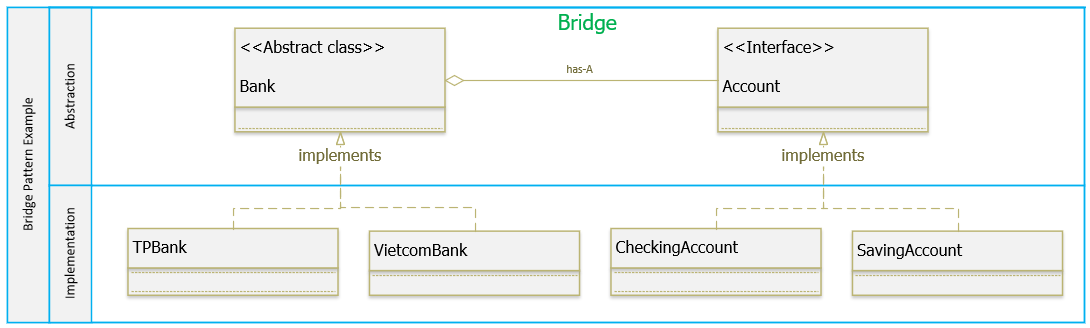
Ví dụ:

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.

|  |
| --- |
| **package** structural.bridge;  **public interface** Account {  **void** openAccount(); } |

|  |
| --- |
| **package** structural.bridge;  **public class** CheckingAccount **implements** Account {  @Override  **public void** openAccount() {  System.***out***.println(**"Checking Account"**);  } } |

|  |
| --- |
| **package** structural.bridge;  **public class** SavingAccount **implements** Account {  @Override  **public void** openAccount() {  System.***out***.println(**"Saving Account"**);  } } |

|  |
| --- |
| **package** structural.bridge;  **public abstract class** Bank {  **protected** Account **account**;   **public** Bank(Account account) {  **this**.**account** = account;  }   **public abstract void** openAccount(); } |

|  |
| --- |
| **package** structural.bridge;  **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();  } } |

|  |
| --- |
| **package** structural.bridge;  **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();  } } |

|  |
| --- |
| **package** structural.bridge;  **public class** Client {   **public static void** main(String[] args) {  Bank vietcomBank = **new** VietcomBank(**new** SavingAccount());  vietcomBank.openAccount();   Bank tpBank = **new** TPBank(**new** CheckingAccount());  tpBank.openAccount();  } } |

## Bridge Pattern trong Python

|  |
| --- |
| **from** abc **import** abstractmethod  **class** Account:  @abstractmethod  **def** openAccount(self):  **pass  class** CheckingAccount(Account):  **def** openAccount(self):  print(**"Checking Account"**)  **class** SavingAccount(Account):  **def** openAccount(self):  print(**"Saving Account"**)  **class** Bank:  account: Account  **def** \_\_init\_\_(self, account):  self.account = account   @abstractmethod  **def** createAccount(self):  **pass  class** VietcomBank(Bank):  **def** \_\_init\_\_(self, account):  super().\_\_init\_\_(account)   **def** createAccount(self):  print(**"Open your account at VietcomBank is a"**)  self.account.openAccount()   **class** TPBank(Bank):  **def** \_\_init\_\_(self, account):  super().\_\_init\_\_(account)   **def** createAccount(self):  print(**"Open your account at TPBank is a"**)  self.account.openAccount()  **if** \_\_name\_\_ == **"\_\_main\_\_"**:  vietcombank = VietcomBank(SavingAccount())  vietcombank.createAccount()   tpbank = TPBank(CheckingAccount())  tpbank.createAccount() |

## 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.

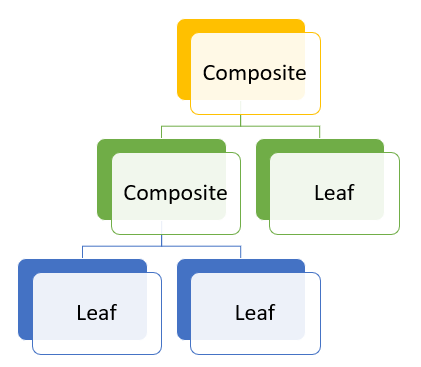
## Sử dụng Bridge Pattern khi nào

* 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.

