Object Oriented Principles

Samudra Gupta

That Java is an Object Oriented language does not necessarily mean that the code written in Java is always Object Oriented. If this statement surprises you, this series is for you. In this series, I will try to demonstrate some design aspects, both good and bad, that are the key to well written software in Java. The first of these is the Liskov's Substitution Principle (LSP), which will lead to the Design by Contract and Dependency Inversion Principle. We will then see how all of them conform to one most vital principle of OO design, the Open-Closed Principle.

Liskov's Substitution Principle (LSP)

The *Liskov's Substitution Principle* provides a guideline to sub-typing any existing type. Stated formally it reads:

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

In a simpler term, if a program module is using the reference of a Base class, then it should be able to replace the Base class with a Derived class without affecting the functioning of the program module.

An example

When I was in the beginning of my IT career, I was working on a banking project and I was asked to design an account-handling module. To simplify the scenario, let us assume that I had two types of accounts to handle. One is the "Current Account" and the other is a special type of current account with a better interest rate, but with some restrictions that the account cannot be closed before a certain time period. Having done the preliminary analysis, I decided to come up with two account objects "CurrentAccount" and "SpecialCurrentAccount," and also decided to write another class that would offer the interfaces to operate on these "XXXAccount" objects. I also analysed that the "SpecialCurrentAccount" and "CurrentAccount" share lot in common. Delighted with such a straightforward relationship, I came up with the following class diagram. The SpecialCurrentAccount is a sub-type of the CurrentAccount.

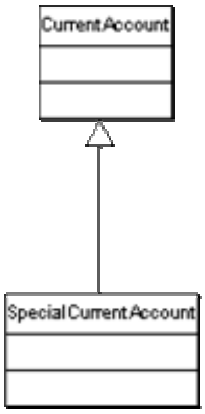

Figure 1.

The program specification for the module specified the following constraints.

The closing of Current Account should check for a balance greater than zero. If satisfied, proceed to close the account.

The closing of Special Current Account should check for a balance greater than zero and also that the minimum period for the account is covered. If satisfied, proceed to close the account.

From the above specification and my own class diagram, I decided that I will override a closeAccount() method in the derived SpecialAccount class. The following programs CurrentAccount.java and SpecialCurrentAccount.java describe the designed classes.

```
/*
 * Account.java
 *
 */

package sam.oo.bad;

public class CurrentAccount {

    /** Holds value of property balance. */
    protected int balance;

    /** Holds value of property period. */
    protected int period;

    /** Creates a new instance of Account */
    public CurrentAccount(int balance, int period) {
        this.balance = balance;
        this.period = period;
    }


    /**
     * open a current account with the given balance
     */
    public boolean openAccount(int balance)
    {
        this.balance = balance;
        return true;
    }


    /**
     *closes the account
     */
    public boolean closeAccount()
    {
        if(balance >0)
            return true;
        else
            return false;
    }
```

```
    /** Getter for property balance.
     * @return Value of property balance.
     */
    public int getBalance() {
        return this.balance;
    }


    /** Setter for property balance.
     * @param balance New value of property balance.
     */
    public void setBalance(int balance) {
        this.balance = balance;
    }


    /** Getter for property period.
     * @return Value of property period.
     */
    public int getPeriod() {
        return this.period;
    }


    /** Setter for property period.
     * @param period New value of property period.
     */
    public void setPeriod(int period) {
        this.period = period;
    }

}
```

Here is the code for the SpecialCurrentAccount.

```
/*
 * SpecialCurrentAccount.java
 *
 */

package sam.oo.bad;


public class SpecialCurrentAccount extends CurrentAccount{

    /** Holds value of property defaultPeriod. */
    private int defaultPeriod;

    /** Creates a new instance of SavingsAccount */
    public SpecialCurrentAccount(int balance, int period) {
        super(balance, period);
    }
```

```
    public boolean closeAccount()
    {
        if(balance>0 && period>defaultPeriod)
            return true;
        else
            return false;
    }


   /** Getter for property defaultPeriod.
     * @return Value of property defaultPeriod.
     */
    public int getDefaultPeriod() {
        return this.defaultPeriod;
    }

    /** Setter for property defaultPeriod.
     * @param defaultPeriod New value of property defaultPeriod.
     */
    public void setDefaultPeriod(int defaultPeriod) {
        this.defaultPeriod = defaultPeriod;
    }

}
```

Also I decided to offer an interface through another class to operate on these account objects, which looked like:

```
public void closeAnAccount(CurrentAccount ac)
    {
        System.out.println("Account close result: "+ac.closeAccount());
    }


    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        AccountTest test = new AccountTest();

        CurrentAccount ac = new CurrentAccount(100,2);
        SpecialCurrentAccount sac = new SpecialCurrentAccount(200,5);

        test.closeAnAccount(ac);
        test.closeAnAccount(sac);


    }
```

Everything went fine, and I was happy, until a user of this module discovered an unexpected behaviour with the system. When he tried to do the above, the result was contrary to his expectation.

