DESIGN PATTERNS

Exploring Joshua Bloch's Builder design pattern in Java

Bloch's Builder pattern can be thought of as a workaround for a missing language feature.

by Frank Kiwy

May 28, 2021

The Builder pattern, which is one of the 23 Gang of Four (GoF) design patterns described by Erich Gamma et al., is a creational design pattern that lets you construct complex objects step by step. It allows you to produce different types and representations of a product using the same construction code. However, this pattern should be used only if you need to build different immutable objects using the same building process.

The Builder pattern differs not very much from another important GoF creational pattern, the Abstract Factory pattern. While the Builder pattern focuses on constructing a complex object step by step, the Abstract Factory pattern emphasizes a family of `Product` objects, either simple or complex. Whereas the Builder pattern returns the final `Product` as a last step, the Abstract Factory pattern returns the `Product` immediately.

Although design patterns are language agnostic, their implementation varies from language to language depending on the features of each language, making some patterns even unnecessary, as I will show in the last section of this article.

In this article, I focus on Joshua Bloch's version of the Builder pattern (also known as the *Effective Java's Builder pattern*, named for his book). This version of the pattern is a variation on the GoF Builder pattern and is often confused with it.

Bloch's version of the Builder pattern provides a simple and safe way to build objects that have many optional parameters, so it addresses the telescoping constructor problem (which I describe shortly). In addition, with large constructors, which in most cases have several parameters of the same type, it is not always

obvious which value belongs to which parameter. Therefore, the likelihood of mixing up parameter values is high.

The idiom used by Bloch's Builder pattern addresses these issues by creating a static inner `Builder` class that can be accessed without creating an instance of the outer class (the product being built) but that still has access to the outer private constructor.

For the sake of clarity, when I use the term *Builder pattern* going forward, I mean Bloch's version of the Builder pattern unless I specifically state otherwise.

Before diving any deeper, the following example class will be used throughout this article:

```
import builder.pattern.Genre;
import java.time.Year;

public class Book {
    private final String isbn;
    private final String title;
    private final Genre genre;
    private final String author;
    private final Year published;
    private final String description;
    public Book(String isbn, String title, Ge
        this.isbn = isbn;
        this.title = title;
        this.genre = genre;
        this.author = author;
        this.published = published;
        this.description = description;
    }

    public String getIsbn() {
        return isbn;

    }
    public String getTitle() {

        return title;
    }

    public Genre getGenre() {
        return genre;
    }

    public String getAuthor() {
        return author;
    }

    public Year getPublished() {
        return published;
    }

    public String getDescription() {
        return description;
    }

}
```

The `Book` class has six final fields, one constructor taking all the parameters to be set, and the corresponding getters to read the object's fields once the object has been created. As a consequence, all objects derived from this class are immutable.

Further, the `Book` class has two mandatory fields: ISBN, which refers to a book's 10-digit or 13-digit International Standard Book Number, and `Title`. All remaining fields are optional.

Now the question arises, how can you construct objects with different combinations of optional parameters by using an appropriate constructor for each given combination? Because the objects are intended to be immutable, Enterprise JavaBean–like setters are out of question.

Telescoping constructors

One possible solution consists of *telescoping constructors*, where the first constructor takes only the mandatory fields; for every optional field, there is a further constructor that takes the mandatory fields *plus* the optional fields. Every constructor calls the subsequent one by passing a `null` value in place of the missing parameter. Only the final constructor in the chain will set all the fields by using the values provided by the parameters.

Below, you can see the `Book` class with the telescoping constructor solution.

```
import builder.pattern.Genre;
import java.time.Year;

public class Book {
    private final String isbn;
    private final String title;
    private final Genre genre;
    private final String author;
    private final Year published;
    private final String description;

    public Book(String isbn, String title) {
        this(isbn, title, null);
    }

    public Book(String isbn, String title, Ge
        this(isbn, title, genre, null);
    }

    public Book(String isbn, String title, Ge
        this(isbn, title, genre, author, null
    }

    public Book(String isbn, String title, Ge
        this(isbn, title, genre, author, publ
    }

    public Book(String isbn, String title, Ge
        this.isbn = isbn;
        this.title = title;
        this.genre = genre;
```

```
        this.author = author;
        this.published = published;
        this.description = description;
    }

    public String getIsbn() {
        return isbn;
    }

    public String getTitle() {
        return title;
    }

    public Genre getGenre() {
        return genre;
    }

    public String getAuthor() {
        return author;
    }

    public Year getPublished() {
        return published;
    }

    public String getDescription() {
        return description;
    }

}
```

Unfortunately, the telescoping constructors will not prevent you from having to pass `null` values in some cases. For instance, if you had to create a `Book` with `ISBN`, `title`, and `author`, what would you do? There is no such constructor!