# Composite Pattern

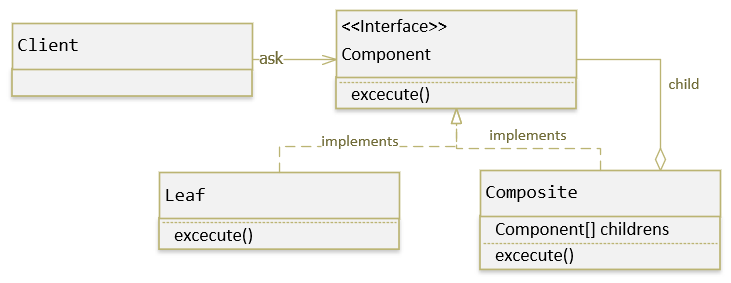
## Composite pattern là gì?

**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.

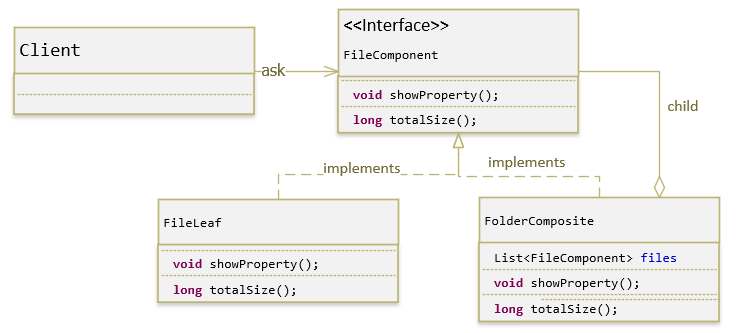
## Cài đặt Composite Pattern



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



|  |
| --- |
| **package** structural.composite;  **public interface** FileComponent {  **void** showProperty();  **long** totalSize(); } |

|  |
| --- |
| **package** 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** + **"]"**);  } } |

|  |
| --- |
| **package** structural.composite;  **import** java.util.ArrayList; **import** java.util.List;  **public class** FolderComposite **implements** FileComponent {   **private** List<FileComponent> **files** = **new** ArrayList<>();   **public** FolderComposite() {  }   **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;  } } |

|  |
| --- |
| **package** 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());  } } |

## 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.

## Composite Pattern trong python

|  |
| --- |
| **from** abc **import** abstractmethod  **class** FileComponent:  @abstractmethod  **def** showProperty(self):  **pass** @abstractmethod  **def** totalSize(self):  **pass  class** FileLeaf(FileComponent):  **def** \_\_init\_\_(self, name, size):  super().\_\_init\_\_()  self.\_\_name = name  self.\_\_size = size   **def** showProperty(self):  print(**"FileLeaf[name=%s, size=%s]"** %(self.\_\_name, self.\_\_size))   **def** totalSize(self):  **return** self.\_\_size  **class** FolderComposite(FileComponent):  \_\_files = []   **def** showProperty(self):  **for** f **in** self.\_\_files:  f.showProperty()   **def** totalSize(self):  total = 0  **for** f **in** self.\_\_files:  total += f.totalSize()  **return** total   **def** \_\_init\_\_(self, files) -> **None**:  super().\_\_init\_\_()  self.\_\_files = files  **if** \_\_name\_\_ == **"\_\_main\_\_"**:  file1 = FileLeaf(**"File 1"**,10)  file2 = FileLeaf(**"File 2"**,5)  folder = FolderComposite([file1, file2])  folder.showProperty()  print(**"Total size: %s"** %(folder.totalSize())) |