Account close result: true
Account close result: false

The closing of a SpecialCurrentAccount object failed and the user has no clue why, without looking into the source code of the SpecialCurrentAccount class. But looking into interface closeAnAccount(CurrrentAccount), he is justified to make an assumption that any type of CurrentAccount object will behave in a similar manner. But in reality it did not. That made me think a while and I came to the following conclusions:

While designing the class hierarchy I have violated the LSP.

My overridden method closeAccount() in the derived class SpecialCurrentAccount had broken the module by not allowing a base class CurrentAccount object to be replaced by derived class object SpecialCurrentAcount.

By creating the above hierarchy I had expected the user to know about the internal implementation of both the account classes before he can be assured of a guaranteed behaviour of the module. Clearly, this is not desirable and often he might not have access to the source code to do so.

Design by Contract Principle

The above discussion leads us to the discussion of the *Design by Contract Principle*. In Design by Contract, each method in a class can have a pre-condition and a post-condition attached to it. The pre-condition defines the criteria to be met before the method offers a certain behaviour and the post-condition is the state or behaviour offered by the method once the pre-conditions are met. Following that, if we note down the pre-conditions and post-conditions followed by the CurrentAccount and the SpecialCurrentAccount classes, they will look like the following:

```
/*
 * CurrentAccount.java
 *
 */

/**
* pre condition: the balance >0
* post-condition: the account is closed
**/
public boolean closeAccount()

/*
 *SpecialCurrentAccount.java
 *
*/

/**
* pre condition: the balance >0
*                    the period > default period
* post condition: the account is closed
**/
public boolean closeAccount();
```

The pre-condition set in the sub-type SpecialCurrentAccount contains more conditions than the base-type CurrentAccount. This is against the Design by Contract principle. According to the *Design by Contract Principle*:

A sub-type can only have weaker pre-conditions and stronger post-conditions than its base class.

Clearly, my previous design violated the Design by Contract principle.Having these facts in mind, I had a clear idea what was going wrong and approached a better LSP and Design by Contract compliant solution.

LSP compliant solution

One of the main lessons I learned is that sub-typing needs to be done with respect to the behaviour of the types and not with respect to the data only. In this aspect, the SpecialCurrentAccount IS NOT A CurrentAccount. A banker will shout at me for sure if I say this to him. But for you the intelligent programmers this would make sense. This is because they do not exhibit the same behaviour against the same message. I decided to break the hierarchy and come up with is new design with Account as an abstract base type and CurrentAccount and SpecialCurrentAccount as its sub-types.

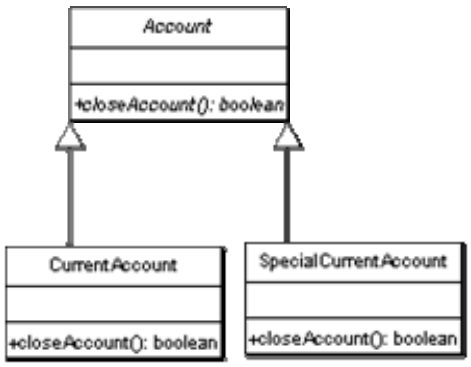

Figure 2.

The abstract Account class declares an abstract method closeAccount(), which is implemented by both CurrentAccount and SpecialCurrentAccount. I have now changed the interface class to accept an Account type of Object as opposed a particular type of Account Object. The new interface class looked like this.

public boolean closeAnAccount(Account ac)

With this hierarchy, what may surprise you is that we have not changed the implementation of the closeAccount() method in any of the classes. The pre-conditions and the post-conditions remain the same. But in essence, what has happened is that the user will not make any assumption about the behaviour of the account object he is dealing with. This makes the module more maintainable and reusable in the sense that now it is very easy to add another type of account which may impose some other pre-condition and post-condition without breaking the LSP and Design by Contract.

