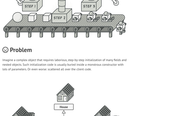
Builder

Intent

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.



Problem

Imagine a complex object that requires laborious step-by-step initialization of many fields and nested objects. Such initialization code is usually buried inside a monstrous constructor with lots of parameters. Or even worse: scattered all over the client code.

Many flavors of houses require lots of different constructions.

The constructor with lots of parameters has its own drawbacks since it's not pretty even in the case when all of the parameters are used, meaning **the constructor calls pretty ugly**. For instance, only a fraction of houses have swimming pools, so the parameters related to swimming pools will be useless nine times out of ten.

Solution

The Builder pattern suggests that you extract the object construction code out of its own class and move it to separate objects called builders.

The Builder pattern lets you construct objects step by step. The Builder doesn't allow other objects access to the product while it is being built.

The pattern organizes object construction into a set of steps (`reset()`, `setWalls()`, etc.). To create an object, you execute a series of these steps on a builder object. The important part is that not all of the steps need to be called. You can call only those steps that are necessary for producing a particular configuration of an object.

Some of the construction steps might require different implementation when you need to build various representations of the product. For example, walls of a cottage may be built of wood and the castle walls must be built with stone.

In this case, you can create several different builder classes that implement the same set of building steps, but in a different manner. Then you can use these builders in the construction process (i.e., an ordered set of calls to the building steps) to produce different kinds of objects.

Different builders execute the same task in different ways.

For example, imagine a builder that builds everything from wood and glass, a second one that builds everything with stone and iron and a third one that uses gold and diamonds. By calling steps from the base interface in order, you ask the builder to start the process of producing a house, a castle or a house with diamonds. Then, the wood-using guy in charge of the cottage builder knows that he should place windows and doors, the stone-throwing dude in charge of the castle builder knows that he should also build the castle walls and the king of diamonds knows that he should place a heliport on the roof of his house.

Director

You can go further and extract a series of calls to the builder steps you use to construct a product into a separate class called director. The director class defines the order in which to execute the building steps, while the builder provides the implementations for these steps.

The director knows which building steps to execute to get a working product.

Having a director class in your program isn't strictly necessary. You can always call the building steps in a specific order directly from the client code. However, the director class might be a good place to put various construction routines so you can reuse them across your program.

In addition, the director class completely hides the details of product construction from the client code. The client only needs to associate a builder with a director, launch the construction with the director, and get the result from the builder.

Structure

The **Builder** interface declares product construction steps common to all kinds of builders.

Concrete Builders provide different implementations of the construction steps. Concrete builders may produce products that don't follow the common interface.

Products are resulting objects. Products constructed by different builders don't belong to a common class hierarchy or interface.

The **Director** class defines the order in which to call construction steps, so you can create and reuse specific configurations of products.

The **Client** must associate one of the builders with the director. Usually it's enough to do this once via parameters of the director's constructor. Then the director uses that builder object for all further building needs. The client gets the building result from the builder when the director finishes assembling a product.

Pseudocode

This example of the **Builder** pattern illustrates how you can reuse the same object construction code when building different types of products, such as cars and create the corresponding manuals for them.

The removal of step by step construction of cars and the car guide that fit those car models

A car is a complex object that can be constructed in a hundred different ways. Instead of polluting the car class with a huge constructor, we've put car assembly algorithms into a separate car builder class. This class has a set of methods for configuring various parts of a car.

If the client code needs to assemble a special, fine-tuned model of a car, it can send with the builder directly (in the other hand, the client can delegate the assembly to the director object which knows how to use a builder to construct several of the most popular models of cars).

You might be shocked but every car needs a manual (at least, the boring kind). The manual describes every feature of the car, so the details in the manuals vary across the different models of the same car and different configurations of the same model. Since all manuals are plain and their respective manuals. Of course building a manual isn't the same as building a car and that's why we have provide another builder class that specializes in composing manuals. This builder implements the same building steps as the car builder, but instead of assembling cars it can parts it describes them. By passing these builders to the same director object, we can construct either a car or a manual.