# Decorator Pattern

## Decorator pattern là gì?

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**

**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)

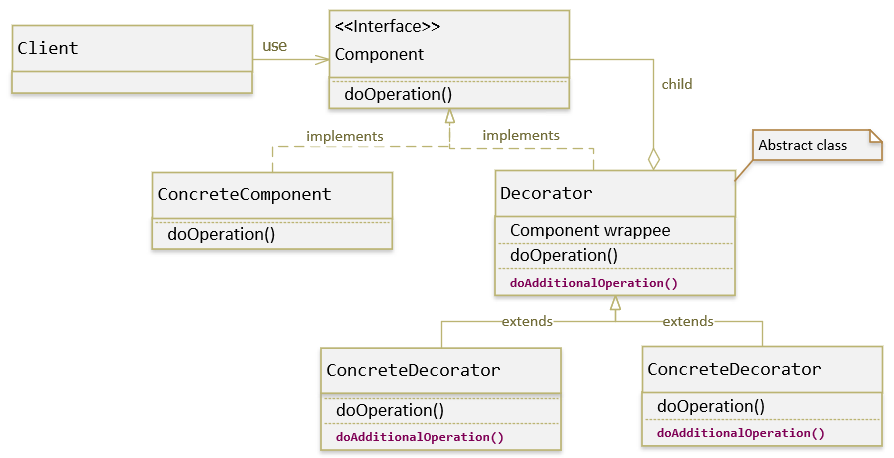
Decorator pattern sử dụng composition thay vì inheritance (thừa kế) để mở rộng đối tượng. Decorator pattern còn được gọi là **Wrapper (Smart Proxy)**.

## Cách cài đặt

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.

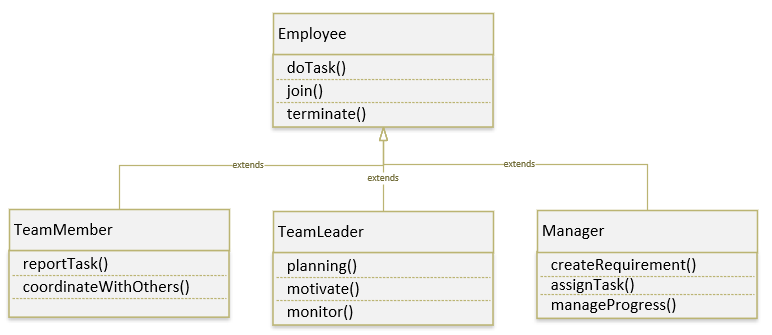
**Ví dụ:**

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:

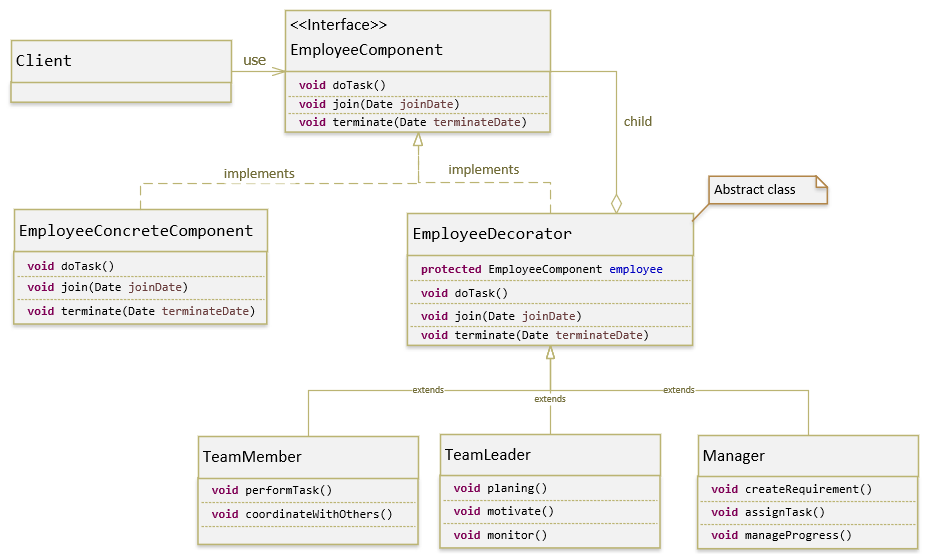


Bất cứ khi nào một thành viên trong nhóm trở thành một Team Lead, chúng ta phải tạo một đối tượng mới của Team Lead và đối tượng trước đó tham chiếu vào nhân viên đó (Team Member trong nhóm) có thể bị hủy hoặc lưu trữ. Đó không phải là cách tiếp cận được khuyến nghị khi nhân viên vẫn là một phần của 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.

