This blog has been moved to http://jlordiales.me

This blog has been moved to http://jlordiales.me

The builder pattern in practice

Posted on **December 13, 2012**

IMPORTANT: This blog has been moved to jlordiales.me. The content here might be outdated. To see this post on the new blog go to http://jlordiales.me/2012/12/13/the-builder-pattern-in-practice

So, this is my first post (and my first blog for that matter). I can't remember exactly where I read this (although I'm almost sure it was on Practices of an Agile Developer), but writing in a blog is supposed to help you get your thoughts together. Concretely, by taking the time to explain what you know, you get a better understanding of it yourself.

And that's exactly what I'm going to try to do here, explain things to get a better understanding of them. And, as a bonus, it will also serve me as centralized place to go to when I want to revisit something I've done in the past. Hopefully, this will help some of you in the process.

With the introduction out of the way, lets jump straight into this first post which, as the title so eloquently says🙂, is about the builder pattern. I'm not going to dive into much details about the pattern because there's already tons of posts and books that explain it in fine detail. Instead, I'm going to tell you why and when you should consider using it. However, it is worth mentioning that this pattern is a bit different to the one presented in the Gang of Four book. While the original pattern focuses on abstracting the steps of construction so that by varying the builder implementation used we can get a different result, the pattern explained in this post deals with removing the unnecessary complexity that stems from multiple constructors, multiple optional parameters and overuse of setters.

Imagine you have a class with a substantial amount of attributes like the *User* class below. Let's assume that you want to make the class immutable (which, by the way, unless there's a really good reason not to you should always strive to do. But we'll get to that in a different post).

```
1  public class User {
2      private final String firstName;    //required
3      private final String lastName;    //required
4      private final int age;    //optional
5      private final String phone;    //optional
6      private final String address;    //optional
7  ...
8  }
```

Now, imagine that some of the attributes in your class are required while others are optional. How would you go about building an object of this class? All attributes are declared final so you have to set them all in the constructor, but you also want to give the clients of this class the chance of ignoring the optional attributes.

A first and valid option would be to have a constructor that only takes the required attributes as parameters, one that takes all the required attributes plus the first optional one, another one that takes two optional attributes and so on. What does that look like? Something like this:

```
public User(String firstName, String lastName) {
    this(firstName, lastName, 0);
}

public User(String firstName, String lastName, int age) {
    this(firstName, lastName, age, "");
}

public User(String firstName, String lastName, int age, String phone)
    this(firstName, lastName, age, phone, "");
}

public User(String firstName, String lastName, int age, String phone,
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.phone = phone;
    this.address = address;
}
```

The good thing about this way of building objects of the class is that it works. However, the problem with this approach should be pretty obvious. When you only have a couple of attributes is not such a big deal, but as that number increases the code becomes harder to read and maintain. More importantly, the code becomes increasingly harder for clients. Which constructor should I invoke as a client? The one with 2 parameters? The one with 3? What is the default value for those parameters where I don't pass an explicit value? What if I want to set a value for address but not for age and phone? In that case I would have to call the constructor that takes all the parameters and pass default values for those that I don't care about. Additionally, several parameters with the same type can be confusing. Was the first String the phone number or the address?

So what other choice do we have for these cases? We can always follow the JavaBeans convention, where we have a default no-arg constructor and have setters and getters for every attribute. Something like:

```
public class User {
    private String firstName; // required
    private String lastName; // required
    private int age; // optional
    private String phone; // optional
    private String address;  //optional

    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
```

```
        this.lastName = lastName;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getPhone() {
        return phone;
    }
    public void setPhone(String phone) {
        this.phone = phone;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
}
```

This approach seems easier to read and maintain. As a client I can just create an empty object and then set only the attributes that I'm interested in. So what's wrong with it? There are two main problems with this solution. The first issue has to do with having an instance of this class in an inconsistent state. If you want to create an *User* object with values for all its 5 attributes then the object will not have a complete state until all the *setX* methods have been invoked. This means that some part of the client application might see this object and assume that is already constructed while that's actually not the case.
The second disadvantage of this approach is that now the *User* class is mutable. You're loosing all the benefits of immutable objects.

