Java Design Pattern

Design Pattern là giải pháp kỹ thuật thiết kế chương trình để giải quyết tối ưu các vấn đề chung, thường gặp trong lập trình.

- Java Design Pattern overview - Mẫu thiết kế trong Java
- Java Singleton Pattern - Mẫu thiết kế Singleton trong Java
- Java Abstract Factory Pattern - Mẫu thiết kế Abstract Factory trong Java
- Java Factory Design Pattern - Mẫu thiết kế Factory Design trong Java
- Java Builder Pattern - Mẫu thiết kế Builder trong Java

Java Design Pattern overview - Mẫu thiết kế trong Java

Design Pattern là giải pháp kỹ thuật thiết kế (hay còn gọi là mẫu thiết kế) chương trình để giải quyết tối ưu các vấn đề chung, thường gặp trong lập trình.

1. Design Pattern là gì

Design patterns là các giải pháp đã được tối ưu hóa, được tái sử dụng cho các vấn đề lập trình mà chúng ta gặp phải hàng ngày. Nó là một khuôn mẫu có thể được áp dụng vào mỗi trường hợp cụ thể.

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

Để xây dựng và áp dụng Design Pattern, cần nắm rõ các kiến thức sau:

- Bốn đặc tính của OOP: Thừa kế, Đa hình, Trừu tượng, Bao đóng.
- Khái niệm interface và abstract. Cái này cực kỳ quan trọng, để hiểu và áp dụng 2 khái niệm này có thể sẽ mất một thời gian, nhưng khi bạn nắm chắc nó bạn sẽ thấy nó thực sự cần thiết.

2. Lợi ích của Design Pattern

- Các mẫu thiết kế đã được xác định và cung cấp cách tiếp cận chuẩn theo ngành để giải quyết vấn đề lặp lại, vì vậy sẽ tiết kiệm được thời gian nếu chúng ta sử dụng mẫu thiết kế hợp lý.
- Sử dụng các mẫu thiết kế thúc đẩy khả năng tái sử dụng dẫn đến phát triển phần mềm nhanh hơn. Nó giúp giảm tổng chi phí của sản phẩm phần mềm.
- Khi áp dụng 1 mẫu thiết kế, nó làm cho code dễ hiểu và gỡ lỗi.
- Design Pattern giúp bạn tái sử dụng mã lệnh và dễ 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 k mất quá nhiều thời gian.

3. Các loại Design Pattern

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

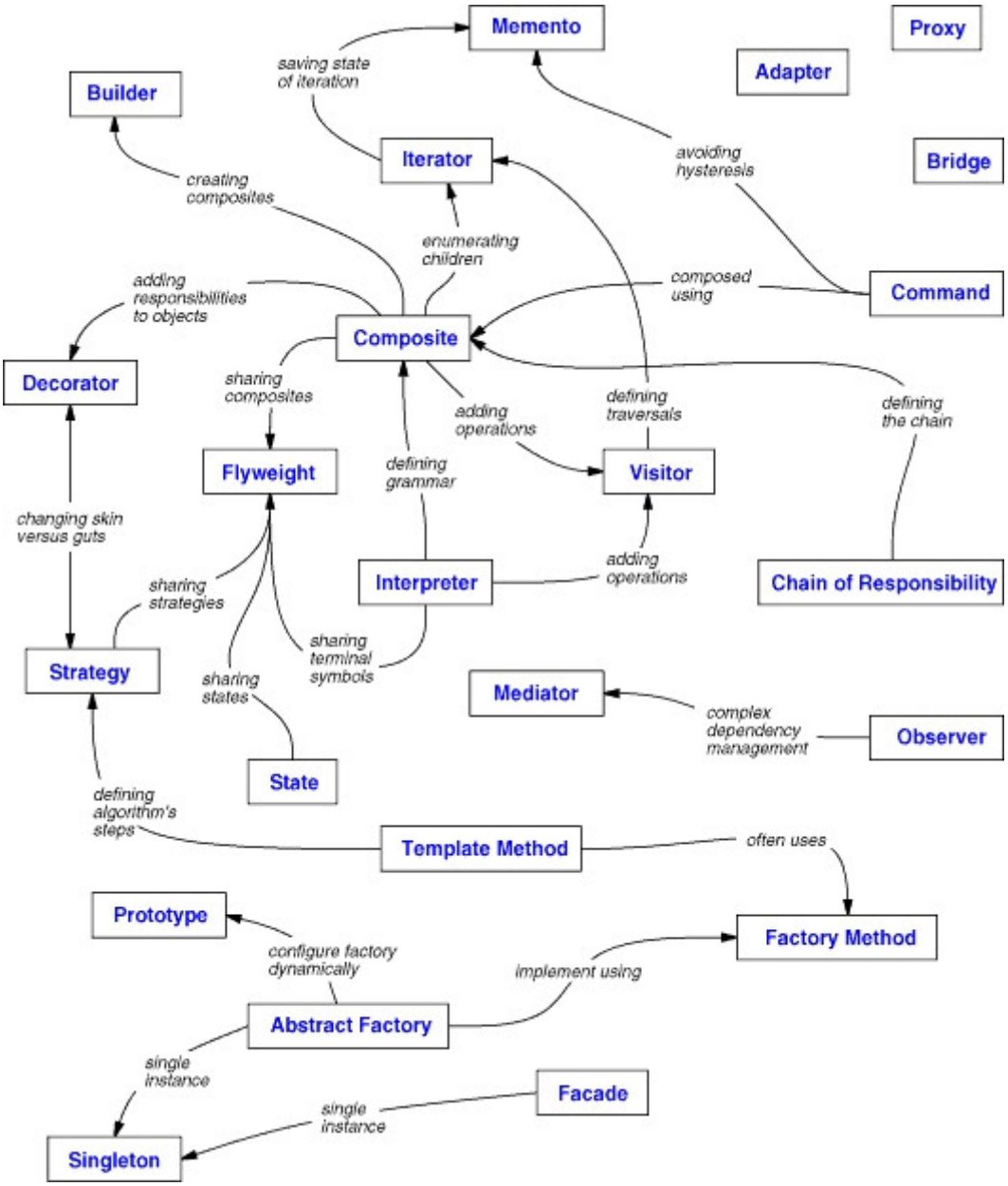
Bốn tác giả trên được biết đến rộng rãi dưới tên Gang of Four (bộ tứ). Theo quan điểm của bốn người, design pattern chủ yếu được dựa theo những quy tắc sau đây về thiết kế hướng đối tượng.

Có 4 loại Design Pattern sau:

TT	Design Pattern	Mô tả	Các loại

1	**Creational Patterns**	Nhóm này cung cấp phương pháp tạo ra các đối tượng một cách linh hoạt hơn. Nghĩa là quyết định đối tượng nào được tạo ra tuỳ thuộc vào trường hợp sử dụng nhất định.	5 mẫu: Factory Method, Abstract Factory, Builder, Prototype, Singleton
2	**Structural Pattern**	Nhóm này liên quan đến sự kết hợp giữa các đối tượng với nhau	7 mẫu: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
3	**Behavioral Patterns**	Mẫu thiết kế này trình bày phương pháp thiết kế liên quan đến hành vi của các đối tượng.	11 mẫu: Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor
4	**J2EE Pattern**	Nhóm này cung cấp phương pháp thiết kế chương trình theo mô hình nhiều tầng (multiple tier)	

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



3.1. Nhóm Creational (nhóm khởi tạo)

Nhóm này cung cấp phương pháp tạo ra các đối tượng một cách linh hoạt hơn. Nghĩa là quyết định đối tượng nào được tạo ra tuỳ thuộc vào trường hợp sử dụng nhất định.

- **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.

3.2. Nhóm Structural (nhóm cấu trúc)

Nhóm này liên quan đến sự kết hợp giữa các đối tượng với nhau.

- **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.