|  |
| --- |
| **package** 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 calendar = Calendar.*getInstance*();  calendar.add(Calendar.***YEAR***, 2);  terminate(calendar.getTime());  } } |

|  |
| --- |
| **package** 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** doTask() {  }   @Override  **public void** join(Date joinDate) {  System.***out***.println(**this**.getName() + **" join on "** + formatDate(joinDate));   }   @Override  **public void** terminate(Date terminateDate) {  System.***out***.println(**this**.getName() + **" terminate on "** + formatDate(terminateDate));  } } |

|  |
| --- |
| **package** structural.decorator;  **import** java.util.Date;  **public abstract class** EmployeeDecorator **implements** EmployeeComponent {  **protected** EmployeeComponent **employeeComponent**;   **public** EmployeeDecorator(EmployeeComponent employeeComponent) {  **this**.**employeeComponent** = employeeComponent;  }   @Override  **public** String getName() {  **return employeeComponent**.getName();  }   @Override  **public void** join(Date joinDate) {  **employeeComponent**.join(joinDate);  }   @Override  **public void** terminate(Date terminateDate) {  **employeeComponent**.terminate(terminateDate);  } } |

|  |
| --- |
| **package** structural.decorator;  **import** java.util.Date;  **public class** Manager **extends** EmployeeDecorator {   **public** Manager(EmployeeComponent employeeComponent) {  **super**(employeeComponent);  }  **public void** createRequirement(){  System.***out***.println(**this**.getName() + **" create some requirements"**);  }  **public void** assignTask(){  System.***out***.println(**this**.getName() + **" assign the task"**);  }   **public void** manageProcess(){  System.***out***.println(**this**.getName() + **" manage process"**);  }   @Override  **public void** doTask() {  **employeeComponent**.doTask();  createRequirement();  assignTask();  manageProcess();  } } |

|  |
| --- |
| **package** structural.decorator;  **public class** TeamLeader **extends** EmployeeDecorator {  **public** TeamLeader(EmployeeComponent employeeComponent) {  **super**(employeeComponent);  }  **public void** planning(){  System.***out***.println(**this**.getName() + **" is planning"**);  }  **public void** motivate(){  System.***out***.println(**this**.getName() + **" is motivating"**);  }   **public void** monitor(){  System.***out***.println(**this**.getName() + **" is monitoring"**);  }   @Override  **public void** doTask() {  **employeeComponent**.doTask();  planning();  motivate();  monitor();  } } |

|  |
| --- |
| **package** structural.decorator;  **public class** TeamMember **extends** EmployeeDecorator {  **public** TeamMember(EmployeeComponent employeeComponent) {  **super**(employeeComponent);  }  **public void** reportTask(){  System.***out***.println(**this**.getName() + **" report task to manager"**);  }  **public void** coordinateWithOther(){  System.***out***.println(**this**.getName() + **" coordinates with Others"**);  }   @Override  **public void** doTask() {  **employeeComponent**.doTask();  reportTask();  coordinateWithOther();  } } |

|  |
| --- |
| **package** structural.decorator;  **public class** Client {  **public static void** main(String[] args) {  System.***out***.println(**"--NORMAL EMPLOYEE:--"**);  EmployeeComponent employee = **new** EmployeeConcreteComponent(**"Nguyen Van A"**);  employee.showBasicInformation();  employee.doTask();   System.***out***.println();  System.***out***.println(**"--TEAM LEADER EMPLOYEE:--"**);  EmployeeComponent teamLeader = **new** TeamLeader(employee);  teamLeader.showBasicInformation();  teamLeader.doTask();   System.***out***.println();  System.***out***.println(**"--MANAGER EMPLOYEE:--"**);  EmployeeComponent manager = **new** Manager(employee);  manager.showBasicInformation();  manager.doTask();   System.***out***.println();  System.***out***.println(**"--MANAGER AND TEAM LEADER EMPLOYEE:--"**);  EmployeeComponent teamLeaderAnManager = **new** Manager(teamLeader);  teamLeaderAnManager.showBasicInformation();  teamLeaderAnManager.doTask();  } } |

## Lợi ích của Decorator Pattern

12

* 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** như định nghĩa của GOF.
* Cả hai sử dụng cách **composition** 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).