The final part is fetching the resulting object. A metal car and a paper manual, although models are still very different. Brings the car's place a method for fetching results is the responsibility of the director class. The director knows about details of the product, thus, we attach the result of the construction from the builder which performed the job.

```
// Using the Builder pattern makes sense only when your products
// are quite complex and require extensive configuration.
// Following two products are related, although they don't have
// common interface.
class Car {
    // A car can have a GPS, trip computer and some number of
    // seats. Different models of cars (sports car, SUV
    // limousine) may have different features implemented in
    // another
}

class Manual {
    // 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 {
    methods setCarType(type)
    method setSeats(seats)
    method setEngine()
    method setTransmission()
    method setTripComputer()
    method setGPS()

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

    // A fresh builder instance contains a blank product
    // object, which is used in further assembly.
    constructor CarBuilder() {
        this.reset()
    }

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

    // All construction steps work with the same product instance.
    method setCarType(type: CarType)
        // Set the number of seats in the car.
    method setSeats(seats: int)
    method setEngine(engine: Engine)
    method setTransmission(t: Transmission)
        // Install a trip computer.
    method setTripComputer(c: TripComputer)
    method setGPS(g: GPS)
        // Install a global positioning system.

    // Concrete builders are expected to provide their own
    // method for fetching results. That's because different kinds
    // of builders may create quite varying different products
    // that don't share common interface. In our case, only
    // methods can't be declared in the builder interface so
    // that each of a particular kind of builder. Usually,
    // the client knows which concrete builder it
    // is working with, so it can get the result from it
    // without any doubts.
    // A typical client gets the result from the builder
    // instance that it controls, for example, the last car
    // assembled without any doubts. This definition can't
    // be in the builder interface, because not all builders
    // produce cars and for the cases when builders assembling
    // other products, this method may differ.
    method getProduct(): Car {
        // Save the current product to local
        // variable.
        var product = this.car
        // Indicate that this builder is ready to
        // assemble another car.
        this.reset()
        return product
    }

// Unlike other builders, CarManualBuilder lets you construct
// manuals for different cars models.
class CarManualBuilder implements Builder {
    private Manual manual;

    constructor CarManualBuilder() {
        this.reset()
    }

    method reset() {
        this.manual = new Manual()
    }

    method setCarType(type: CarType)
        // Select a manual template based on car type.
    method setSeats(seats: int)
        // Add seats section.
    method setEngine(engine: Engine)
        // Add engine instructions.
    method setTransmission(t: Transmission)
        // Add trip computer instructions.
    method setTripComputer(c: TripComputer)
    method setGPS(g: GPS)
        // Add Global Positioning System chapter.
    method getProduct(): Manual {
        // Return the manual and reset the builder.

// The director is only responsible for executing the building
// steps in a particular sequence. It is useful when there are
// products according to a specific order of configuration.
// Strictly speaking, the director class is optional, since the
// client can control builders directly.
class Director {
    // 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.
    // The director can construct several product variations
    // using the same building steps.
    protected Builder builder;

    // The client passes a builder object to the
    // director.
    constructor setBuilder(b: Builder) {
        this.builder = b;
    }

    // The client asks the director to construct a car.
    // Maybe there are other ways to assemble a car. For
    // example, we could also define several such methods
    // in the director class.
    // A car with a basic set of features: a single
    // seat and an internal combustion engine.
    method buildMinimalViewer() {
        this.builder.setCarType(CarType.CITY_CAR)
        this.builder.setSeats(1)
        this.builder.setEngine(new Engine(0.75, false))
        this.builder.setTransmission(Transmission.SINGLE_SPEED)
    }

    // A car with all the bells and whistles.
    method buildFullFeaturedViewer() {
        this.builder.setCarType(CarType.SUV)
        this.builder.setSeats(5)
        this.builder.setEngine(new Engine(2.5, true))
        this.builder.setTransmission(Transmission.MANUAL)
        this.builder.setTripComputer(true)
        this.builder.setGPS(true)
    }

// The client code creates a builder object, passes it to the
// director and then gets the product from the director. The
// director is optional since the client knows
// which product needs to be built.
client setup() {
    director = new Director()
    CarBuilder builder = new CarBuilder()
    director.setBuilder(builder)

    // Produce a sports car.
    director.buildMinimalViewer()
    var car = builder.getProduct()

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

    // Two front panels in one kitchen and one in the
    // bathroom.
    // Produce a user manual for a car.
    director.buildFullFeaturedViewer()
    var manual = builder.getProduct()
```

Applicability

Use the Builder pattern to get rid of a "telescoping constructor"

You've here a constructor with ten optional parameters. Calling such a beast is very inconvenient; therefore, you overload the constructor and create several shorter versions with fewer parameters. Such constructors still refer to the main one passing some default values into unlike parameters.

```
class Pizza {
    Pizza(int size) {...}
    Pizza(int size, boolean cheese) {...}
    Pizza(int size, boolean cheese, boolean pepperoni) {...}
    // ...
}
```

Creating such a monster only is possible in languages that support method overloading, such as C# or Java.