You would probably use an existing constructor and pass a `null` value in place of the missing parameter.

```
new Book("0-12-345678-9", "Moby-Dick", null,
```

However, the use of `null` values can be avoided by creating an appropriate constructor, as follows:

```
public Book(String isbn, String title, String
    this.isbn = isbn;
    this.title = title;
    this.author = author;
}
```

The resulting constructor call should work fine but may lead to a different problem.

```
new Book("0-12-345678-9", "Moby-Dick", "Herma
```

Imagine you also had to create a `Book` with `ISBN` and `title` but with `description` instead of `author`. You might be tempted to add a constructor like the following:

```
public Book(String isbn, String title, String
    this.isbn = isbn;
    this.title = title;
    this.description = description;
}
```

This would not work. Two constructors of the same signature cannot coexist in the same class, because the compiler would not know which one to choose. In addition, creating a constructor for every useful combination of parameters would result in a large combination of constructors, making the resulting code hard to read and even harder to maintain.

Therefore, neither telescoping constructors nor any other possible combination of constructor parameters is a good approach to solve the issues related to the construction of objects that have numerous optional fields.

This is where Bloch's version of the Builder pattern comes in.

The Effective Java Builder pattern

As mentioned earlier, Bloch's Builder pattern is a variation of the GoF Builder pattern.

The GoF Builder pattern has four components: the `Director`, the `Builder` (interface), the `ConcreteBuilder` (implementation), and the `Product`. I will not go into the individual components here, because that is beyond the scope of this article.

Bloch's Builder pattern is shorthand for the GoF's counterpart in the sense that it consists of only two of the four components: the `ConcreteBuilder` and the `Product`. In addition, Bloch's Builder has a Java-specific implementation since the `Builder` consists of a nested static class (located inside the `Product` class itself).

If fact, the idiom is a workaround for a missing language feature, which is the lack of named parameters, rather than an object-oriented design pattern.

How does it work?

First, you create an instance of the `Builder` class by passing the mandatory fields to its constructor. Then, you set the values for the optional fields by calling the setter-like methods of the `Builder` class. Once you have set all the fields, you call the `build` method on the `Builder` instance. This method creates the `Product` by passing the previously set values to the

`Product`'s constructor, and it eventually returns a new `Product` instance.

Here is the implementation.

```
import builder.pattern.Genre;
import java.time.Year;

public class Book {
    private final String isbn;
    private final String title;
    private final Genre genre;
    private final String author;
    private final Year published;
    private final String description;
    private Book(Builder builder) {
        this.isbn = builder.isbn;
        this.title = builder.title;
        this.genre = builder.genre;
        this.author = builder.author;
        this.published = builder.published;
        this.description = builder.descriptio
    }

    public String getIsbn() {
        return isbn;
    }

    public String getTitle() {
        return title;
    }

    public Genre getGenre() {
        return genre;
    }

    public String getAuthor() {
        return author;
    }

    public Year getPublished() {
        return published;
    }

    public String getDescription() {
        return description;
    }

    public static class Builder {
        private final String isbn;
        private final String title;
        private Genre genre;
        private String author;
        private Year published;
        private String description;

        public Builder(String isbn, String ti
            this.isbn = isbn;
            this.title = title;
        }

        public Builder genre(Genre genre) {
            this.genre = genre;
            return this;
        }
```

```
        public Builder author(String author)
            this.author = author;
            return this;
        }

        public Builder published(Year publish
            this.published = published;
            return this;
        }

        public Builder description(String des
            this.description = description;
            return this;
        }

        public Book build() {
            return new Book(this);
        }

    }

}
```

The following are some things to note:

- The scope of the `Book` constructor has been changed to `private`, so that it cannot be accessed from the outside of the `Book` class. This makes it impossible to create a `Book` instance directly. The object creation process is delegated to the `Builder` class.
- The `Book` constructor takes a `Builder` instance as its only parameter, which contains all the values to be set by the `Book` constructor. Alternatively, the `Book` constructor could take all the parameters corresponding to the `Book` fields, but this would mean that you must deal again with many parameters to be set in the right order when you call the `Book` constructor from the `Builder`'s `build` method. Mixing up parameters of the same type is one of the potential issues developers try to avoid by implementing the `Builder` pattern.
- The `Builder` class contains the same fields as the `Book` class, which is necessary to hold the values to be passed to the `Book` constructor. This has often been rightly criticized as code duplication.
- For every optional field to be set, the `Builder` class exposes a setter-like method, which assigns the field's value and returns the current `Builder` instance to build the object in a fluent way. Since each method call returns the same `Builder` instance, method calls can be chained, which makes the client code more concise and readable.
- The `build` method calls the `Book` constructor by passing the current `Builder` instance as the only parameter. The values held by the `Builder` instance are then unpacked by the `Book` constructor, which assigns them to the corresponding `Book` fields.

This is how the `Builder` is used.

```
Book book = new Book.Builder("0-12-345678-9",
                .genre(Genre.ADVENTURE_FICTIO
                .author("Herman Melville")
                .published(Year.of(1851))
                .description(
                        "The book is the sail
                        + "Ahab, captain of t
                        + "the giant white sp
                        + "off Ahab's leg at
                )
                .build();
```

Reusability and limitations

The Builder pattern also allows for reusing existing `Builder` instances, which already have been populated in a previous construction process. This makes it easy to create a new object that has only a few different attribute values, since you do not have to set all the values again.

Let's see how this works with the `Book` example. Herman Melville's *Moby Dick* has been published in several editions. The first was released in 1851. Another, which appeared in 1952, included a 25-page introduction and more than 250 pages of explanatory notes.

If you wanted to create a new `Book` object for the 1952 edition, you could simply reuse a previously created `Builder` instance for the 1851 version, override the publishing date, and call the `build` method again to produce a new `Book` object corresponding to the 1952 edition.

```
Book.Builder bookBuilder = new Book.Builder("
                .genre(Genre.ADVENTURE_FICTIO
                .author("Herman Melville")
                .published(Year.of(1851))
                .description("description omi

// Create a first Book object
Book book = bookBuilder.build();

// Create a second, slightly different, objec
book = bookBuilder.published(Year.of(1952)).b
```

However, the example above is not very realistic, because you also would have to change the ISBN—which is not possible since the ISBN field is `final` and, therefore, must be set via the `Builder`'s constructor. This, in turn, would result in the creation of a new `Book` instance. That example reveals the limits of the `Builder`'s reusability.

State validation

Bloch's Builder pattern also allows for convenient state validation during the construction process of the `Product` instance. Since all the `Book` fields are `final`, and thus can't be changed after a `Book` instance has been created, the state needs to be validated only once, specifically at construction time. The validation logic can be implemented (or called) either in the `Builder`'s `build` method or in the `Book` constructor. In the following example, the logic is called from the `build` method:

```
public static class Builder {
    private final IsbnValidator isbnValidator
    private final String isbn;
    private final String title;
    private Genre genre;
    private String author;
    private Year published;
    private String description;

    public Builder(String isbn, String title)
        this.isbn = isbn;
        this.title = title;
    }

    public Builder genre(Genre genre) {
        this.genre = genre;
        return this;
    }

    public Builder author(String author) {
        this.author = author;
        return this;
    }

    public Builder published(Year published)
        this.published = published;
        return this;
    }

    public Builder description(String descrip
        this.description = description;
        return this;
    }

    public Book build() throws IllegalStateEx
        validate();
        return new Book(this);
    }

    private void validate() throws IllegalSta
        MessageBuilder mb = new MessageBuilde
        if (isbn == null) {
            mb.append("ISBN must not be null.
        } else if (!isbnValidator.isValid(isb
            mb.append("Invalid ISBN!");
        }
        if (title == null) {
            mb.append("Title must not be null
        } else if (title.length() < 2) {
            mb.append("Title must have at lea
        } else if (title.length() > 100) {
            mb.append("Title cannot have more
        }
        if (author != null && author.length()
```

```
            mb.append("Author cannot have mor
        }
        if (published != null && published.is
            mb.append("Year published cannot
        }
        if (description != null && descriptio
            mb.append("Description cannot hav
        }
        if (mb.length() > 0) {
            throw new IllegalStateException(m
        }
    }

}
```

By calling the validation logic *before* the actual object is created, you can be guaranteed that every `Book` instance created by the `Builder` has a valid state.