## Decorator Pattern trong Python

|  |
| --- |
| **from** abc **import** abstractmethod **from** datetime **import** datetime,date  **class** EmployeeComponent:   @abstractmethod  **def** getName(self):  **pass** @abstractmethod  **def** doTask(self):  **pass** @abstractmethod  **def** join(self, joinDate):  **pass** @abstractmethod  **def** terminate(self, terminateDate):  **pass  def** formatDate(self, theDate):  **return** theDate.strftime(**"%d/%m/%Y"**)  **def** showBasicInformation(self):  print(**"------------------------------"**)  print(**"The basic information of %s"** %(self.getName()))  now = datetime.now()  self.join(now)  terminateDate = date(now.year + 2, now.month, now.day)  self.terminate(terminateDate)  **class** EmployeeConcreteComponent(EmployeeComponent):  **def** \_\_init\_\_(self, name) -> **None**:  super().\_\_init\_\_()  self.\_\_name = name   **def** getName(self):  **return** self.\_\_name   **def** doTask(self):  **pass   def** join(self, joinDate):  print(**"%s join on %s"** %(self.getName(),self.formatDate(joinDate)) )   **def** terminate(self, terminateDate):  print(**"%s terminate on %s"** %(self.getName(),self.formatDate(terminateDate)) )  **class** EmployeeDecorator(EmployeeComponent):  \_employee:EmployeeComponent   **def** \_\_init\_\_(self, employee) -> **None**:  super().\_\_init\_\_()  self.\_employee = employee   **def** getName(self):  self.\_employee.getName()   **def** join(self, joinDate):  self.\_employee.join(joinDate)   **def** terminate(self, terminateDate):  self.\_employee.terminate(terminateDate)  **class** Manager(EmployeeDecorator):   **def** \_\_init\_\_(self, employee) -> **None**:  super().\_\_init\_\_(employee)  **def** createRequirement(self):  print(**"%s create some requirements"** %(self.getName()))  **def** assignTask(self):  print(**"%s assign task"** % (self.getName()))  **def** manageProcess(self):  print(**"%s manage process"** % (self.getName()))  **def** doTask(self):  self.\_employee.doTask()  self.createRequirement()  self.assignTask()  self.manageProcess()  **class** TeamLeader(EmployeeDecorator):   **def** \_\_init\_\_(self, employee) -> **None**:  super().\_\_init\_\_(employee)  **def** planning(self):  print(**"%s is planning"** %(self.getName()))  **def** motivate(self):  print(**"%s is motivating"**%(self.getName()))  **def** monitor(self):  print(**"%s is monitor"** %(self.getName()))   **def** doTask(self):  self.\_employee.doTask()  self.planning()  self.motivate()  self.monitor()  **if** \_\_name\_\_ == **"\_\_main\_\_"**:  print(**"--NORMAL EMPLOYEE:----"**)  employee = EmployeeConcreteComponent(**"Nguyen Van A"**)  employee.showBasicInformation()  employee.doTask()  print()   print(**"--TEAM LEADER EMPLOYEE:----"**)  teamleader = TeamLeader(employee)  teamleader.showBasicInformation()  teamleader.doTask()  print()   print(**"--MANAGER EMPLOYEE:----"**)  manager = Manager(employee)  manager.showBasicInformation()  manager.doTask()  print()   print(**"--TEAM LEADER AND MANAGER:----"**)  teamLeaderAnManager = TeamLeader(manager)  teamLeaderAnManager.showBasicInformation()  teamLeaderAnManager.doTask() |

# Facade Pattern

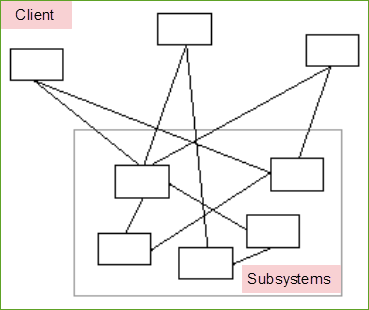
## Facade pattern là gì?

