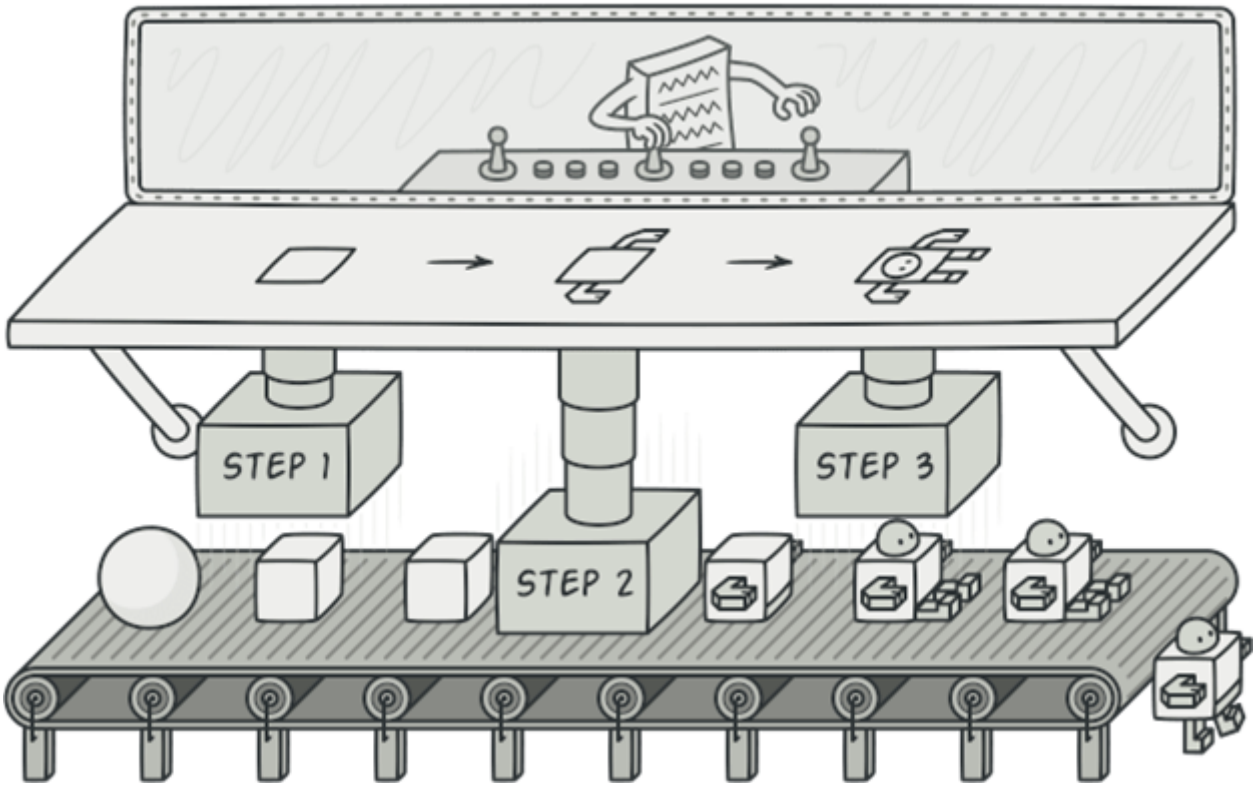
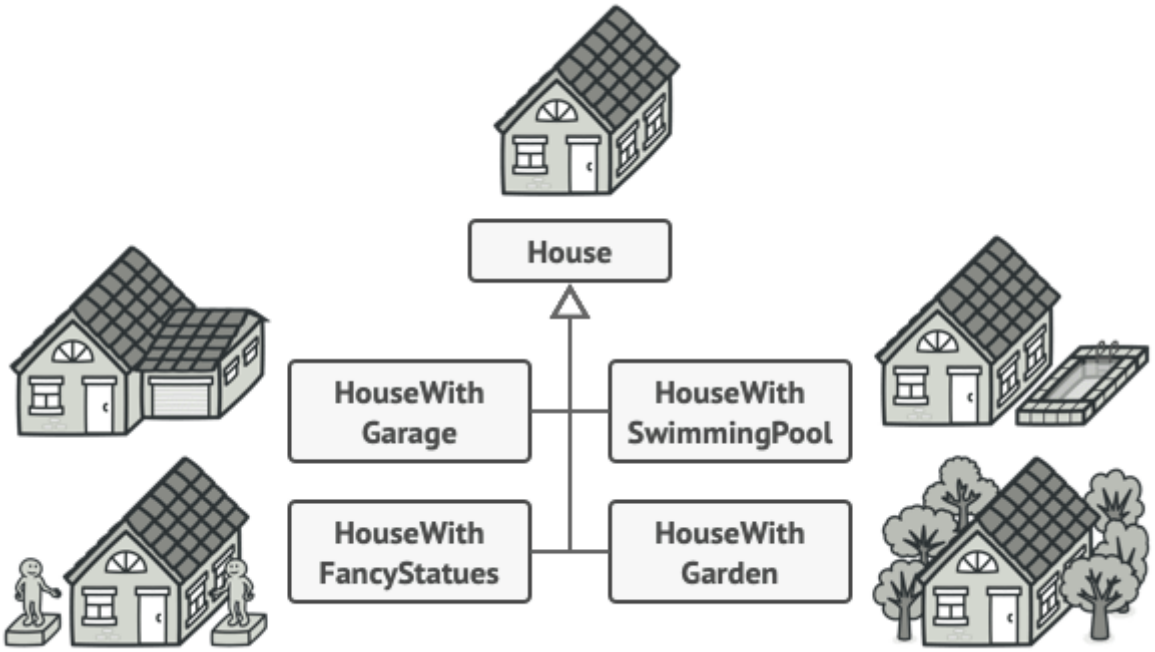
Java Builder Pattern - Mẫu thiết kế Builder trong Java