No overriding!!!

It may seem from the above discussion that overriding is the main problem with LSP and in that case inheritance makes no sense. To some extent, it is puzzling but always keep the Design by Contract principle in mind before you override. If you are unable to comply with the same, then it might be worth revisiting your class diagrams. Always make sure that the derived class must enforce less strict pre-conditions while overriding any base class method. Try to think about the situation on your own and surely you will come up with some logic why LSP is so important for a good OO design.

Designing Abstraction

While LSP describes different considerations to be made while creating sub-types in the application architecture, the abstraction is all about the design flexibility.

Is it Open or Closed? This is the first question you should ask yourself after you have finished writing your application module. Beware if the answer is **closed**. It means you need to take a fresh look at your design. Object Oriented software does not only mean that your application is a combination of several objects but it must satisfy the requirements that the design is flexible, and OPEN for extension.

Open-Closed Principle (OCP)

A software module which should be closed for modification but open for extension.

You may wonder how it is possible to extend an application without modifying it. Precisely, a good OO design gives you this power. Remember, bug fixing your application is not an extension. Think about how Object1 uses Object 2 and Object 2 provides some business logic to Object1. This business logic may be customer specific. To meet different customer needs, if you have to change the business logic embedded in Object 2, your application is violating OCP.

So far so good but the example speaks more than a thousand words and I will present examples to illustrate the points about OCP. If you follow the examples, you should be able to understand and apply OCP in your own application design.

Strong Coupling

What it is? It means that one application module is totally dependent on the implementation of another module. This is the first sign of violation of OCP. Consider this example. You are going to design a banking application where a particular loan request is passed through a *LoanValidator* to approve the loan request. The *LoanValidator* looks to see if the balance is above certain values and then approves the request. Given this problem, one guy comes up with the following classes.

The *LoanRequestHandler* can assess a loan request. This class holds the balance and period of existence for a particular bank account. I have simplified all other features that this class should normally bear to represent a physical bank account. But still this is sufficient to make the point.

```
/*
 *LoanRequestHandler.java
 *
*/

public class LoanRequestHandler {

        private int balance;
        private int period;
```

```
        /** Creates a new instance of LoanRequestHandler */
        public LoanRequestHandler(int balance, int period) {
                this.balance = balance;
                this.period = period;
        }

        public void approveLoan(PersonalLoanValidator validator) {
                if(validator.isValid(balance))
                  //sanction the loan
                  System.out.println("Loan approved...");
                else
                  System.out.println("Sorry not enough balance...");
        }
}
```

This program can tell you if your loan request is approved or not. It uses another object *PersonalLoanValidator* to decide if the loan request is valid or not. The *PersonalLoanValidator* just checks to see if the balance is more than 1000 and approves the loan or else rejects it.

```
/*
 * PersonalLoanValidator.java
 *
*/

public class PersonalLoanValidator {
        /** Creates a new instance of PersonalLoanValidator */
        public PersonalLoanValidator() {
        }

        public boolean isValid(int balance) {
                if(balance>1000)
                        return true;
                else
                        return false;
        }
}
```

Problem

Do you see the problem with the above design? Probably yes. The *LoanRequestHandler* object is strongly coupled with *PersonalLoanValidator* object. What happens if the *LoanRequestHandler* object should be used to verify a business loan. Certainly, your business logic to approve a business loan will be different and *PersonalLoanValidator* is not capable of handling this business logic.

One solution may to be add another method to the *PersonalLoanValidator* class to verify the business loan approval process. Also we need to modify the *LoanRequestHandler* object with an if-else loop to apply different *Validator* objects for personal loans and business loans. This means every time the bank decides to provide a different type of loan, we need to make change in the *LoanRequestHandler* and *PersonalLoanValidator* objects.

This application is certainly not closed for modification. This is where the OCP comes into the picture. This application has violated the OCP and therefore it is not properly designed.

To make the design OCP compliant, we need to make the following changes:

The strong coupling between *LoanRequestHandler* and any *Validator* objects should be avoided.

Individual validator objects implementing different business logics for approving a loan request should represent an *abstract* type and the L*oanRequestHandler* should use this *abstract* type rather than any specific sub-type to avoid any strong coupling.