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. **Facade Pattern** được sử dụng để giải quyết vấn đề này.

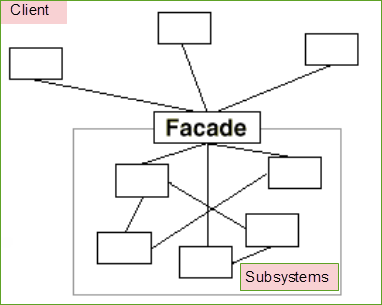
**Facade Pattern** là một trong những Pattern thuộc nhóm cấu trúc (Structural Pattern). Pattern này cung cấp một giao diện chung đơn giản thay cho một nhóm các giao diện có trong một hệ thống con (subsystem). Facade Pattern định nghĩa một giao diện ở một cấp độ cao hơn để giúp cho người dùng có thể dễ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

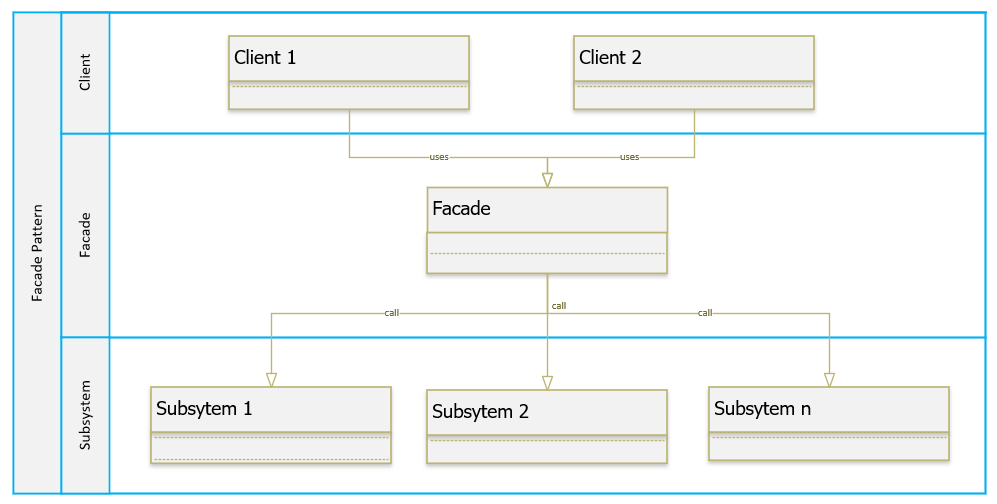


Khi có Facade, nó trông giống như sau:



**Facade Pattern** tương tự với **Adapter Pattern**. 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 **Facade Pattern** cho phép bao bọc mã nguồn gốc để nó có thể giao tiếp với mã nguồn khác dễ dàng hơn.