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.

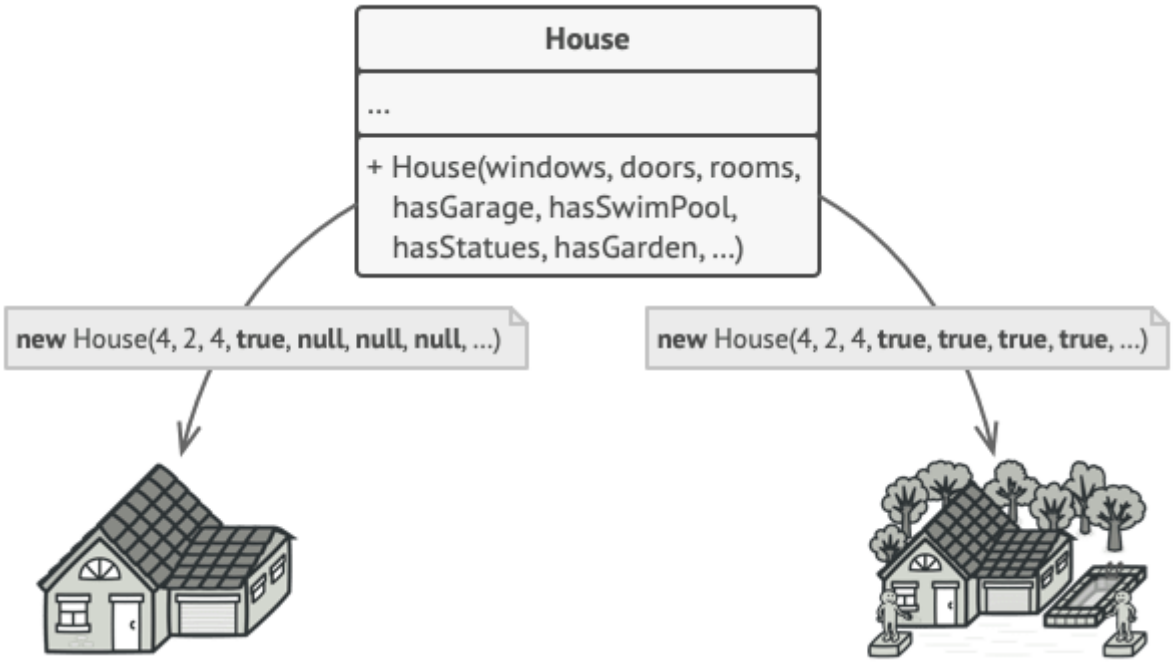


Bạn có thể làm cho chương trình trở nên quá phức tạp bằng cách tạo một lớp con cho mọi cấu hình có thể có của một đối tượng.

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.