3.3. Nhóm Behavioral (nhóm hành vi/ tương tác)

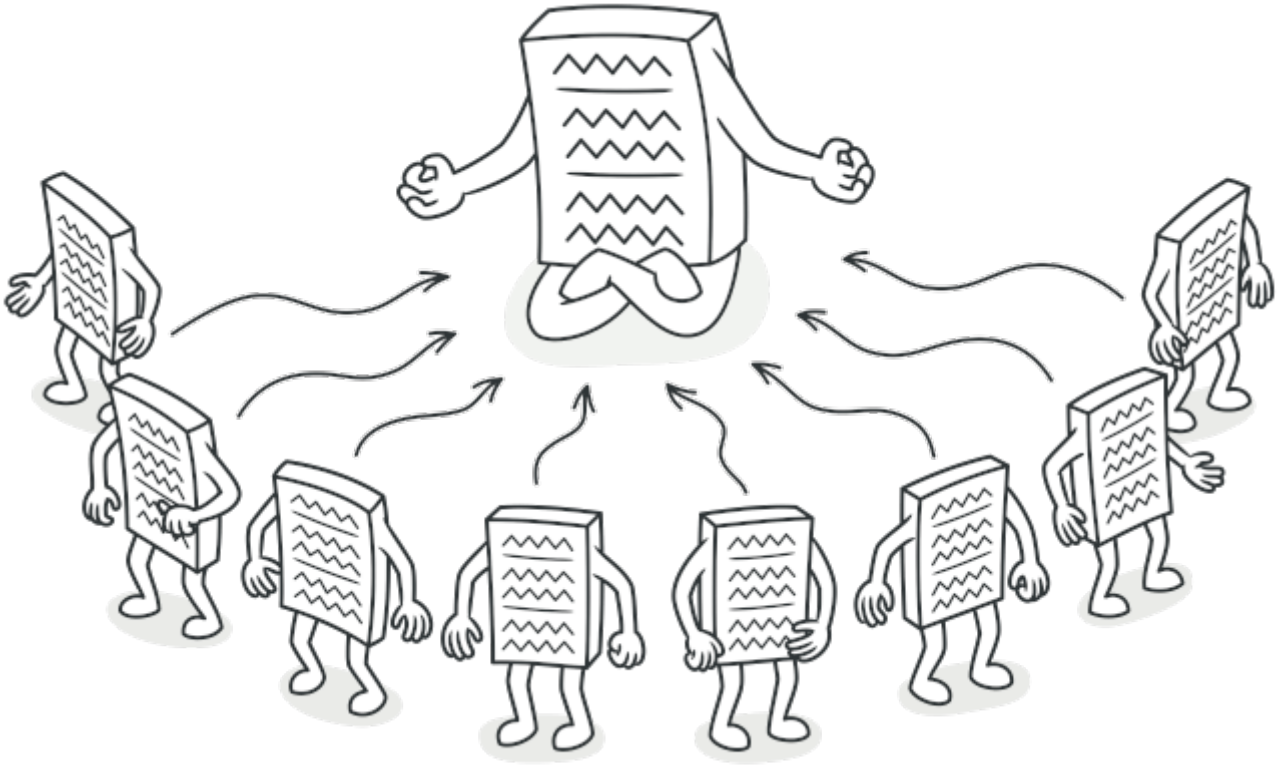
Mẫu thiết kế này trình bày phương pháp thiết kế liên quan đến hành vi của các đối tượng.

- **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.

Java Singleton Pattern - Mẫu thiết kế Singleton trong Java

Singleton là một Design Pattern cho phép bạn đảm bảo rằng một lớp chỉ có một thể hiện, và cung cấp một truy xuất toàn cục cho thể hiện này.

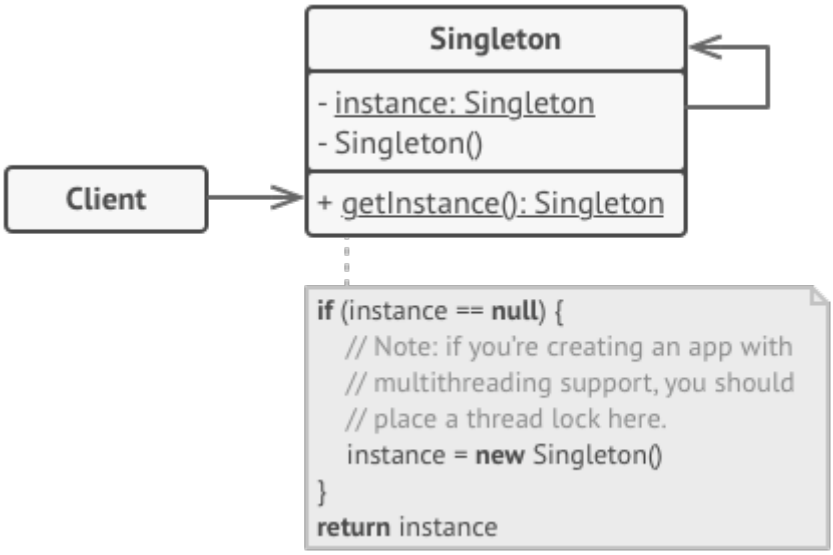


1. Tại sao cần có Singleton Pattern

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

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

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



Sử dụng Singleton khi chúng ta muốn:

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

Một số trường hợp sử dụng của Singleton Pattern thường gặp:

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

2. Thực thi Singleton Pattern

Có rất nhiều cách để implement Singleton Pattern bằng các dựa trên nguyên tắc dưới đây cơ bản dưới đây:

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

2.1. 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.

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

```
public class SingletonPatternDemo {

   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

2.2. Static block initialization

Vì `static block` sẽ chỉ được gọi một lần, chúng ta có thể sử dụng `static block` để phát triển lớp singleton. Dưới đây ví dụ cho thấy cách tạo các lớp singleton bằng cách sử dụng `static block`:

- Để tạo lớp singleton, hãy tạo constructor thành private, để bạn không thể tạo đối tượng bên ngoài lớp.
- Tạo một biến private static có cùng loại class, để đối tượng được tạo sẽ được trỏ đến tham chiếu này. Bây giờ tạo `static block` và tạo đối tượng bên trong `static block`. Vì `static block` sẽ chỉ được gọi một lần, nên đối tượng sẽ chỉ được tạo một lần.

```
public class MyStaticSingleton {

    public static void main(String a[]){
        MySingleton ms = MySingleton.getInstance();
        ms.testSingleton();
    }
}

class MySingleton{

    private static MySingleton instance;

    static{
        instance = new MySingleton();
    }

    private MySingleton(){
        System.out.println("Creating MySingleton object...");
    }

    public static MySingleton getInstance(){
        return instance;
    }

    public void testSingleton(){
        System.out.println("Hey.... Instance got created...");
    }
}
```

Output:

```
Creating MySingleton object...
Hey.... Instance got created...
```

2.3. 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).

```
public class LazyInitializedSingleton {

    private static LazyInitializedSingleton instance;

    private LazyInitializedSingleton() {
    }

    public static LazyInitializedSingleton getInstance() {
        if (instance == null) {
            instance = new LazyInitializedSingleton();
        }
        return instance;
    }
}
```

- Ưu điểm: Cách này đã khắc phục được nhược điểm của cách **Eager initialization**, chỉ khi nào **getInstance()** được gọi thì instance mới được khởi tạo.

- Nhược điểm:

- 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.

2.4. Thread Safe Singleton

Để đảm bảo khi khởi tạo Singletong trong Thread, chúng ta có thể sử dụng phương thức **synchronized** trong 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.

```
// Java program to create Thread Safe
// Singleton class
public class GFG {
    // private instance, so that it can be
    // accessed by only by getInstance() method
    private static GFG instance;
```

```
    private GFG() {
        // private constructor
    }

    //synchronized method to control simultaneous access
    synchronized public static GFG getInstance() {
        if (instance == null) {
            // if instance is null, initialize
            instance = new GFG();
        }
        return instance;
    }
}
```

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

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

2.5. Lazy initialization with Double check locking

Để implement theo cách này, chúng ta sẽ kiểm tra sự tồn tại thể hiện của lớp, với sự hỗ 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.

