
S.O.L.I.D
OBJECT ORIENTED DESIGN


CLASSIFIED
CLASSIFIED
CLASSIFIED
CLASSIFIED
CLASSIFIED

Single Responsibility Principle

In object-oriented programming (Java, among other languages, follows this paradigm), you will often hear terms such as robustness, cohesion, coupling etc. **Cohesion** is a way to measure how much the code segments within one module (methods of a class, classes inside a package…) belong together. The higher the cohesion – the better, since high cohesion implies easier maintenance and debugging, greater code functionality and reusability. The term cohesion is sometimes contrasted with the concept of **coupling**, and often, loose coupling of modules is related to high cohesion.
Another widely used term is **robustness,** which could be defined as the ability of a computer system or algorithm to handle mistakes and malfunctions (which could be caused by various factors such as programmer's mistake or incorrectly formatted user input). A robust system is one that can handle these unwanted situations elegantly. There are various ways for a software engineer to achieve robustness, such as testing the code for different kinds of inputs, but generally, in order to achieve robustness (and high cohesion), programmers follow a certain set of rules and principles for better organization of object-oriented programs. One such principle is the single responsibility principle.

The single responsibility principle revolves around the claim that a certain code module (most often, a class) should only have responsibility over one part of the functionality provided by the software. In software engineering books, this is sometimes also defined like this: the module should only have **one reason to change**. This means that a division of concerns is performed in the program, and the methods for every concern should be completely encapsulated by a single class. Now it is obvious that this approach contributes to the high cohesion – since methods related to the same concern (same part of the functionality) will be members of the same class, and robustness – since this reduces the possibility of error. Furthermore, if an error does occur, the programmer will be more likely to find the cause, and finally, solve the problem.
The single responsibility principle is founded on one of the basic, general ideas of object-oriented programming – the so-called *divide and conquer* principle – solving a problem by solving its multiple sub-problems. This approach prevents the creation of *"God objects"* – objects that "*know too much or do too much*".
The classes you write, should not be a swiss army knife. They should do one thing, and to that one thing well.

(Bad) Example

Let's consider this classic example in Java – "objects that can print themselves".

```
class Text {
    String text;
    String author;
    int length;
    String getText() { ... }
    void setText(String s) { ... }
    String getAuthor() { ... }
    void setAuthor(String s) { ... }
    int getLength() { ... }
    void setLength(int k) { ... }
    /*methods that change the text*/
    void allLettersToUpperCase() { ... }
    void findSubTextAndDelete(String s) { ... }
    /*method for formatting output*/
    void printText() { ... }
}
```

At first glance, this class might look correctly written. However, it contradicts the single responsibility principle, in that it has multiple reasons to change: we have tho methods which change the text itself, and one which prints the text for the user. If any of these methods is called, the class will change. This is also not good because it mixes the logic of the class with the presentation.

Better Example

One way of fixing this is writing another class whose only concern is to print text. This way, we will separate the functional and the “cosmetic” parts of the class.

```
class Text {
    String text;
    String author;
    int length;
    String getText() { ... }
    void setText(String s) { ... }
    String getAuthor() { ... }
    void setAuthor(String s) { ... }
    int getLength() { ... }
    void setLength(int k) { ... }
    /*methods that change the text*/
    void allLettersToUpperCase() { ... }
    void findSubTextAndDelete(String s) { ... }
}
class Printer {
    Text text;
    Printer(Text t) {
       this.text = t;
    }
    void printText() { ... }
}
```

Summary

In the second example we have divided the responsibilities of editing text and printing text between two classes. You can notice that, if an error occurred, the debugging would be easier, since it wouldn’t be that difficult to recognize where the mistake is. Also, there is less risk of accidentally introducing software bugs, since you’re modifying a smaller portion of code.
Even though it’s not that noticeable in this example (since it is small), this kind of approach allows you to see the “bigger picture” and not lose yourself in the code; it makes programs easier to upgrade and expand, without the classes being too extensive, and the code becoming confusing.