Bearing these facts in mind, we come up with the following design. I will only show the skeleton of the modified design and the code inside each of them remains same as before. Only the design changes, not the implementation.

```
public class LoanRequestHandler {

        private int balance;
        private int period;
        /** Creates a new instance of LoanRequestHandler */
        public LoanRequestHandler(int balance, int period) {
                this.balance = balance;
                this.period = period;
        }

        public void approveLoan(Validator validator) {
                if(validator.isValid(balance))
                  //sanction the loan
                  System.out.println("Loan approved...");
                else
                  System.out.println("Sorry not enough balance...");
        }
}
```

The *LoanRequestHandler* now uses an interface type of object named *Validator*, which defines the method for approving the loan. Each individual *Validator* object will implement this method, with the object specific business logic within the method inherited from the defined *Validator* interface.

```
public interface Validator {

  public Boolean isValid(int balance);
}
```

Now each individual *Validator* object will implement this interface and make use of their individual business logic to approve or reject a loan request. Based on this the *PersonalLoanValidator* object changes to the following.

```
public class PersonalLoanValidator implements Validator{

	/** Creates a new instance of PersonalLoanValidator */
	public PersonalLoanValidator() {
	}

	public boolean isValid(int balance) {
		if(balance>1000)
			return true;
		else
			return false;
	}
}
```

Now we will implement another *Validator* named *BusinessLoanValidator*.

```
public class BusinessLoanValidator implements Validator{
	/** Creates a new instance of PersonalLoanValidator */
	public PersonalLoanValidator() {
	}

	public boolean isValid(int balance) {
		if(balance>5000)
		  return true;
		else
		  return false;
	}
}
```

The *BusinessLoanValidator* only approves the loan request if the balance is more than 5000, whereas the *PersonalLoanValidator* will approve it if the balance is more than 1000. But the *LoanRequestHandler* remains unaffected by this change in the business logic change. So long it is supplied the correct *Validator*, it will operate correctly.

Clearly, this is a better design. Any new loan type can be handled by adding a new *XXXValidator* object, implementing the *Validator* interface. This is OCP compliant solution.

The OCP is the guiding principle for good Object-Oriented design. It is easy and it is difficult. The main problem is how you look at your design. The design typically has two aspects with it. One, you design an application module to make it work and second, you need to take care whether your design, and thereby your application, module is reusable, flexible and robust.

The OCP tells you that the software module should be open for extension but closed for modification. As you might have already started thinking, this is a very high level statement and the real problem is how to achieve this. Well, it comes through practice, experience and constant inspection of any piece of design, understanding that how it performs and works tackles with the expanding requirements of the application. But even though you are a new designer, you can follow certain principles to make sure that your design is a good one.

Dependency Inversion Principle (DIP)

One of these principles is the *Dependency Inversion Principle*, which helps you make your design OCP compliant. Formally stated, the principle makes two points:

High level modules should not depend upon low level modules. Both should depend upon abstractions

Abstractions should not depend upon details. Details should depend upon abstractions.

What this means? Think about a typical software design process. You typically start with high-level modules. For example, if you are designing a module, which collects data from a data source and writes to a database, you will try to break down the architecture in the following way.

There is a component which accepts certain objects and writes data to the database.

There is a component which reads data from a certain data source and creates certain suitable objects for the data writing component.

Look at it like this, your idea started from the point that you need to write some data to the database and then you think about getting the data in an acceptable format. There is nothing wrong in this process. This is what your application looks like from the high-level. But there is a danger if you fail to transform your thinking into a right design.

Consider, given the above problem, you come up with the following design in Figure 1.

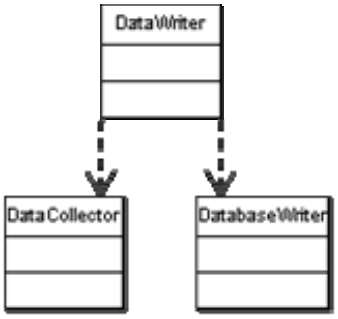

Figure 3.

The *DataWriter* class uses the *DataCollector* class. The *DataCollector* class is responsible for collecting the data from some data source and passing the required object to the *DataWriter* class. The *DataWriter* class in turn accepts the passed object and uses another class *DatabaseWriter* to write data to the database.