```
// Java code to explain double check locking
public class GFG {
    // private instance, so that it can be
    // accessed by only by getInstance() method
    private static volatile GFG instance;

    private GFG() {
        // private constructor
    }

    public static GFG getInstance() {
```

```
        if (instance == null) {
            //synchronized block to remove overhead
            synchronized(GFG.class) {
                if (instance == null) {
                    // if instance is null, initialize
                    instance = new GFG();
                }
            }
        }
        return instance;
    }
}
```

2.6. Bill Pugh Singleton Implementation

Trước Java5, mô hình bộ nhớ có rất nhiều vấn đề và các phương thức trên gây ra lỗi trong các kịch bản nhất định trong môi trường đa luồng. Vì vậy, Bill Pugh đã đề xuất một khái niệm về các lớp inner static để sử dụng cho singleton.

```
// Java code for Bill Pugh Singleton Implementaion
public class GFG {

    private GFG() {
        // private constructor
    }

    // Inner class to provide instance of class
    private static class BillPughSingleton {
        private static final GFG INSTANCE = new GFG();
    }

    public static GFG getInstance() {
        return BillPughSingleton.INSTANCE;
    }
}
```

Khi lớp singleton được load, lớp bên trong không được load và do đó không tạo đối tượng khi load class. Inner class chỉ được tạo khi phương thức getInstance() được gọi. Vì vậy, nó có vẻ giống như eager initialization nhưng đó là lazy initialization.

Đây là cách tiếp cận được sử dụng rộng rãi nhất vì nó không sử dụng đồng bộ hóa.

2.7. 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.

```
/**
 * 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**.

3. Tổng kết

1. `Eager initialization` là cách dễ dàng nhất để thực thi Singleton nhưng nó có nhược điểm khi instance được khởi tạo mà có thể không được sử dụng.
2. Sử dụng `Static block` trong Eager initialization, chúng ta có thể cung cấp xử lý ngoại lệ và kiểm soát được instance.
3. Sử dụng `synchronized` chúng ta có thể tạo Singleton class trong môi trường đa luồng multi-threading nhưng nó có thể gây chậm chương trình, một cách tối ưu hơn là sử dụng cơ chế Double check locking.
4. `Bill Pugh` là phương pháp được sử dụng rộng rãi nhất trong thực thi Singleton do việc khai báo đơn giản và có nhiều ưu điểm.

Java Abstract Factory Pattern - Mẫu thiết kế Abstract Factory trong Java

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.

1. Tại sao cần có Abstract 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.

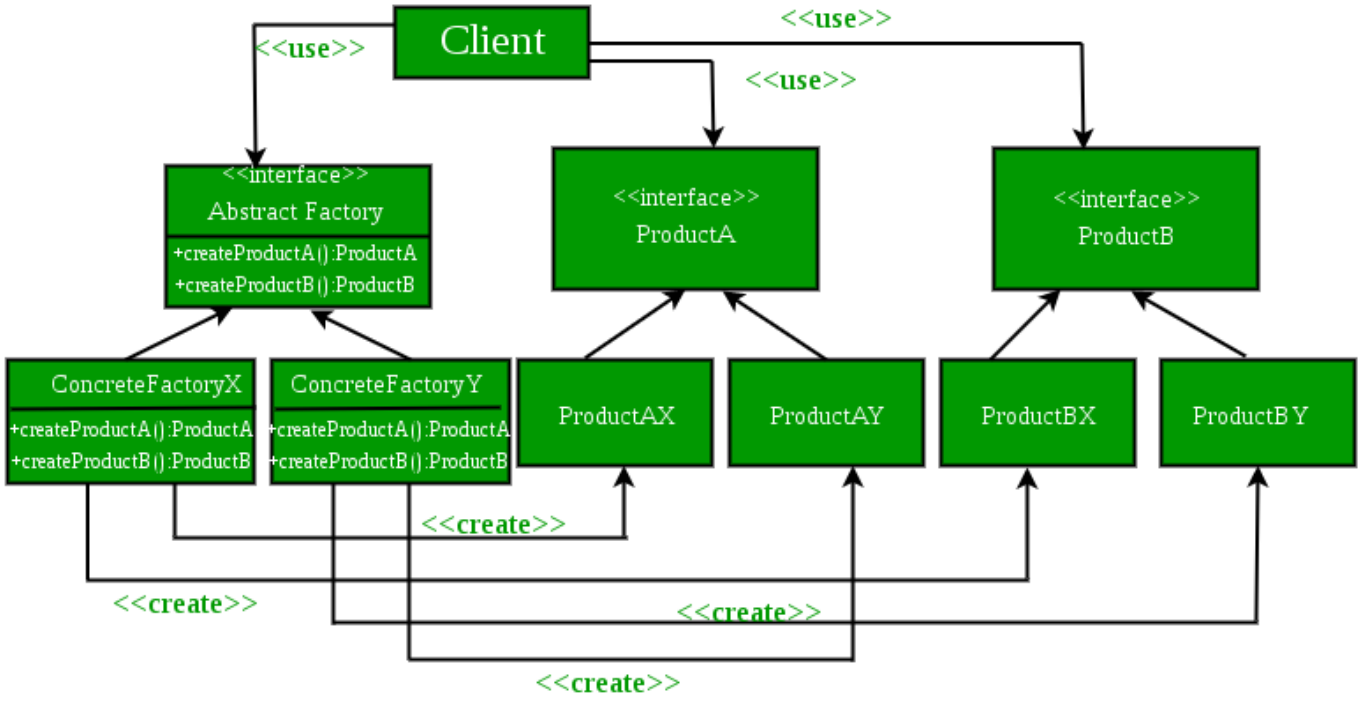
- Cung cấp hướng tiếp cận với Interface thay thì các implement, che giấu sự phức tạp của việc khởi tạo các đối tượng với người dùng (client), độc lập giữa việc khởi tạo đối tượng và hệ thống sử dụng, ...
- Giúp tránh được việc sử dụng điều kiện logic bên trong **Factory Pattern**. Khi một **Factory Method** 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.

2. Thực thi Abstract Factory Pattern

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 du: Chúng ta có các nhà máy sản xuất Ô tô được đặt tại IndiaCarFactory, USACarFactory và DefaultCarFactory. Và chương trình của chúng ta cần đủ thông minh để xác định được vị trí mà nhà máy được đặt, vì vậy chúng ta sẽ có thể sử dụng nhà máy ô tô phù hợp mà không cần biết việc thực hiện nhà máy ô tô nào sẽ được sử dụng trong nội bộ. Điều này cũng tiết kiệm cho chúng ta từ một người gọi nhầm nhà máy cho một vị trí cụ thể.



`CarType.class`

```
// Java Program to demonstrate the working of Abstract Factory Pattern
enum CarType {
    MICRO,
    MINI,
    LUXURY
}
```

`Car.class`

```
abstract class Car {
    Car(CarType model, Location location) {
        this.model = model;
        this.location = location;
    }

    abstract void construct();

    CarType model = null;
```

```
    Location location = null;

    CarType getModel() {
        return model;
    }

    void setModel(CarType model) {
        this.model = model;
    }

    Location getLocation() {
        return location;
    }

    void setLocation(Location location) {
        this.location = location;
    }

    @Override
    public String toString() {
        return "CarModel - " + model + " located in " + location;
    }
}
```

`LuxuryCar.class`

```
class LuxuryCar extends Car {
    LuxuryCar(Location location) {
        super(CarType.LUXURY, location);
        construct();
    }
    @Override
    protected void construct() {
        System.out.println("Connecting to luxury car");
    }
}
```

`MicroCar.class`

```
class MicroCar extends Car {
    MicroCar(Location location) {
```

```
        super(CarType.MICRO, location);
        construct();
    }
    @Override
    protected void construct() {
        System.out.println("Connecting to Micro Car ");
    }
}
```