Single Responsibility Principle in Spring

As you become more comfortable using Spring components and coding to support Inversion of Control and Dependency Injection in Spring, you will find your classes will naturally adhere to the single responsibility principle. A typical violation of the single responsibility principle I often see in legacy Spring applications is an abundance of code in controller actions. I’ve seen Spring controllers getting JDBC connections to make calls to the database. This is a clear violation of the single responsibility principle. Controller objects have no business interacting with the database. Nor do controllers have any business implementing other business logic. In practice your controller methods should be very simple and light. Database calls and other business logic belong in a service layer.

Open Closed Principle

As applications evolve, changes are required. Changes are required when a new functionality is added or an existing functionality is updated in the application. Often in both situations, you need to modify the existing code, and that carries the risk of breaking the application's functionality. For good application design and the code writing part, you should avoid change in the existing code when requirements change. Instead, you should extend the existing functionality by adding new code to meet the new requirements. You can achieve this by following the Open Closed Principle.

The Open Closed Principle represents the "O" of the five SOLID software engineering principles to write well-designed code that are more readable, maintainable, and easier to upgrade and modify. Bertrand Meyer coined the term Open Closed Principle, which first appeared in his book Object-Oriented Software Construction, release in 1988. This was about eight years before the initial release of Java.

This principle states: "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification ". Let's zero in on the two key phrases of the statement:

1. "Open for extension ": This means that the behavior of a software module, say a class can be extended to make it behave in new and different ways. It is important to note here that the term "extended " is not limited to inheritance using the Java extend keyword. As mentioned earlier, Java did not exist at that time. What it means here is that a module should provide extension points to alter its behavior. One way is to make use of polymorphism to invoke extended behaviors of an object at run time.
2. "Closed for modification ": This means that the source code of such a module remains unchanged.

It might initially appear that the phrases are conflicting- How can we change the behavior of a module without making changes to it? The answer in Java is abstraction. You can create abstractions (Java interfaces and abstract classes) that are fixed and yet represent an unbounded group of possible behaviors through concrete subclasses.

Before we write code which follows the Open Closed Principle, let's look at the consequences of violating the Open Closed principle.

Open Closed Principle Violation (Bad Example)

Consider an insurance system that validates health insurance claims before approving one. We can follow the complementary Single Responsibility Principle to model this requirement by creating two separate classes. A HealthInsuranceSurveyor class responsible to validate claims and a ClaimApprovalManager class responsible to approve claims.

HealthInsuranceSurveyor.java

```java
package com.vishal;

public class HealthInsuranceSurveyor{
    public boolean isValidClaim(){
        System.out.println("HealthInsuranceSurveyor: Validating health insurance claim...");
        /*Logic to validate health insurance claims*/
        return true;
    }
}
```

ClaimApprovalManager.java

```java
package com.vishal;

public class ClaimApprovalManager {
    public void processHealthClaim (HealthInsuranceSurveyor surveyor)
    {
        if(surveyor.isValidClaim()){
            System.out.println("ClaimApprovalManager: Valid claim. Currently processing claim for approval....");
        }
    }
}
```

Both the HealthInsuranceSurveyor and ClaimApprovalManager classes work fine and the design for the insurance system appears perfect until a new requirement to process vehicle insurance claims arises. We now need to include a new VehicleInsuranceSurveyor class, and this should not create any problems. But, what we also need is to modify the ClaimApprovalManager class to process vehicle insurance claims. This is how the modified ClaimApprovalManager will be:

Modified ClaimApprovalManager.java

```java
package com.vishal;

public class ModifiedClaimApprovalManager {
    public void processHealthClaim (HealthInsuranceSurveyor surveyor)
    {
        if(surveyor.isValidClaim()){
            System.out.println("ClaimApprovalManager: Valid claim. Currently processing claim for approval....");
        }
    }
    public void processVehicleClaim (VehicleInsuranceSurveyor surveyor)
    {
        if(surveyor.isValidClaim()){
            System.out.println("ClaimApprovalManager: Valid claim. Currently processing claim for approval....");
        }
    }
}
```

