Insterface Segregation Principle in Java

1. Introduction

In this tutorial, we'll be discussing the Interface Segregation Principle, one of the SOLID principles. Representing the “I” in “SOLID”, interface segregation simply means that we should break larger interfaces into smaller ones.

Thus ensuring that implementing classes need not implement unwanted methods.

2. Interface Segregation Principle

This principle was first defined by Robert C. Martin as: “**Clients should not be forced to depend upon interfaces that they do not use**“.

The goal of this principle is to **reduce the side effects of using larger interfaces by breaking application interfaces into smaller ones**. It's similar to the Single Responsibility Principle, where each class or interface serves a single purpose.

Precise application design and correct abstraction is the key behind the Interface Segregation Principle. **Though it'll take more time and effort in the design phase of an application and might increase the code complexity, in the end, we get a flexible code.**

We'll look into some examples in the later sections where we have a violation of the principle, and then we'll fix the problem by applying the principle correctly.

3. Sample Interface and Implementation

Let's look into a situation where we've got a *Payment* interface used by an implementation *BankPayment*:

```
public interface Payment {
    void initiatePayments();
    Object status();
    List<Object> getPayments();
}
```

And the implementation:

```
public class BankPayment implements Payment {

    @Override
    public void initiatePayments() {
        // ...
    }

    @Override
    public Object status() {
        // ...
    }

    @Override
    public List<Object> getPayments() {
        // ...
    }
}
```

For simplicity, let's ignore the actual business implementation of these methods.

This is very clear — so far, the implementing class *BankPayment* needs all the methods in the *Payment* interface. Thus, it doesn't violate the principle.

4. Polluting the Interface

Now, as we move ahead in time, and more features come in, there's a need to add a *LoanPayment* service. This service is also a kind of *Payment* but has a few more operations.

To develop this new feature, we'll add the new methods to the *Payment* interface:

```
public interface Payment {

    // original methods
    void initiatePayments();
    Object status();
    List<Object> getPayments();

    // new requirements
    void intiateLoanSettlement();
    void initiateRePayment();
}
```

Next, we'll have the *LoanPayment* implementation:

```
public class LoanPayment implements Payment {

    @Override
    public void initiatePayments() {
        throw new UnsupportedOperationException("This is not a bank
payment");
    }

    @Override
    public Object status() {
        // ...
    }

    @Override
    public List<Object> getPayments() {
        // ...
    }

    @Override
    public void intiateLoanSettlement() {
        // ...
    }

    @Override
    public void initiateRePayment() {
        // ...
    }
}
```

Now, since the *Payment* interface has changed and more methods were added, all the implementing classes now have to implement the new methods. **The problem is, implementing them is unwanted and could lead to many side effects.** Here,
the *LoanPayment* implementation class has to implement the *initiatePayments()* without any actual need for this. And so, the principle is violated.

So, what happens to our *BankPayment* class:

```
public class BankPayment implements Payment {

    @Override
    public void initiatePayments() {
        // ...
    }

    @Override
    public Object status() {
        // ...
    }

    @Override
    public List<Object> getPayments() {
        // ...
    }

    @Override
```

```
    public void intiateLoanSettlement() {
        throw new UnsupportedOperationException("This is not a loan
payment");
    }

    @Override
    public void initiateRePayment() {
        throw new UnsupportedOperationException("This is not a loan
payment");
    }
}
```