`Minicar.class`

```
class MiniCar extends Car {
    MiniCar(Location location) {
        super(CarType.MINI, location);
        construct();
    }

    @Override
    void construct() {
        System.out.println("Connecting to Mini car");
    }
}
```

`Location.class`

```
enum Location {
    DEFAULT,
    USA,
    INDIA
}
```

`INDIACarFactory.class`

```
class INDIACarFactory {
    static Car buildCar(CarType model) {
        Car car = null;
        switch (model) {
            case MICRO:
                car = new MicroCar(Location.INDIA);
                break;
```

```
            case MINI:
                car = new MiniCar(Location.INDIA);
                break;

            case LUXURY:
                car = new LuxuryCar(Location.INDIA);
                break;

            default:
                break;

        }
        return car;
    }
}
```

`DefaultCarFactory.class`

```
class DefaultCarFactory {
    public static Car buildCar(CarType model) {
        Car car = null;
        switch (model) {
            case MICRO:
                car = new MicroCar(Location.DEFAULT);
                break;

            case MINI:
                car = new MiniCar(Location.DEFAULT);
                break;

            case LUXURY:
                car = new LuxuryCar(Location.DEFAULT);
                break;

            default:
                break;

        }
        return car;
    }
```

```
}
```

`USACarFactory.class`

```
class USACarFactory {
    public static Car buildCar(CarType model) {
        Car car = null;
        switch (model) {
            case MICRO:
                car = new MicroCar(Location.USA);
                break;

            case MINI:
                car = new MiniCar(Location.USA);
                break;

            case LUXURY:
                car = new LuxuryCar(Location.USA);
                break;

            default:
                break;

        }
        return car;
    }
}
```

`CarFactory.class`

```
class CarFactory {
    private CarFactory() {

    }
    public static Car buildCar(CarType type) {
        Car car = null;
        // We can add any GPS Function here which
        // read location property somewhere from configuration
        // and use location specific car factory
        // Currently I'm just using INDIA as Location
        Location location = Location.INDIA;
```

```
        switch (location) {
            case USA:
                car = USACarFactory.buildCar(type);
                break;

            case INDIA:
                car = INDIACarFactory.buildCar(type);
                break;

            default:
                car = DefaultCarFactory.buildCar(type);

        }

        return car;

    }
}
```

`AbstractDesign.class`

```
class AbstractDesign {
    public static void main(String[] args) {
        System.out.println(CarFactory.buildCar(CarType.MICRO));
        System.out.println(CarFactory.buildCar(CarType.MINI));
        System.out.println(CarFactory.buildCar(CarType.LUXURY));
    }
}
```

Output:

```
Connecting to Micro Car
CarModel - MICRO located in INDIA
Connecting to Mini car
CarModel - MINI located in INDIA
Connecting to luxury car
CarModel - LUXURY located in INDIA
```

Java Factory Design Pattern - Mẫu thiết kế Factory Design trong Java

Factory Design Pattern được sử dụng khi chúng ta có một super-class với nhiều sub-class và dựa trên đầu vào, chúng ta cần trả về một trong các sub-class. Design Pattern này nhận trách nhiệm khởi tạo một lớp từ chương trình client đến lớp factory.

1. Tại sao cần Factory Design Pattern

- Factory Pattern được sử dụng khi:

- Chúng ta có một super class với nhiều sub-class và dựa trên đầu vào, chúng ta cần trả về một sub-class. 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 sub-class nào nữa. Khi cần mở rộng, hãy tạo ra sub class và implement thêm vào factory method cho việc khởi tạo sub class này.

- Lợi ích của Factory Pattern:

- 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 chương 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.

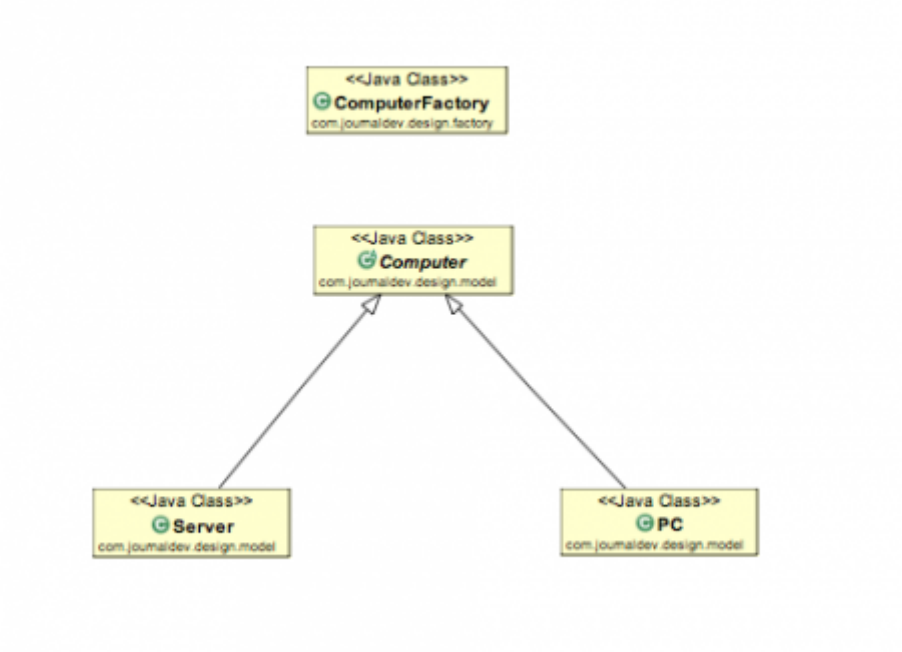
2. Thực thi Factory Design Pattern

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.

Ví dụ: Chương trình sau thể hiện cách chúng ta xây dựng mô hình quan hệ giữa các computer: PC, server



- Factory Design Pattern Super Class

```
public abstract class Computer {

	public abstract String getRAM();
	public abstract String getHDD();
	public abstract String getCPU();

	@Override
	public String toString(){
		return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+", CPU="+this.getCPU();
	}
}
```

- Factory Design Pattern Sub Classes

```
public class PC extends Computer {
```

```
    private String ram;
    private String hdd;
    private String cpu;

    public PC(String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }
    @Override
    public String getRAM() {
        return this.ram;
    }

    @Override
    public String getHDD() {
        return this.hdd;
    }

    @Override
    public String getCPU() {
        return this.cpu;
    }

}
```

```
public class Server extends Computer {

    private String ram;
    private String hdd;
    private String cpu;

    public Server(String ram, String hdd, String cpu){
        this.ram=ram;
        this.hdd=hdd;
        this.cpu=cpu;
    }
    @Override
    public String getRAM() {
        return this.ram;
```

```
    }

    @Override
    public String getHDD() {
        return this.hdd;
    }

    @Override
    public String getCPU() {
        return this.cpu;
    }

}
```

- Factory Class

```
import design.model.Computer;
import design.model.PC;
import design.model.Server;

public class ComputerFactory {

    public static Computer getComputer(String type, String ram, String hdd, String cpu){
        if("PC".equalsIgnoreCase(type)) return new PC(ram, hdd, cpu);
        else if("Server".equalsIgnoreCase(type)) return new Server(ram, hdd, cpu);

        return null;
    }
}
```

- Test Class

```
import design.factory.ComputerFactory;
import design.model.Computer;

public class TestFactory {

    public static void main(String[] args) {
        Computer pc = ComputerFactory.getComputer("pc","2 GB","500 GB","2.4 GHz");
        Computer server = ComputerFactory.getComputer("server","16 GB","1 TB","2.9 GHz");
```