In the example above, we modified the ClaimApprovalManager class by adding a new processVehicleClaim( ) method to incorporate a new functionality (claim approval of vehicle insurance).
As apparent, this is a clear violation of the Open Closed Principle. We need to modify the class to add support for a new functionality. In fact, we violated the Open Closed Principle at the very first instance we wrote the ClaimApprovalManager class. This may appear innocuous in the current example, but consider the consequences in an enterprise application that needs to keep pace with fast changing business demands. For each change, you need to modify, test, and deploy the entire application. That not only makes the application fragile and expensive to extend but also makes it prone to software bugs.

Coding to the Open Closed Principle

The ideal approach for the insurance claim example would have been to design the ClaimApprovalManager class in a way that it remains:

- **Open** to support more types of insurance claims.
- **Closed** for any modifications whenever support for a new type of claim is added.

To achieve this, let's introduce a layer of abstraction by creating an abstract class to represent different claim validation behaviors. We will name the class InsuranceSurveyor.

InsuranceSurveyor.java

```java
package com.vishal;

public abstract class InsuranceSurveyor {
    public abstract boolean isValidClaim();
}
```

Next, we will write the specific classes for each type of claim validation.

HealthInsuranceSurveyor.java

```java
package com.vishal;

public class HealthInsuranceSurveyor extends InsuranceSurveyor{
    public boolean isValidClaim(){
        System.out.println("HealthInsuranceSurveyor: Validating health insurance claim...");
        /*Logic to validate health insurance claims*/
        return true;
    }
}
```

VehicleInsuranceSurveyor.java

```java
package com.vishal;

public class VehicleInsuranceSurveyor extends InsuranceSurveyor{
    public boolean isValidClaim(){
       System.out.println("VehicleInsuranceSurveyor: Validating vehicle insurance claim...");
        /*Logic to validate vehicle insurance claims*/
        return true;
    }
}
```

In the examples above, we wrote the HealthInsuranceSurveyor and VehicleInsuranceSurveyor classes that extend the abstract InsuranceSurveyor class. Both classes provide different implementations of the isValidClaim( ) method. We will now write the ClaimApprovalManager class to follow the Open/Closed Principle.

ClaimApprovalManager.java

```java
package com.vishal;

public class ClaimApprovalManager {
    public void processClaim(InsuranceSurveyor surveyor){
        if(surveyor.isValidClaim()){
            System.out.println("ClaimApprovalManager: Valid claim. Currently processing claim for approval....");
        }
    }
}
```

In the example above, we wrote a processClaim( ) method to accept a InsuranceSurveyor type instead of specifying a concrete type. In this way, any further addition of InsuranceSurveyor implementations will not affect the ClaimApprovalManager class. Our insurance system is now **open** to support more types of

insurance claims, and **closed** for any modifications whenever a new claim type is added. To test our example, let's write this unit test.

ClaimApprovalManagerTest.java

```
package guru.springframework.blog.openclosedprinciple;
import org.junit.Test;
import static org.junit.Assert.*;
public class ClaimApprovalManagerTest {
    @Test
    public void testProcessClaim() throws Exception {
      HealthInsuranceSurveyor healthInsuranceSurveyor=new HealthInsuranceSurveyor();
      ClaimApprovalManager claim1=new ClaimApprovalManager();
      claim1.processClaim(healthInsuranceSurveyor);
        VehicleInsuranceSurveyor vehicleInsuranceSurveyor=new VehicleInsuranceSurveyor();
        ClaimApprovalManager claim2=new ClaimApprovalManager();
        claim2.processClaim(vehicleInsuranceSurveyor);
    }
}
```

The output is:

```
.   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.2.3.RELEASE)
Running guru.springframework.blog.openclosedprinciple.ClaimApprovalManagerTest
HealthInsuranceSurveyor: Validating health insurance claim...
ClaimApprovalManager: Valid claim. Currently processing claim for approval....
VehicleInsuranceSurveyor: Validating vehicle insurance claim...
ClaimApprovalManager: Valid claim. Currently processing claim for approval....
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec - in guru.springfram
```

Summary

Most of the times real closure of a software entity is practically not possible because there is always a chance that a change will violate the closure. For example, in our insurance example a change in the business rule to process a specific type of claim will require modifying the ClaimApprovalManager class. So, during enterprise application development, even if you might not always manage to write code that satisfies the Open Closed Principle in every aspect, taking the steps towards it will be beneficial as the application evolves.