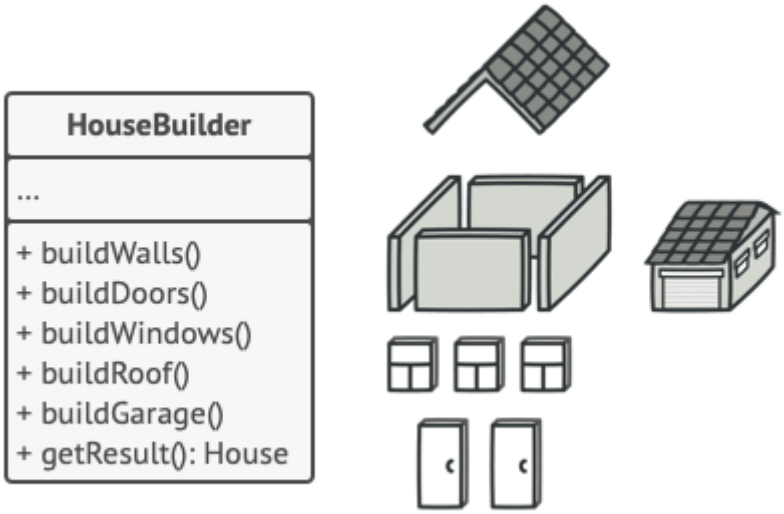


Hàm tạo với nhiều tham số có nhược điểm của nó: không phải lúc nào cũng cần có tất cả các tham số.

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*.

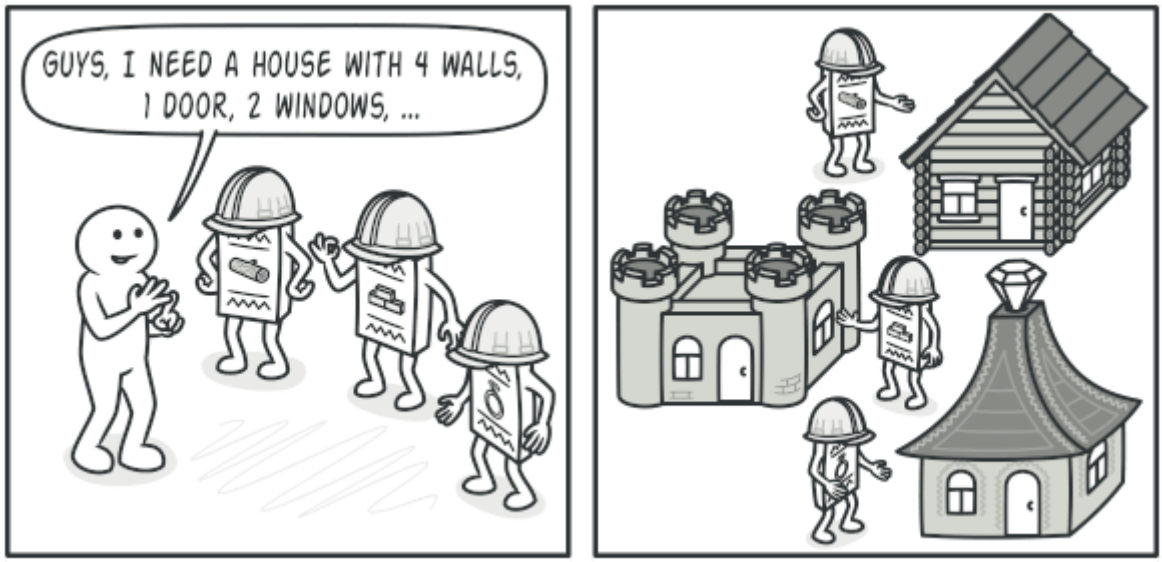


Mẫu Builder cho phép bạn xây dựng các đối tượng phức tạp theo từng bước. Builder không cho phép các đối tượng khác truy cập vào sản phẩm khi nó đang được xây dựng.

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.