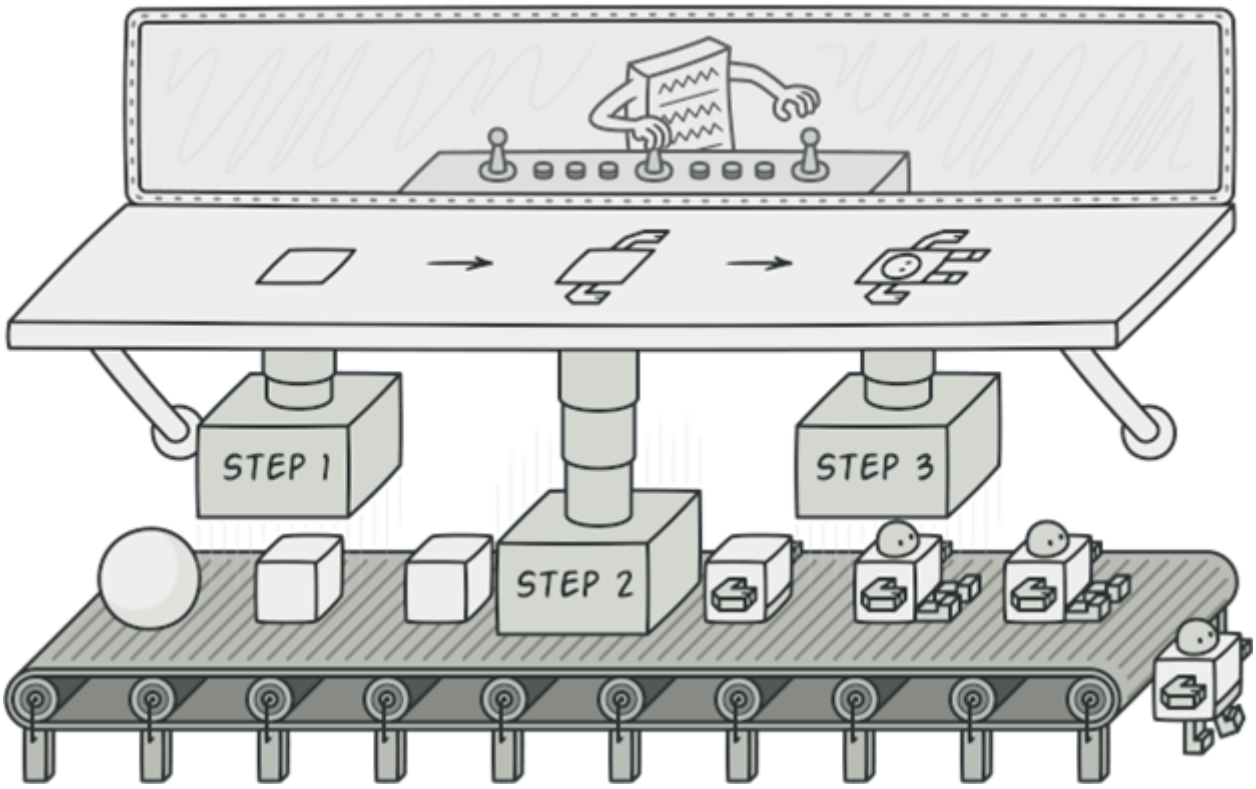
```
    System.out.println("Factory PC Config::"+pc);
    System.out.println("Factory Server Config::"+server);
    }

}
```

- Output:

```
Factory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz
Factory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz
```

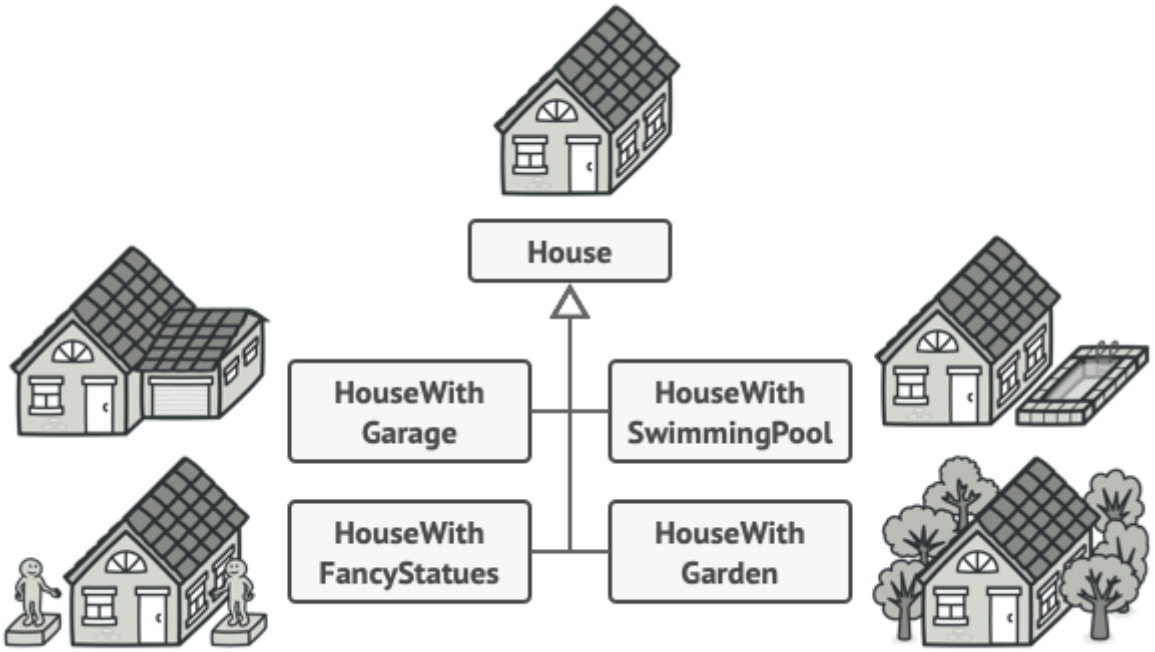
Java Builder Pattern - Mẫu thiết kế Builder trong Java

Builder là một mẫu thiết kế sáng tạo cho phép bạn xây dựng các đối tượng phức tạp theo từng bước. Mẫu cho phép bạn tạo ra các kiểu và biểu diễn khác nhau của một đối tượng bằng cách sử dụng cùng một hàm khởi tạo (construction).



1. Vấn đề

Hãy tưởng tượng một đối tượng phức tạp đòi hỏi nhiều công sức, khởi tạo từng bước của nhiều trường và các đối tượng lồng nhau. Hàm khởi tạo như vậy thường được tạo bên trong một hàm tạo khổng lồ với rất nhiều tham số. Hoặc thậm chí tệ hơn: nằm rải rác trên toàn bộ mã code.

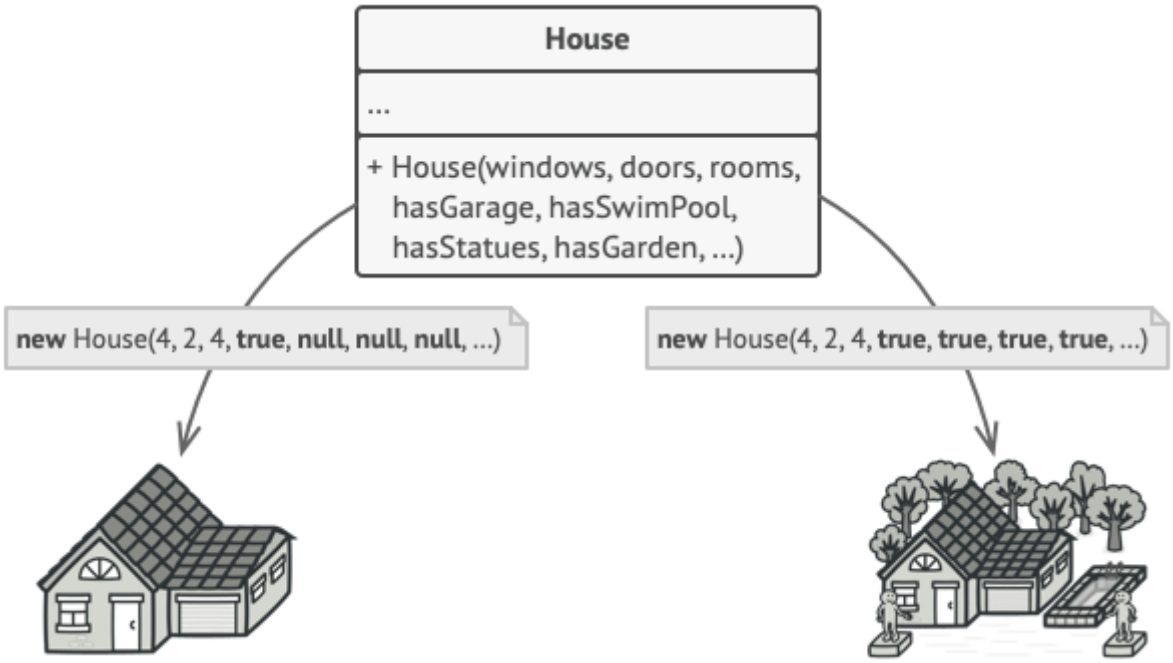


Bạn có thể làm cho chương trình trở nên quá phức tạp bằng cách tạo một lớp con cho mọi cấu hình có thể có của một đối tượng.