Assuming this architecture, the pseudo-code for the above classes may look like this.

```
\*
 *DataCollector.java
 *
*\

public class DataCollector
{
	public void collectData(String source) {

		//collect the data from source, let us say String data

		//initialise a DataWriter object
		DataWriter writer = new DataWriter();

		//ask the writer object to write the data
		writer.writeData(the collected data);
		}
}
```

The pseudo-code for the *DataWriter* class may look like this.

```
\*
 * DataWriter.java
 *
*\


public class DataWriter {

	public void writeData(the data) {
	  //write the data to the database
	  DatabaseWriter dw = new DatabaseWriter();
	  dw.writeToDB(params);
	}
}
```

The pseudo code for the *DatabaseWriter* can in turn look like this.

Listing 8 DatabaseWriter.java

```
public class DatabaseWriter
{

	public void writeToDB(params) {

		//write physically to the database
	}
}
```

Look at this design, implement it and your application will run fine. Now there is a problem, your high level component is dependent on the details of the low level implementation. Clearly,

it is tied to the implementation of the *DatabaseWriter* object. This makes your design inflexible. Really, what you want your *DataWriter* object to be able to write to any destination. This gives you the ability to reuse your *DataWriter* class to write data to any other destination should the application require it in the future.

One solution may be to put an if-else loop within the *DataWriter*, so that it can decide which destination to write to. We have discussed in OCP that this is the point not at all desirable. This makes your software open for modification. A better approach of thinking would be:

"We need to make *DataWriter* object independent of any specific destination."

How can we do this? Abstraction is the answer. This is the key for flexible design. Make the abstractions depend on each other, not the concrete implementations. Look at the following modified design in Figure 4.

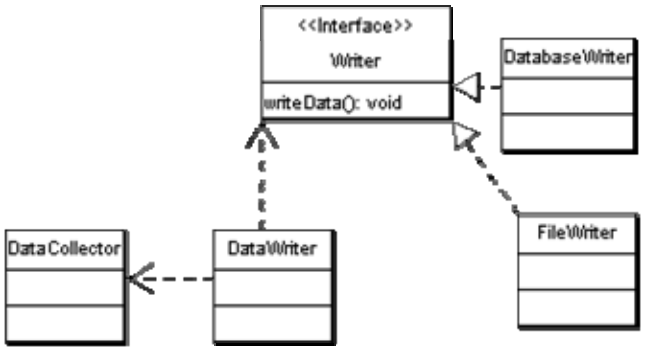

Figure 4.

In this design, we have abstracted the implementation of the data writing components. Now the *DataWriter* class uses the <<interface>> *Writer*. There are two concrete implementations of this interface viz. *DatabaseWriter* and *FleWriter*. You can pass any of the implementations depending on the requirement. Notice, that this design is flexible because you can add any other *Writer* to the application structure should it be required.

All said and done, why is it called Dependency Inversion Principle? The answer is again in thought thinking patterns. In the traditional approach, we start thinking from the high-level modules and cascade down to the lower levels. For example, we initially thought, we need an object to write data to the database and then created another object to write the data to the database and strongly coupled the high-level writer object to the low-level database writer object. This led to the bad design.

In Dependency Inversion Principle, we started thinking from the low-level modules. We identified that we may need to write to different destinations and thereby may need many *Writer* objects in the system. But all of them are writing data to somewhere and abstracted it to the *Writer* object. Then we associated this abstraction to the high-level module, the *DataWriter*.

You might at this point question and might have truly experienced that designing with abstraction is sometimes overkill. In many cases, the requirement is rigid and unlikely to change too often. It is a trade-off, as to how far we go and how much time we spend on a good design. It may be quite important to get a reasonable designs out to the developers and let the project get on. True, but again if you look at any project, you will often find distinct layers of application components. They may typically be like this:

High-level policy objects holding different piece of business logic or common functionality across different modules.

The Policy level components use some middle layer objects to perform different business level operations such as sending messages to other components, writing data to some data source etc.

The lowest-level objects are Utility objects, which hold the concrete implementation of different operations.
In my opinion, for a reasonable design, you need to concentrate on the Policy level objects. It is important to get them well designed as a policy can be reused by many other applications. The rest of the layers can be left for now if you have run out of time and resource.

Conclusion

Hope you have understood the power of OO designs following the LSP, OCP and DIP. It might need some practice in the initial days of design but once you get it by heart, this becomes easy. The important thing is to consider the alternate scenarios that can arise to break your application design. If you find them, try to see if your design can handle them. If not, think about these principles and you might see some light.