Original article located at:
https://www.linkedin.com/pulse/design-patterns-builder-pattern-riaan-nel/?published=t

Design Patterns: The Builder Pattern

- Published on December 5, 2016

Riaan Nel

I've been meaning to write a series of articles on design patterns for quite a while. Patterns are incredibly valuable components in a developer's toolbox – they address common problems that have accepted, effective solutions. In addition, they contribute to a shared vocabulary amongst developers.

This series assumes an understanding of object-oriented programming (OOP). I will, however, try to keep the examples as simple and accessible as possible, favouring practical implementations over obscure examples. If you're looking for an authoritative, academic text on patterns, this is what you want: Design Patterns: Elements of Reusable Object-Oriented Software.

We'll start with the Builder pattern (one of my favourites). The Builder pattern is a creational pattern – in other words, it's used to create and configure objects. I particularly like the example that Joshua Bloch uses in Effective Java.

The Problem

For this example, we'll pretend that we're part of a Java team working on a piece of software for a bank. Among other things, we'll need a way to represent bank accounts. Our first pass looks like this (note that using double for actual monetary values is a bad idea).

```
public class BankAccount {
```

Original article located at:
https://www.linkedin.com/pulse/design-patterns-builder-pattern-riaan-nel/?published=t

```
    private long accountNumber;
    private String owner;
    private double balance;

    public BankAccount(long accountNumber, String owner, double
balance) {
        this.accountNumber = accountNumber;
        this.owner = owner;
        this.balance = balance;
    }

    //Getters and setters omitted for brevity.
}
```

This is reasonably straight-forward – we can use it as follows.

```
BankAccount account = new BankAccount(123L, "Bart", 100.00);
```

Unfortunately, solutions rarely stay simple. A new requirement arrives, which says that we should keep track of the monthly interest rate applicable to each account, the date on which it was opened, and optionally, the branch at which it was opened. It sounds easy enough, so we come up with version 2.0 of the **BankAccount** class.

```
public class BankAccount {

    private long accountNumber;
    private String owner;
    private String branch;
    private double balance;
    private double interestRate;

    public BankAccount(long accountNumber, String owner, String
branch, double balance, double interestRate) {
        this.accountNumber = accountNumber;
        this.owner = owner;
        this.branch = branch;
        this.balance = balance;
        this.interestRate = interestRate;
    }

    //Getters and setters omitted for brevity.
}
```

Thanks to our new and improved account handling process, we get some new clients.

```
BankAccount account = new BankAccount(456L, "Marge", "Springfield",
100.00, 2.5);
BankAccount anotherAccount = new BankAccount(789L, "Homer", null, 2.5,
100.00);  //Oops!
```

Our compiler, which should be our safety net, thinks that this code is fine. The practical implication, however, is that Homer's money will double every month*. Can you figure out why? Hint: pay close attention to the order of the parameters passed to the constructor.

If we have multiple consecutive arguments of the same type, it's easy to accidentally swop them around. Since the compiler doesn't pick it up as an error, it can manifest as an issue somewhere down the line at runtime – and that can turn into a tricky debugging exercise. In addition, adding more constructor parameters results in code that becomes harder to read . If we had 10 different parameters, it would become very difficult to identify what's what in the constructor at a single glance. To make it worse, some of those values might be optional, which means that we'll need to create a bunch of overloaded constructors to deal with all possible combinations, or we'll have to pass nulls to our constructor (ugly!).

You might be thinking that we can mitigate the issue by calling a no-arg constructor and then setting up the account via setter methods instead. However, that leaves us open to another issue – what happens if a developer forgets to call a particular setter method? We could end up with an object that is only partially initialized, and again, the compiler wouldn't see any problems with it.

Thus, there are two specific problems that we need to solve:

1. Too many constructor arguments.
2. Incorrect object state.

This is where the Builder pattern comes into play.

The Pattern

Original article located at:
https://www.linkedin.com/pulse/design-patterns-builder-pattern-riaan-nel/?published=t

The Builder pattern allows us to write readable, understandable code to set up complex objects. It is often implemented with a fluent interface, which you may have seen in tools like Apache Camel or Hamcrest. The builder will contain all of the fields that exist on the **BankAccount** class itself. We will configure all of the fields that we want on the builder, and then we'll use the builder to create accounts. At the same time, we'll remove the public constructor from the **BankAccount** class and replace it with a private constructor so that accounts can only be created via the builder.

For our example, we'll put the builder within the **BankAccount** class. It looks like this.

```
public class BankAccount {

    public static class Builder {

        private long accountNumber; //This is important, so we'll
pass it to the constructor.
        private String owner;
        private String branch;
        private double balance;
        private double interestRate;

        public Builder(long accountNumber) {
            this.accountNumber = accountNumber;
        }

        public Builder withOwner(String owner){
            this.owner = owner;

            return this;  //By returning the builder each time, we
can create a fluent interface.
        }

        public Builder atBranch(String branch){
            this.branch = branch;

            return this;
        }

        public Builder openingBalance(double balance){
            this.balance = balance;

            return this;
        }

        public Builder atRate(double interestRate){
```

Original article located at:
https://www.linkedin.com/pulse/design-patterns-builder-pattern-riaan-nel/?published=t

```
            this.interestRate = interestRate;

            return this;
        }

        public BankAccount build(){
            //Here we create the actual bank account object, which
is always in a fully initialised state when it's returned.
            BankAccount account = new BankAccount();  //Since the builder
is in the BankAccount class, we can invoke its private
constructor.
            account.accountNumber = this.accountNumber;
            account.owner = this.owner;
            account.branch = this.branch;
            account.balance = this.balance;
            account.interestRate = this.interestRate;

            return account;
        }
    }

    //Fields omitted for brevity.
    private BankAccount() {
        //Constructor is now private.
    }

    //Getters and setters omitted for brevity.

}
```

We can now create new accounts as follows.

```
BankAccount account = new BankAccount.Builder(1234L)
            .withOwner("Marge")
            .atBranch("Springfield")
            .openingBalance(100)
            .atRate(2.5)
            .build();

BankAccount anotherAccount = new BankAccount.Builder(4567L)
            .withOwner("Homer")
            .atBranch("Springfield")
            .openingBalance(100)
            .atRate(2.5)
            .build();
```

Is this code more verbose? Yes. Is it clearer? Yes. Is it better? Since a large chunk of our time is spent reading code rather than writing it, I'm pretty sure it is, yes.

Summary

We worked through an example where code started out simple, and then grew in complexity. We then used the Builder pattern to address the issues that we uncovered.

If you find yourself in a situation where you keep on adding new parameters to a constructor, resulting in code that becomes error-prone and hard to read, perhaps it's a good time to take a step back and consider refactoring your code to use a Builder.

That's it for now. Feel free to drop a comment if you have anything to ask or to add.

**If anyone knows of an account with returns like this, please let me know!*

Additional Reading

https://en.wikipedia.org/wiki/Builder_pattern