Ví dụ: Hãy nghĩ về cách tạo đối tượng Ngôi nhà. Để xây dựng một ngôi nhà đơn giản, bạn cần xây dựng bốn bức tường và một tầng, lắp cửa ra vào, lắp một cặp cửa sổ và xây dựng một mái nhà. Nhưng nếu bạn muốn một ngôi nhà lớn hơn, sáng sủa hơn, có sân sau và các tiện ích khác (như hệ thống sưởi, hệ thống ống nước và hệ thống dây điện)?

Giải pháp đơn giản nhất là mở rộng lớp House cơ sở và tạo một tập hợp các lớp con để bao gồm tất cả các tổ hợp của các tham số. Nhưng cuối cùng bạn sẽ có một số lượng đáng kể các lớp con. Bất kỳ thông số mới nào, chẳng hạn như kiểu hiên nhà, sẽ yêu cầu phát triển hệ thống phân cấp này nhiều hơn nữa.

Có một cách tiếp cận khác không liên quan đến việc lai tạo các lớp con. Bạn có thể tạo một phương thức khởi tạo khổng lồ ngay trong lớp House cơ sở với tất cả các tham số có thể có để điều khiển đối tượng house. Mặc dù cách tiếp cận này thực sự loại bỏ sự cần thiết của các lớp con, nhưng nó lại tạo ra một vấn đề khác.

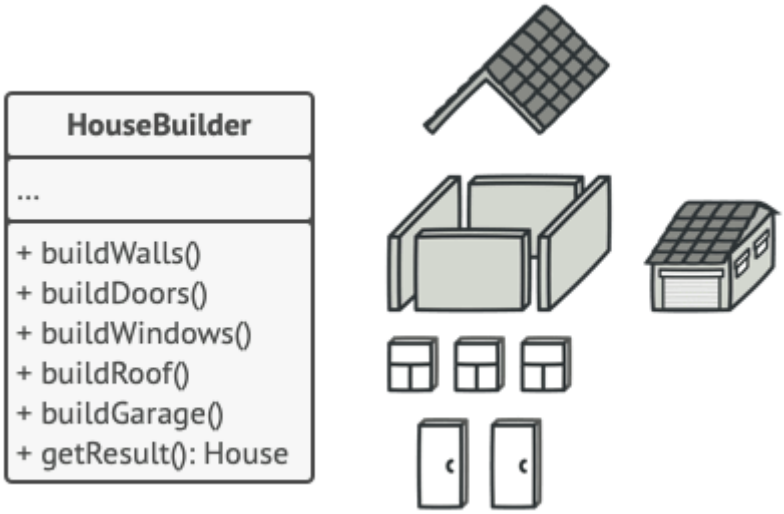


Hàm tạo với nhiều tham số có nhược điểm của nó: không phải lúc nào cũng cần có tất cả các tham số.

Trong hầu hết các trường hợp, hầu hết các tham số sẽ không được sử dụng, làm cho các cuộc gọi hàm tạo khá xấu xí. Ví dụ: Chỉ một phần nhỏ các ngôi nhà có bể bơi, vì vậy các thông số liên quan đến bể bơi sẽ vô dụng.

2. Giải pháp

Mẫu Builder gợi ý rằng bạn trích xuất mã xây dựng đối tượng ra khỏi lớp của chính nó và di chuyển nó đến các đối tượng riêng biệt được gọi là *trình xây dựng*.

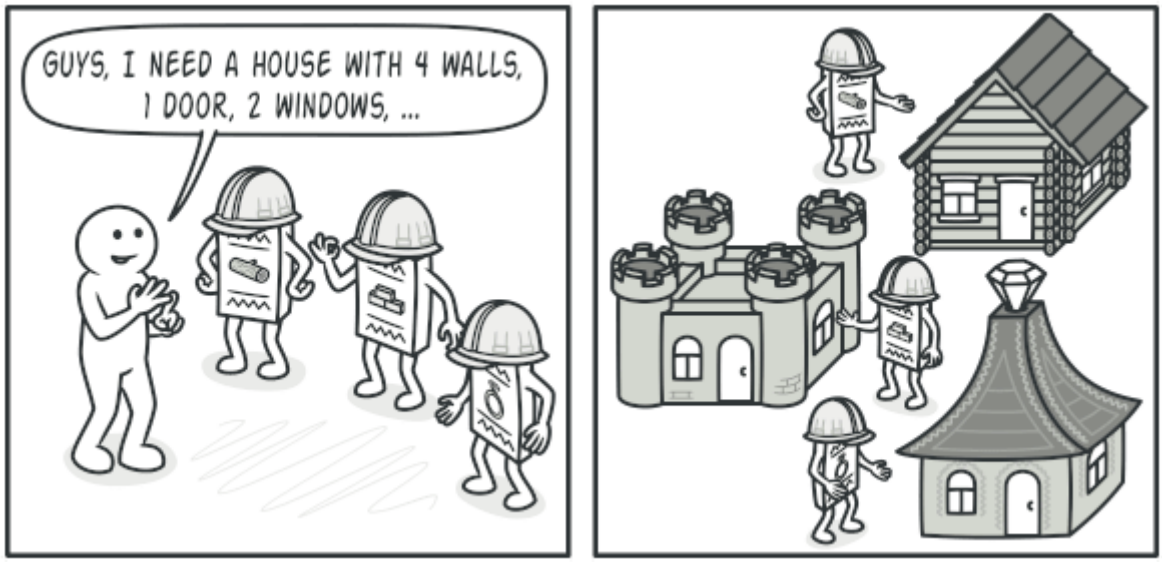


Mẫu Builder cho phép bạn xây dựng các đối tượng phức tạp theo từng bước. Builder không cho phép các đối tượng khác truy cập vào sản phẩm khi nó đang được xây dựng.

Mẫu sắp xếp việc xây dựng đối tượng thành một tập hợp các bước (buildWalls, buildDoor, v.v.). Để tạo một đối tượng, bạn thực hiện một loạt các bước này trên một đối tượng trình tạo. Phần quan trọng là bạn không cần phải gọi tất cả các bước. Bạn chỉ cần gọi những bước cần thiết để tạo ra một cấu hình cụ thể của một đối tượng.

Một số bước xây dựng có thể yêu cầu thực hiện khác nhau khi bạn cần xây dựng các hình ảnh đại diện khác nhau của sản phẩm. Ví dụ: Các bức tường của một cabin có thể được xây dựng bằng gỗ, nhưng các bức tường của lâu đài phải được xây dựng bằng đá.

Trong trường hợp này, bạn có thể tạo một số lớp trình xây dựng khác nhau triển khai cùng một tập hợp các bước xây dựng, nhưng theo một cách khác. Sau đó, bạn có thể sử dụng các trình xây dựng này trong quá trình xây dựng (tức là một tập hợp các lệnh gọi có thứ tự đến các bước xây dựng) để tạo ra các loại đối tượng khác nhau.