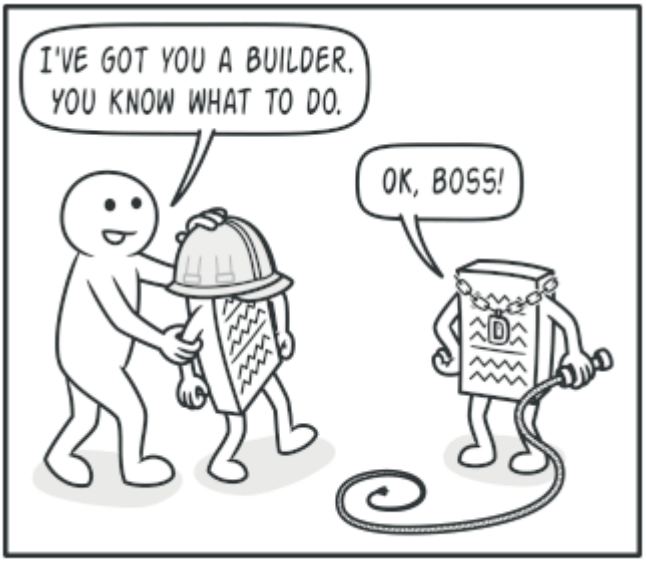


Các *trình xây dựng* khác nhau thực hiện cùng một nhiệm vụ theo nhiều cách khác nhau.

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

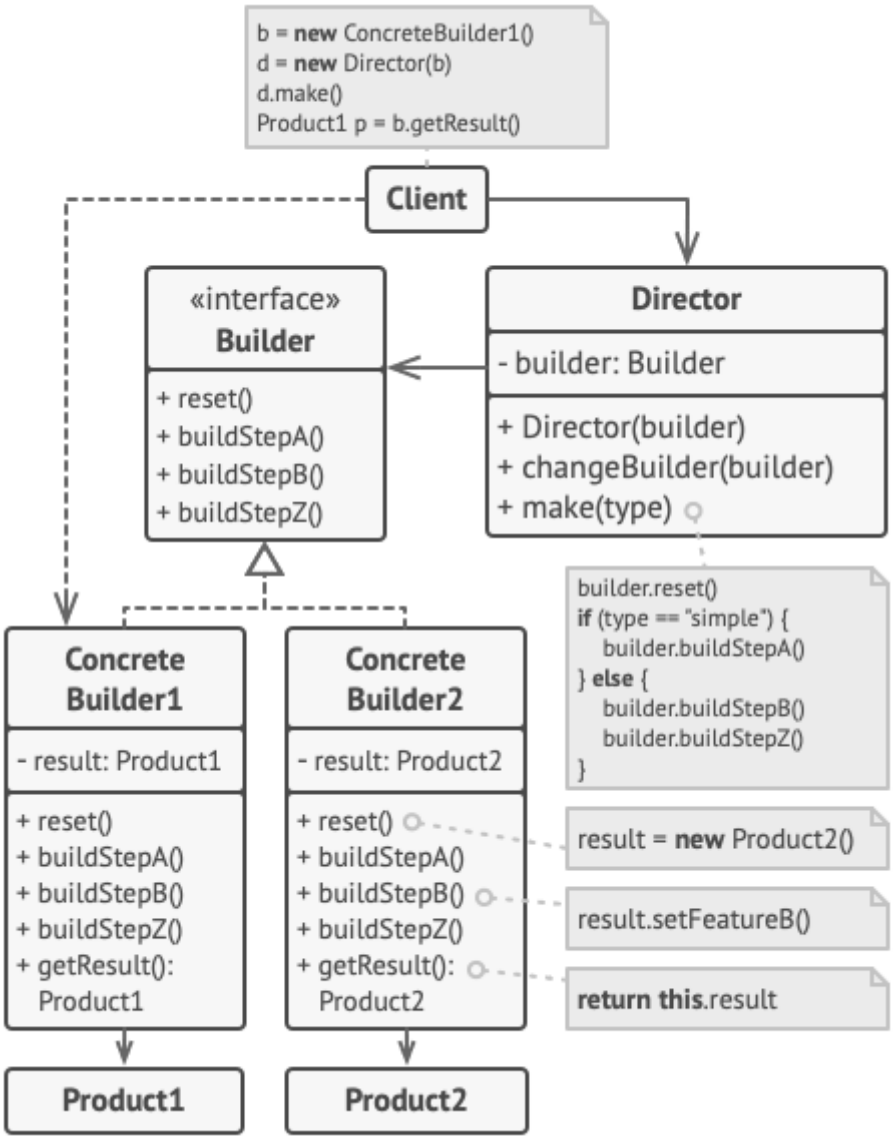
Bạn có thể đi xa hơn và trích xuất một loạt các lệnh gọi đến các bước của trình tạo mà bạn sử dụng để xây dựng một sản phẩm thành một lớp riêng biệt có tên là *director*. Lớp *director* xác định thứ tự thực hiện các bước xây dựng, trong khi trình xây dựng cung cấp việc triển khai cho các bước đó.