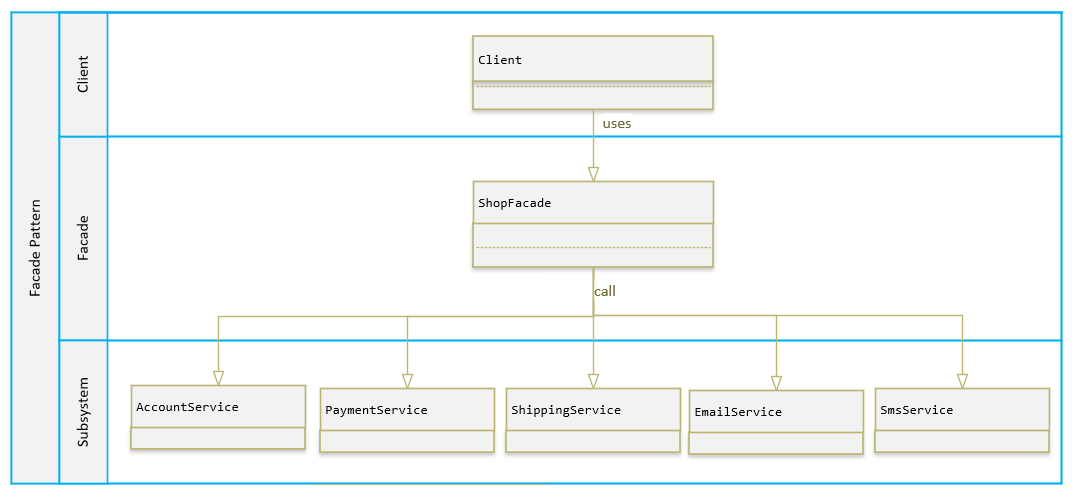
## Cài đặt Facade Pattern



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** bởi vì chỉ cần duy nhất một đối tượng Facade.



|  |
| --- |
| **package** structural.facade;  **public class** CmService {  **protected void** getSubInfo(String account){  System.***out***.println(**"Call CM Service to get sub info of:"**+account);  }   **protected void** getSubType(String account){  System.***out***.println(**"Call CM Service to get sub type of:"**+account);  } } |

|  |
| --- |
| **package** structural.facade;  **public class** CcService {  **public void** getHistoryCalling(String account) {  System.***out***.println(**"Call CC Service to get History calling of "** + account);  }   **public void** getComplaintHistory(String account) {  System.***out***.println(**"Call CC Service to get complain history of "** + account);  } } |

|  |
| --- |
| **package** structural.facade;  **public class** SubFacade {  **private** CcService **ccService**;  **private** CmService **cmService**;  **private static final** SubFacade ***INSTANCE*** = **new** SubFacade();   **public static** SubFacade getInstance() {  **return *INSTANCE***;  }   **private** SubFacade() {  **this**.**ccService** = **new** CcService();  **this**.**cmService** = **new** CmService();  }   **public void** getAllInforOfSub(String account){  **this**.**cmService**.getSubInfo(account);  **this**.**cmService**.getSubType(account);  }   **public void** getHistoryComplaintOfSub(String account){  **this**.**ccService**.getComplaintHistory(account);  **this**.**ccService**.getHistoryCalling(account);  }  } |

|  |
| --- |
| **package** structural.facade;  **public class** Client {  **public static void** main(String[] args) {  SubFacade.*getInstance*().getAllInforOfSub(**"h004\_gftth\_aaa"**);  SubFacade.*getInstance*().getHistoryComplaintOfSub(**"h004\_gftth\_aaa"**);  } } |

## 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.

## Facade Pattern trong Python

|  |
| --- |
| **from** abc **import** ABC **class** CcService:  **def** getHistoryCalling(self,account):  print(**"Call CC Service to get History calling of %s"**%(account))   **def** getComplaintHistory(self,account):  print(**"Call CC Service to get complain history of %s"**%(account))  **class** CmService:  **def** getSubInfo(self,account):  print(**"Call CM Service to get sub info of: %s"** %(account))   **def** getSubType(self,account):  print(**"Call CM Service to get sub type of: %s"**%(account))    **class** SubFacade:  \_\_ccService: **None** \_\_cmService: **None   def** \_\_init\_\_(self):  self.\_\_ccService = CcService()  self.\_\_cmService = CmService()   **def** getAllInforOfSub(self, account):  self.\_\_cmService.getSubInfo(account)  self.\_\_cmService.getSubType(account)   **def** getHistoryComplaintOfSub(self,account):  self.\_\_ccService.getComplaintHistory(account)  self.\_\_ccService.getHistoryCalling(account)  **if** \_\_name\_\_ == **"\_\_main\_\_"**:  \_\_instance = SubFacade()  \_\_instance.getAllInforOfSub(**"h004\_gftth\_aaa"**)  \_\_instance.getHistoryComplaintOfSub(**"h004\_gftth\_aaa"**) |

# Tham khảo

Top các cuốn sách hay về Design Pattern:

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

Các link tham khảo:

<https://www.tutorialspoint.com/python_design_patterns/index.htm>

<https://www.tutorialspoint.com/design_pattern/index.htm>

<https://gpcoder.com/4164-gioi-thieu-design-patterns/>