The Builder pattern lets you build objects step by step, using only those steps that you really need. After implementing the pattern, you don't have to cram all possible parameters into your construction anymore.

Use the Builder pattern when you want your code to be able to create different representations of some product (for example, stone and wooden houses).

The Builder pattern can be applied when construction of various representations of the product involves similar steps that differ only in the details.

The base builder interface defines all possible construction steps and concrete builders implement some of these steps. You can then call such steps without caring about which kind of product is being built.

Use the Builder to construct Composite trees or other complex objects.

The Builder pattern lets you construct products step by step. You could refer execution of some steps without breaking the final product. Your client can start assembling which comes in handy when you need to build an object tree.

A builder doesn't expose the unfinished product while running construction steps. This prevents the client code from fetching an incomplete result.

How to Implement

1. Make sure that you can clearly define the common construction steps for building all available product representations. Otherwise, you won't be able to proceed with implementing the pattern.
2. Declare these steps in the base builder interface.
3. Create a concrete builder class for each of the representations and implement their construction steps.

   Don't forget about implementing a method for fetching the result of the construction. The reason why this method can't be declared inside the builder interface is that various builders may produce products of different types that don't have a common interface. Thus, inside what would be the return type for such a method. Moreover, if you're dealing with products from a single hierarchy, the fetching method can be safely added to the base interface.
4. Think about creating a director class. It may encapsulate various ways to construct a product using the same builder object.
5. The client code creates both the builder and the director objects. Before construction starts, your code should place a builder object to the director. Then, depending on the rules of your business, or preferences of the director class. The client code can fetch the director or the finished product from the director, if the director doesn't care about the product at all.
6. The construction results can be obtained directly from the director only if all products follow the same interface. Otherwise the client should fetch the result from the builder.

Pros and Cons

- You can construct objects step-by-step, defer construction steps or run steps recursively.
- You can reuse the same construction code when building various representations of products.
- Single Responsibility Principle. You can isolate complex construction code from the business logic of the product.
- The overall complexity of the code increases since the pattern requires creating multiple new classes.

Relations with Other Patterns

- Many designs start by using **Factory Method** (less complicated and more customizable via subclasses) and evolve toward **Abstract Factory**, **Prototype**, or **Builder** (more flexible, but more complicated).
- **Builder** focuses on constructing complex objects step by step. **Abstract Factory** specializes in creating families of related products. Builder returns the product as a final step, but in the context of Abstract Factory, the product gets returned immediately after calling the factory method.
- You can use **Builder** when creating complex **Composite** trees because you can program its construction steps to work recursively.
- You can combine **Builder** with **Bridge**: the director class plays the role of the abstraction, while different builders act as implementations.
- **Abstract Factories**, **Builders** and **Prototypes** can all be implemented as **Singletons**.

Code Examples

Why buy the book Design Patterns eBook?
- Learn patterns 10 times faster with examples in your favorite programming language
- Stay motivated with the easy-to-digest style of the book, written by a friendly guide
- Ready-to-use UML diagrams and code examples for all patterns
- Lifetime access to all future updates and additions
Get eBook
Refactoring Guru
Design Patterns
Premium Content
Forum
Contact
Terms & Conditions
Privacy Policy
Content Usage Policy
About us
© 2014-2024 Refactoring Guru. All rights reserved.
Illustrations by Dmitry Zhart
Unionized offices:
Refactoring Guru s.r.o.
Jankovcova 1518/2
Praha, Czechia, 170000
Registered in the Commercial Register
Operated offices:
Refactoring Guru
Dmitriev Alexander
Nizhny Novgorod, 603000
Registered in the Unified State Register of Legal Entities