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.

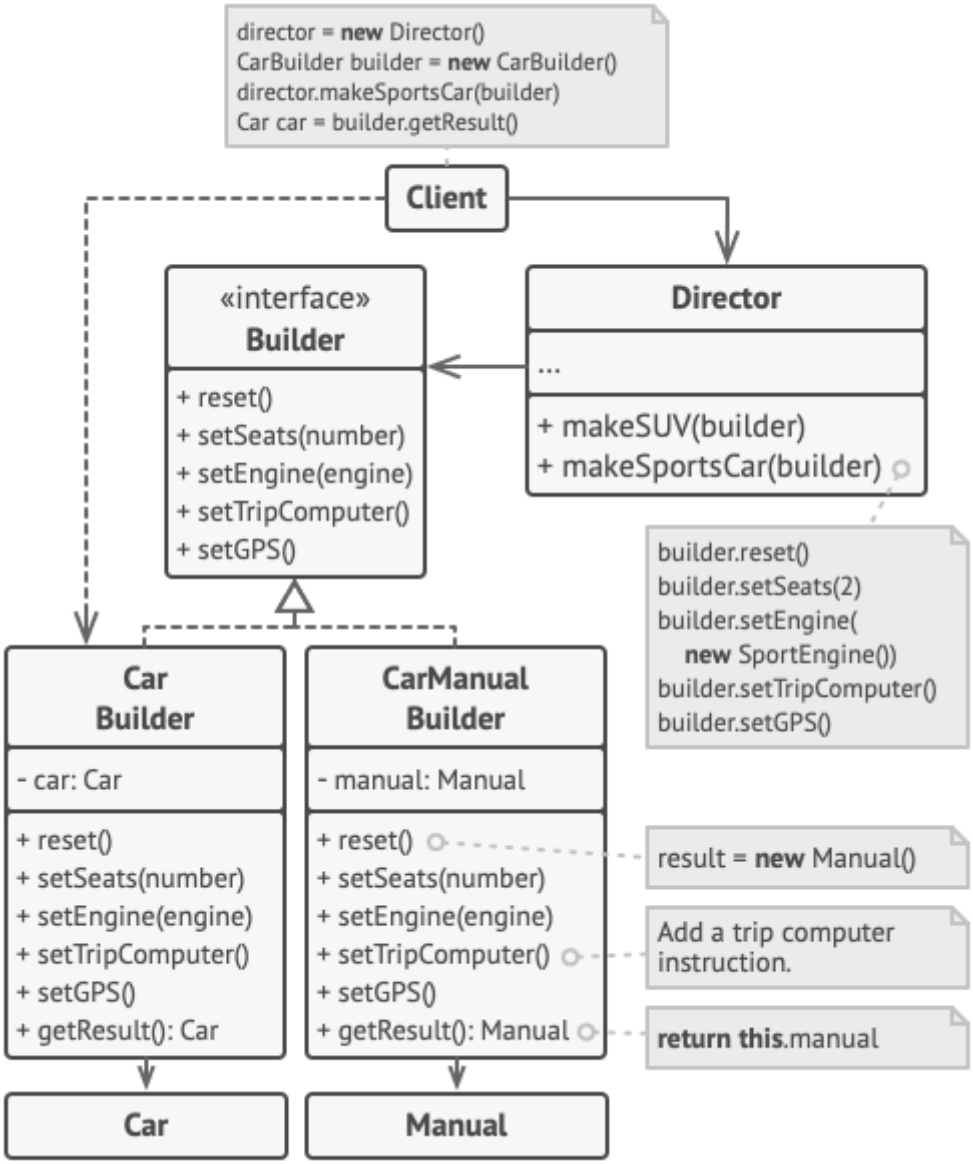
2. `Concrete Builders` cung cấp các cách triển khai khác nhau của các bước xây dựng. Concrete Builders có thể tạo ra các sản phẩm không tuân theo giao diện chung.

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.

Revision #1
Created 14 August 2021 07:43:54 by Laptrinh.vn
Updated 14 August 2021 08:07:02 by Laptrinh.vn