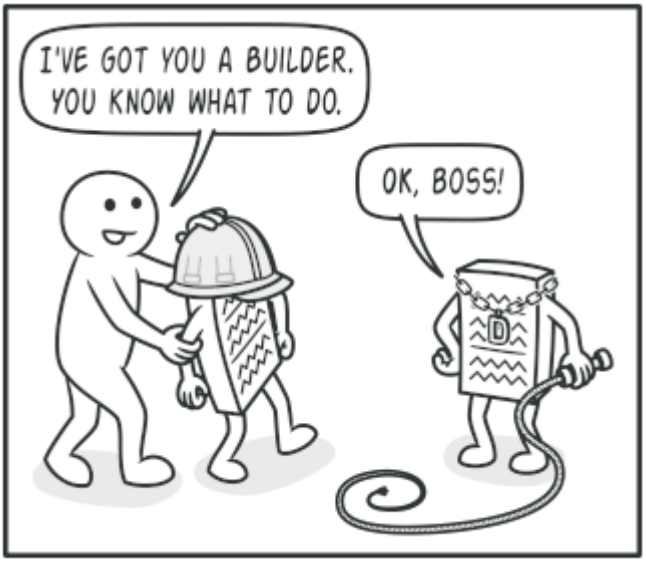


Các *trình xây dựng* khác nhau thực hiện cùng một nhiệm vụ theo nhiều cách khác nhau.

Ví dụ: Hãy tưởng tượng một người thợ xây dựng mọi thứ từ gỗ và kính, người thứ hai xây dựng mọi thứ bằng đá và sắt và người thứ ba sử dụng vàng và kim cương. Bằng cách gọi cùng một nhóm các bước, bạn sẽ có được một ngôi nhà bình thường từ người xây dựng đầu tiên, một lâu đài nhỏ từ người thứ hai và một cung điện từ người thứ ba. Tuy nhiên, điều này sẽ chỉ hoạt động nếu client code gọi các bước xây dựng có thể tương tác với các nhà xây dựng bằng giao diện chung.

3. Class Director

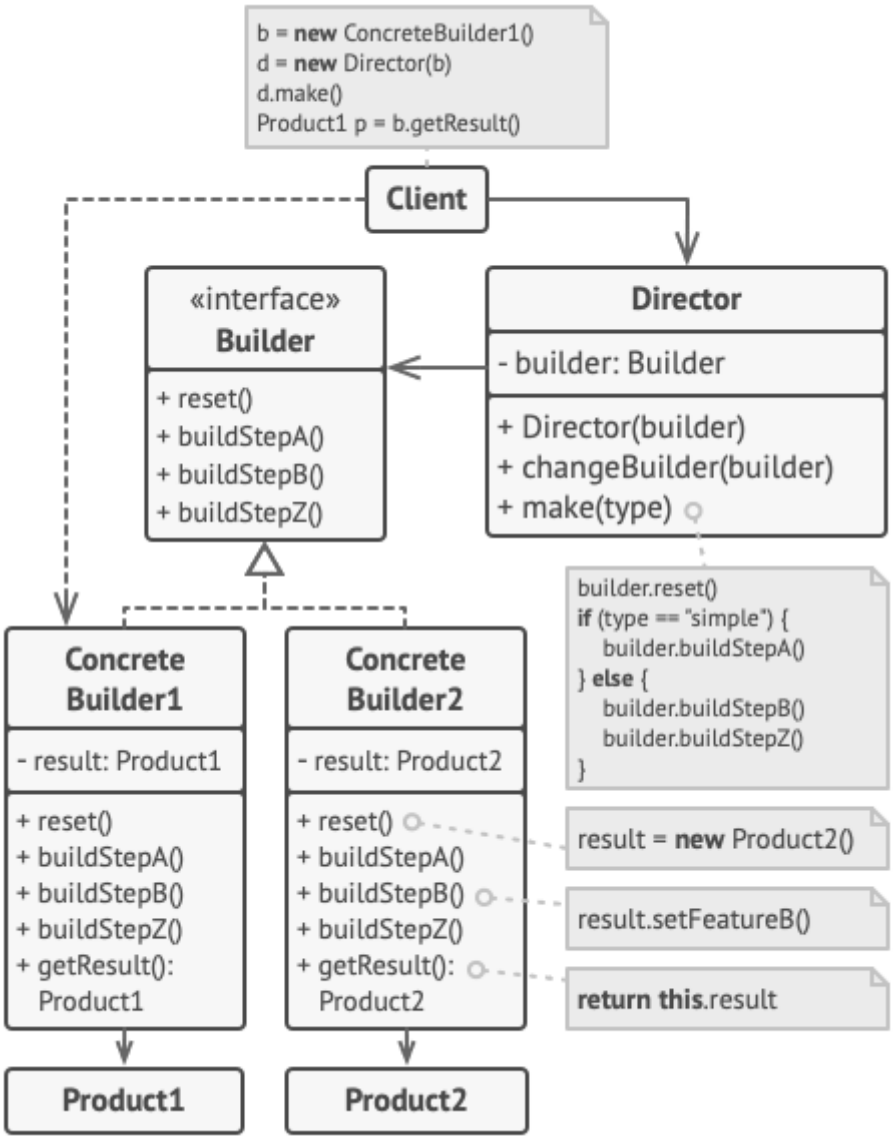
Bạn có thể đi xa hơn và trích xuất một loạt các lệnh gọi đến các bước của trình tạo mà bạn sử dụng để xây dựng một sản phẩm thành một lớp riêng biệt có tên là *director*. Lớp *director* xác định thứ tự thực hiện các bước xây dựng, trong khi trình xây dựng cung cấp việc triển khai cho các bước đó.



Director biết các bước xây dựng cần thực hiện để có được một sản phẩm hoạt động. Việc có một lớp *director* trong chương trình của bạn là không hoàn toàn cần thiết. Bạn luôn có thể gọi các bước xây dựng theo thứ tự cụ thể trực tiếp từ client code. Tuy nhiên, lớp *director* có thể là một nơi tốt để đưa các quy trình xây dựng khác nhau để bạn có thể sử dụng lại chúng trong chương trình của mình.

Ngoài ra, *director* hoàn toàn giấu kín các chi tiết cấu tạo sản phẩm với client code. Client chỉ cần liên kết *builder* với *director*, khởi tạo với *director* và nhận kết quả từ *builder*.