Fortunately there is a third choice for these cases, the builder pattern. The solution will look something like the following.

```
public class User {
    private final String firstName; // required
    private final String lastName; // required
    private final int age; // optional
    private final String phone; // optional
    private final String address; // optional

    private User(UserBuilder builder) {
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.age = builder.age;
        this.phone = builder.phone;
        this.address = builder.address;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public int getAge() {
        return age;
    }

    public String getPhone() {
        return phone;
    }

    public String getAddress() {
        return address;
```

```
34      }
35
36      public static class UserBuilder {
37          private final String firstName;
38          private final String lastName;
39          private int age;
40          private String phone;
41          private String address;
42
43          public UserBuilder(String firstName, String lastName) {
44              this.firstName = firstName;
45              this.lastName = lastName;
46          }
47
48          public UserBuilder age(int age) {
49              this.age = age;
50              return this;
51          }
52
53          public UserBuilder phone(String phone) {
54              this.phone = phone;
55              return this;
56          }
57
58          public UserBuilder address(String address) {
59              this.address = address;
60              return this;
61          }
62
63          public User build() {
64              return new User(this);
65          }
66
67      }
68  }
```

A couple of important points worth noting:

- The User constructor is private, which means that this class can not be directly instantiated from the client code.
- The class is once again immutable. All attributes are final and they're set on the constructor. Additionally, we only provide getters for them.
- The builder uses the Fluent Interface idiom to make the client code more readable (we'll see an example of this in a moment).
- The builder constructor only receives the required attributes and this attributes are the only ones that are defined "final" on the builder to ensure that their values are set on the constructor.

The use of the builder pattern has all the advantages of the first two approaches I mentioned at the beginning and none of their shortcomings. The client code is easier to write and, more importantly, to read. The only critique that I've heard about the pattern is the fact that you have to duplicate the class' attributes on the builder. However, given the fact that the builder class is usually a static member class of the class it builds, they can evolve together fairly easy.

Now, how does the client code trying to create a new *User* object looks like? Let's see:

```
1  public User getUser() {
2      return new
3              User.UserBuilder("Jhon", "Doe")
4              .age(30)
5              .phone("1234567")
6              .address("Fake address 1234")
7              .build();
8  }
```

Pretty neat, isn't it? You can build a *User* object in 1 line of code and, most importantly, is very easy to read. Moreover, you're making sure that whenever you get an object of this class is not going to be on an incomplete state.

This pattern is really flexible. A single builder can be used to create multiple objects by varying the builder attributes between calls to the "build" method. The builder could even auto-complete some generated field between each invocation, such as an id or serial number.

An important point is that, like a constructor, a builder can impose invariants on its parameters. The build method can check these invariants and throw an *IllegalStateException* if they are not valid.
It is critical that they be checked after copying the parameters from the builder to the object, and that they be checked on the object fields rather than the builder fields. The reason for this is that, since the builder is not thread-safe, if we check the parameters before actually creating the object their values can be changed by another thread between the time the parameters are checked and the time they are copied. This period of time is known as the "window of vulnerability". In our *User* example this could look like the following:

```
1  public User build() {
2      User user = new user(this);
3      if (user.getAge() 120) {
4          throw new IllegalStateException("Age out of range"); // thread-
5      }
6      return user;
7  }
```

The previous version is thread-safe because we first create the user and then we check the invariants on the immutable object. The following code looks functionally identical but it's not thread-safe and you should avoid doing things like this:

```
1  public User build() {
2      if (age 120) {
3          throw new IllegalStateException("Age out of range"); // bad, no
4      }
5      // This is the window of opportunity for a second thread to modify
6      return new User(this);
7  }
```

A final advantage of this pattern is that a builder could be passed to a method to enable this method to create one or more objects for the client, without the method needing to know any kind of details about how the objects are created. In order to do this you would usually have a simple interface like:

```
1  public interface Builder {
2      T build();
3  }
```

In the previous *User* example, the *UserBuilder* class could implement *Builder<User>*. Then, we could have something like:

```
1  UserCollection buildUserCollection(Builder<? extends User> userBuilder)
```

Well, that was a pretty long first post. To sum it up, the Builder pattern is an excellent choice for classes with more than a few parameters (is not an exact science but I usually take 4 attributes

to be a good indicator for using the pattern), especially if most of those parameters are optional. You get client code that is easier to read, write and maintain. Additionally, your classes can remain immutable which makes your code safer.

UPDATE: if you use Eclipse as your IDE, it turns out that you have quite a few plugins to avoid most of the boiler plate code that comes with the pattern. The three I've seen are:

- http://code.google.com/p/bpep/
- http://code.google.com/a/eclipselabs.org/p/bob-the-builder/
- http://code.google.com/p/fluent-builders-generator-eclipse-plugin/

I haven't tried any of them personally so I can't really give an informed decision on which one is better. I reckon that similar plugins should exist for other IDEs.

This entry was posted in **Best Practices**, **Java**, **Patterns** and tagged **builder**, **design pattern**, **immutable**, **java** by **jlordiales**. Bookmark the **permalink [https://jlordiales.wordpress.com/2012/12/13/the-builder-pattern-in-practice/]** .

43 THOUGHTS ON "THE BUILDER PATTERN IN PRACTICE"

Kenneth Truyers
on **December 14, 2012 at 11:49 pm** said:

Hello,