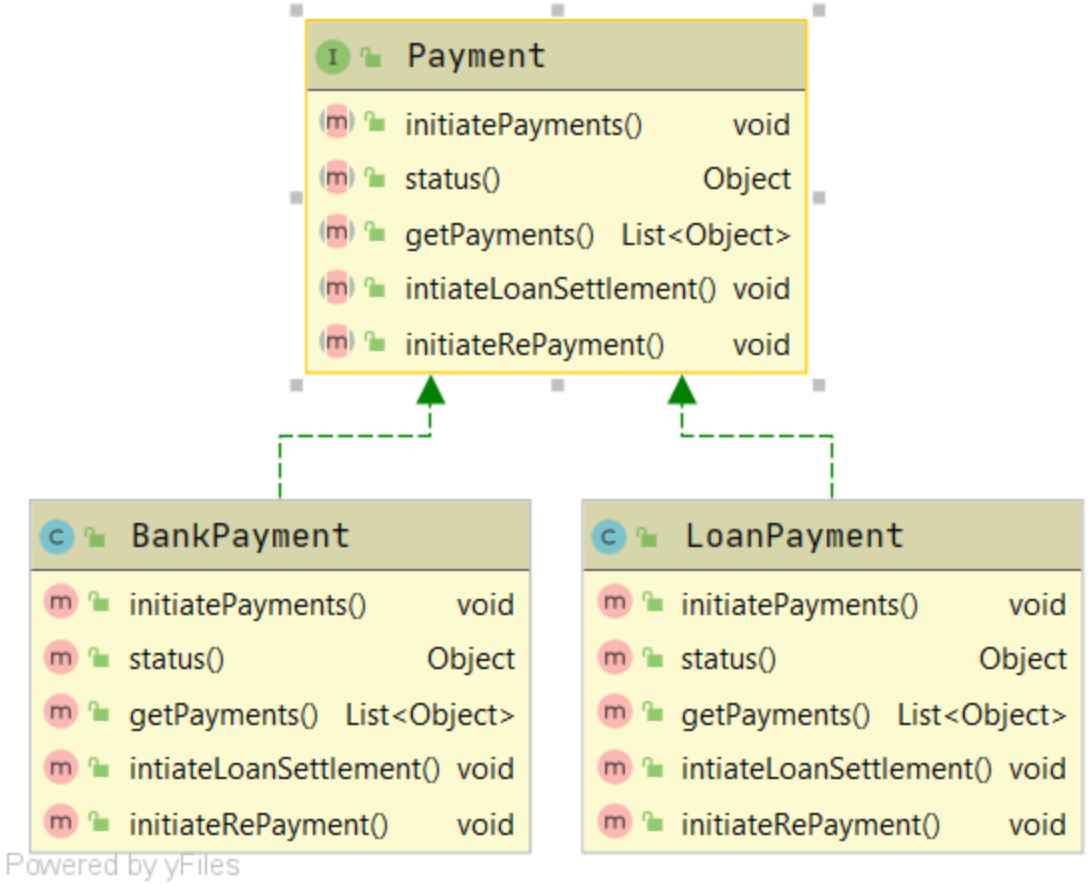
Note that the *BankPayment* implementation now has implemented the new methods. And since it does not need them and has no logic for them, it's **just throwing an *UnsupportedOperationException*. This is where we start violating the principle.**

In the next section, we'll see how we can solve this problem.

5. Applying the Principle

In the last section, we have intentionally polluted the interface and violated the principle. **In this section, we'll look into how to add the new feature for loan payment without violating the principle.**

Let's break down the interface for each payment type. The current situation:



Notice in the class diagram, and referring to the interfaces in the earlier section, that the *status()* and *getPayments()* methods are required in both the implementations. On the other hand, *initiatePayments()* is only required in *BankPayment*, and the *initiateLoanSettlement()* and *initiateRePayment()* methods are only for the *LoanPayment*.

With that sorted, let's break up the interfaces and apply the Interface Segregation Principle. Thus, we now have a common interface:

```
public interface Payment {
    Object status();
    List<Object> getPayments();
}
```

And two more interfaces for the two types of payments:

```
public interface Bank extends Payment {
    void initiatePayments();
}
public interface Loan extends Payment {
    void intiateLoanSettlement();
    void initiateRePayment();
}
```

And the respective implementations, starting with *BankPayment*:

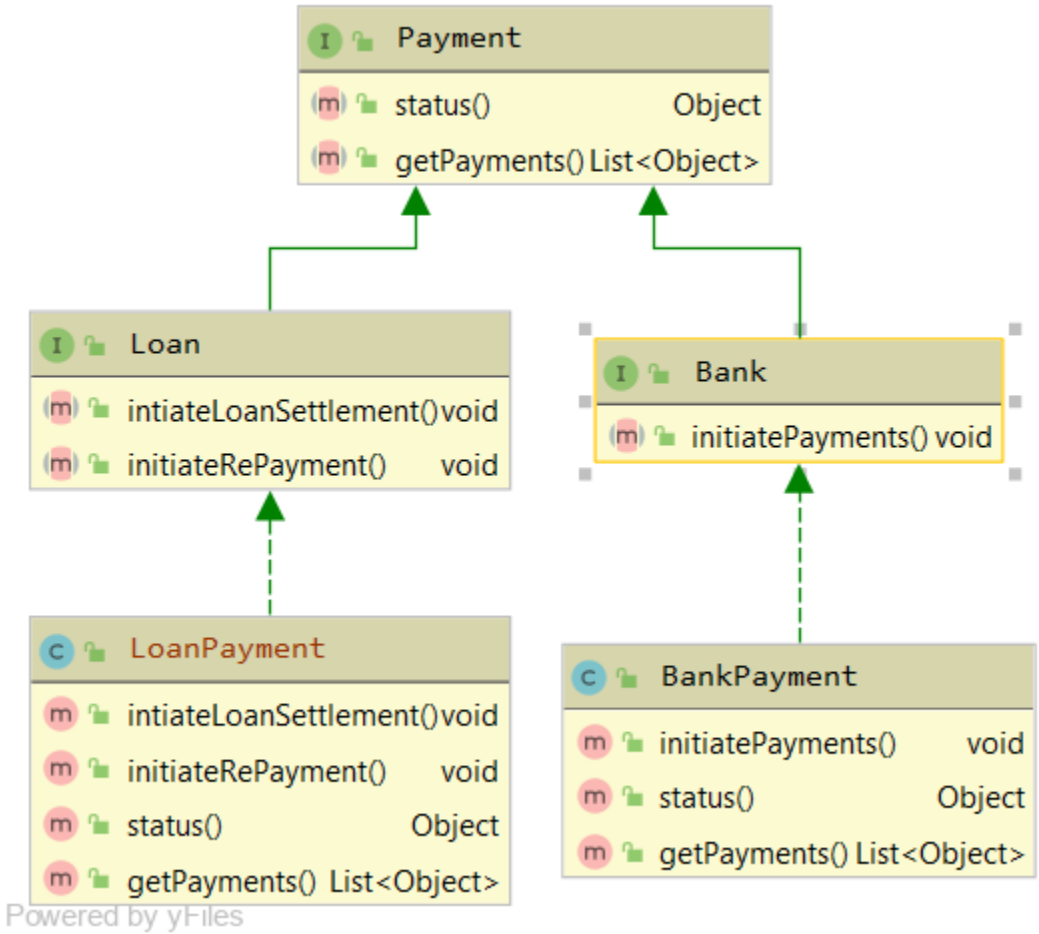
```
public class BankPayment implements Bank {
```

```
    @Override
    public void initiatePayments() {
        // ...
    }

    @Override
    public Object status() {
        // ...
    }

    @Override
    public List<Object> getPayments() {
        // ...
    }
}
```

And finally, our revised *LoanPayment* implementation:

```
public class LoanPayment implements Loan {

    @Override
    public void intiateLoanSettlement() {
        // ...
    }

    @Override
    public void initiateRePayment() {
        // ...
    }

    @Override
    public Object status() {
        // ...
    }

    @Override
    public List<Object> getPayments() {
        // ...
    }
}
```

Now, let's review the new class diagram:



As we can see, the interfaces don't violate the principle. The implementations don't have to provide empty methods. This keeps the code clean and reduces the chance of bugs.

6. Conclusion

In this tutorial, we looked at a simple scenario, where we first deviated from following the Interface Segregation Principle and saw the problems this deviation caused. Then we showed how to apply the principle correctly in order to avoid these problems.

In case we're dealing with polluted legacy interfaces that we cannot modify, the adapter pattern can come in handy.

The Interface Segregation Principle is an important concept while designing and developing applications. Adhering to this principle helps to avoid bloated interfaces with multiple responsibilities. This eventually helps us to follow the Single Responsibility Principle as well.

As always, the code is available over on GitHub.