4. Structure



1. Giao diện `Builder` khai báo các bước xây dựng sản phẩm chung cho tất cả các loại trình xây dựng.

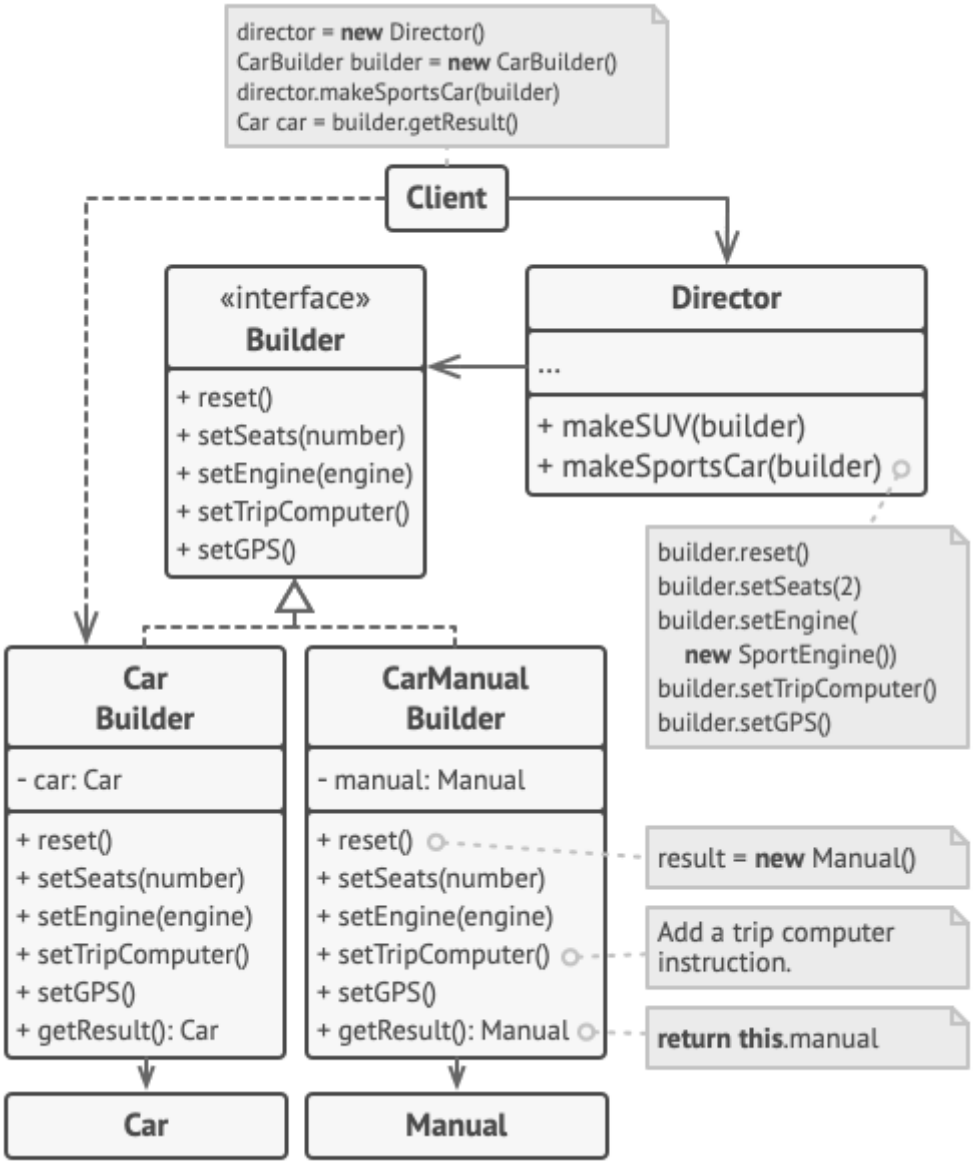
2. `Concrete Builders` cung cấp các cách triển khai khác nhau của các bước xây dựng. Concrete Builders có thể tạo ra các sản phẩm không tuân theo giao diện chung.

3. `Product` là đối tượng kết quả. Các product do các builder khác nhau tạo ra không nhất thiết phải thuộc cùng một hệ thống phân cấp hoặc giao diện lớp.

4. Lớp `Director` xác định thứ tự gọi các bước xây dựng, vì vậy bạn có thể tạo và sử dụng lại các cấu hình cụ thể của sản phẩm.

5. Client phải liên kết một trong các đối tượng builder với director. Thông thường, nó chỉ được thực hiện một lần, thông qua các tham số của hàm tạo của director. Sau đó, director sử dụng đối tượng xây dựng đó cho tất cả các construction xây dựng tiếp theo. Tuy nhiên, có một cách tiếp cận thay thế khi client chuyển đối tượng builder sang method của director. Trong trường hợp này, bạn có thể sử dụng một trình builder khác mỗi khi bạn thực thi với director.

5. Mã giả




```
// Using the Builder pattern makes sense only when your products
// are quite complex and require extensive configuration.  The
// following two products are related, although they don't have
// a common interface.
class Car is
    // A car can have a GPS, trip computer and some number of
    // seats.  Different models of cars (sports car, SUV,
    // cabriolet) might have different features installed or
    // enabled.

class Manual is
    // Each car should have a user manual that corresponds to
    // the car's configuration and describes all its features.
```

```
// The builder interface specifies methods for creating the
// different parts of the product objects.
interface Builder is
    method reset()
    method setSeats(...)
    method setEngine(...)
    method setTripComputer(...)
    method setGPS(...)

// The concrete builder classes follow the builder interface and
// provide specific implementations of the building steps.  Your
// program may have several variations of builders, each
// implemented differently.
class CarBuilder implements Builder is
    private field car: Car

    // A fresh builder instance should contain a blank product
    // object which it uses in further assembly.
    constructor CarBuilder() is
        this.reset()

    // The reset method clears the object being built.
    method reset() is
        this.car = new Car()

    // All production steps work with the same product instance.
    method setSeats(...) is
        // Set the number of seats in the car.

    method setEngine(...) is
        // Install a given engine.

    method setTripComputer(...) is
        // Install a trip computer.

    method setGPS(...) is
        // Install a global positioning system.
```

```
    // Concrete builders are supposed to provide their own
    // methods for retrieving results. That's because various
    // types of builders may create entirely different products
    // that don't all follow the same interface. Therefore such
    // methods can't be declared in the builder interface (at
    // least not in a statically-typed programming language).
    //
    // Usually, after returning the end result to the client, a
    // builder instance is expected to be ready to start
    // producing another product. That's why it's a usual
    // practice to call the reset method at the end of the
    // `getProduct` method body. However, this behavior isn't
    // mandatory, and you can make your builder wait for an
    // explicit reset call from the client code before disposing
    // of the previous result.
    method getProduct(): Car is
        product = this.car
        this.reset()
        return product

// Unlike other creational patterns, builder lets you construct
// products that don't follow the common interface.
class CarManualBuilder implements Builder is
    private field manual: Manual

    constructor CarManualBuilder() is
        this.reset()

    method reset() is
        this.manual = new Manual()

    method setSeats(...) is
        // Document car seat features.

    method setEngine(...) is
        // Add engine instructions.

    method setTripComputer(...) is
        // Add trip computer instructions.
```

```
    method setGPS(...) is
        // Add GPS instructions.

    method getProduct(): Manual is
        // Return the manual and reset the builder.


// The director is only responsible for executing the building
// steps in a particular sequence. It's helpful when producing
// products according to a specific order or configuration.
// Strictly speaking, the director class is optional, since the
// client can control builders directly.
class Director is
    private field builder: Builder

    // The director works with any builder instance that the
    // client code passes to it. This way, the client code may
    // alter the final type of the newly assembled product.
    method setBuilder(builder: Builder)
        this.builder = builder

    // The director can construct several product variations
    // using the same building steps.
    method constructSportsCar(builder: Builder) is
        builder.reset()
        builder.setSeats(2)
        builder.setEngine(new SportEngine())
        builder.setTripComputer(true)
        builder.setGPS(true)

    method constructSUV(builder: Builder) is
        // ...


// The client code creates a builder object, passes it to the
// director and then initiates the construction process. The end
// result is retrieved from the builder object.
class Application is

    method makeCar() is
```

```
        director = new Director()

        CarBuilder builder = new CarBuilder()
        director.constructSportsCar(builder)
        Car car = builder.getProduct()

        CarManualBuilder builder = new CarManualBuilder()
        director.constructSportsCar(builder)

        // The final product is often retrieved from a builder
        // object since the director isn't aware of and not
        // dependent on concrete builders and products.
        Manual manual = builder.getProduct()
```

6. Mối quan hệ với các Pattern khác

- Nhiều thiết kế bắt đầu bằng cách sử dụng Factory Method (ít phức tạp hơn và có thể tùy chỉnh nhiều hơn thông qua các lớp con) và phát triển theo hướng Abstract Factory, Prototype hoặc Builder (linh hoạt hơn nhưng phức tạp hơn).
- Builder tập trung vào việc xây dựng các đối tượng phức tạp theo từng bước. Abstract Factory chuyên tạo các family đối tượng liên quan. Abstract Factory trả lại sản phẩm ngay lập tức, trong khi Builder cho phép bạn chạy một số bước xây dựng bổ sung trước khi tìm nạp sản phẩm.
- Bạn có thể sử dụng Builder khi tạo các cây Composite phức tạp vì bạn có thể lập trình các bước xây dựng của nó để hoạt động một cách đệ quy.
- Bạn có thể kết hợp Builder với Bridge: lớp director đóng vai trò trừu tượng, trong khi các trình xây dựng khác nhau đóng vai trò triển khai.
- Abstract Factories, Builders và Prototypes đều có thể được thực hiện dưới dạng các Singleton.