Java records

My previous article, "Diving into Java records: Serialization, marshaling, and bean state validation," included an example of Bloch's Builder pattern implemented in a Java record. Indeed, records are well suited for Bloch's Builder implementation, because they are inherently immutable constructs.

```
public record BookRecord(String isbn, String

    private BookRecord(Builder builder) {
        this(builder.isbn, builder.title, bui
    }

    public static class Builder {
        private final IsbnValidator isbnValid
        private final String isbn;
        private final String title;
        private Genre genre;
        private String author;
        private Year published;
        private String description;

        public Builder(String isbn, String ti
            this.isbn = isbn;
            this.title = title;
        }

        public Builder genre(Genre genre) {
            this.genre = genre;
            return this;
        }

        public Builder author(String author)
            this.author = author;
            return this;
        }

        public Builder published(Year publish
            this.published = published;
            return this;
        }
```

```
        public Builder description(String des
            this.description = description;
            return this;
        }

        public BookRecord build() throws Ille
            validate();
            return new BookRecord(this);
        }

        private void validate() throws Illega
            MessageBuilder mb = new MessageBu
            if (isbn == null) {
                mb.append("ISBN must not be n
            } else if (!isbnValidator.isValid
                mb.append("Invalid ISBN!");
            }
            if (title == null) {
                mb.append("Title must not be
            } else if (title.length() < 2) {
                mb.append("Title must have at
            } else if (title.length() > 100)
                mb.append("Title cannot have
            }
            if (author != null && author.leng
                mb.append("Author cannot have
            }
            if (published != null && publishe
                mb.append("Year published can
            }
            if (description != null && descri
                mb.append("Description cannot
            }
            if (mb.length() > 0) {
                throw new IllegalStateExcepti
            }
        }

    }

}
```

The example above uses an alternative constructor to pass the `Builder` instance to the `record` constructor. In an alternative constructor, the canonical constructor (the one generated by the compiler) must be called *before* you can add any further statements. This means that the values of the `Builder` fields must be passed to the constructor parameters and, therefore, cannot be assigned directly to the `record` fields.

There's another choice: You can call the canonical constructor directly from the `Builder`'s `build` method. Either way, ensure that the constructor parameters are not mixed up.

Fortunately, with records, you do not have any code duplication as you have with regular classes, because the compiler generates the record fields and accessors. You can declare the fields only once explicitly in the `Builder` class.

Named parameters

If Java had named parameters, Bloch's version of the Builder pattern would be unnecessary, because you could provide only those parameters currently needed to create the object. Look at the following constructor:

```
public Book(String isbn = null, String title
    this.isbn = isbn;
    this.title = title;
    this.genre = genre;
    this.author = author;
    this.published = published;
    this.description = description;
}
```

Below are two examples of how the constructor can be called.

```
new Book(isbn = "0-12-345678-9", title = "Mob
new Book(isbn = "0-12-345678-9", title = "Mob
```

With named parameters, you need to define only a single constructor that works for all possible combinations of parameters. Thus, the number of parameters used and the order in which they are set does not matter. The omitted parameters take the default values specified in the constructor definition.

Conclusion

With Bloch's version of the Builder pattern, you can create objects that have many optional parameters without using cumbersome and error-prone telescoping constructors. Further, the pattern avoids mixing up parameter values in large constructors that often have multiple consecutive parameters of the same type.

In addition, the same `Builder` instance can be used to create other objects of the same type that have slightly different attribute values than the one created in the first construction process.

The Builder pattern also allows for easy state validation by implementing or calling the validation logic in the `build` method, before the actual object is created. This avoids the creation of objects with invalid state.

When the pattern is used with records, there is no code duplication as is the case with regular classes, which require the same fields to be specified in the `Product` and `Builder` classes.

Finally, if Java had named parameters, Bloch's version of the Builder pattern would be superfluous.

Dig deeper

- Diving into Java records: Serialization, marshaling, and bean state validation
- Review of Joshua Bloch's *Effective Java* (third edition)
- Interview of Google's Joshua Bloch

Frank Kiwy

Frank Kiwy is a senior software developer and project leader who works for a government IT center in Europe. His focus is on Java SE, Java EE, and web technologies. Kiwy is also interested in software architecture and is committed to continuous integration and delivery. He is currently involved in implementing the European Union's Common Agricultural Policy, where he's in charge of several projects. When programming, he values well-designed software with clear and easy-to-understand APIs.

Share this Page