Get The Code

I've committed the source code for this post to github. It is a Maven project which you can download and build.

Liskov Substitution Principle

The Liskov Substitution Principle is one of the SOLID principles of object-oriented programming (Single responsibility, Open-closed, Liskov Substitution, Interface Segregation and Dependency Inversion). We have already written about the single responsibility principle, and these five principles combined are used to make object-oriented code more readable, maintainable and easier to upgrade and modify.
Liskov Substitution Principle states the following: *"in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may substitute objects of type T) without altering any of the desirable properties of that program (correctness, task performed, etc.)"*. Simply said, any object of some class in an object-oriented program can be replaced by an object of a child class. In order to understand this principle better, we'll make a small digression to briefly remind ourselves about the concept of inheritance and its properties, as well as subtyping, a form of polymorphism.

Inheritance, Polymorphism, Subtyping

Inheritance is a concept fairly simple to understand – it is when an object or a class are based on another object or class. When a class is "inherited" from another class, it means that the inherited class (also called subclass, or child class) contains all the characteristics of the superclass (parent class), but can also contain new properties. Let's illustrate this with a common example: if you have a class Watch , you can inherit from that class to get a class PocketWatch . A pocket watch is still a watch, it just has some additional features. Another example would be a class called Woman with a child class called Mother. A mother is still a woman, with the addition of having a child. This brings us to the next term we should explain, which is called **polymorphism**: objects can behave in one way in a certain situation, and in another way in some other situation. In object-oriented programming, this is called *context-dependent behavior*. To use the last example: a mother, when taking a walk with her child or attending a school parent's meeting, will behave as a mother. But when she is out with her friends, at work or simply doing errands, she will behave as a woman. (As you can see, this difference is not that strict.)
Subtyping is a concept that is not identical to polymorphism. However, the two are so tightly connected and fused together in common languages like C++, Java and C#, that the difference between them is practically non-existent. We will still give a formal definition of subtyping, though, for the sake of completeness, but the details will not be discussed in this article. *"In programming language theory, subtyping (also subtype polymorphism or inclusion polymorphism) is a form of type polymorphism in which a subtype is a datatype that is related to another datatype (the supertype) by some notion of substitutability, meaning that program elements, typically subroutines or functions, written to operate on elements of the supertype can also operate on elements of the subtype. If S is a subtype of T, the subtyping relation is often written S <: T, to mean that any term of type S can be safely used in a context where a term of type T is expected."*
This brings us to the original theme of the article – the Liskov Substitution Principle.

Liskov Substitution Principle Examples

The Liskov substitution principle, written by Barbara Liskov in 1988, states that functions that reference base classes must be able to use objects of derived (child) classes without knowing it. It's important for a programmer to notice that, unlike some other Gang of Four principles, whose breaking might result in bad, but *working code*, the violation of this principle will most likely lead to buggy or difficult to maintain code.
Let's illustrate this in Java:

```
class TrasportationDevice
{
   String name;
   String getName() { ... }
   void setName(String n) { ... }
   double speed;
   double getSpeed() { ... }
   void setSpeed(double d) { ... }

   Engine engine;
   Engine getEngine() { ... }
   void setEngine(Engine e) { ... }
   void startEngine() { ... }
}
```

```
class Car extends TransportationDevice
{
   @Override
   void startEngine() { ... }
}
```

There is no problem here, right? A car is definitely a transportation device, and here we can see that it overrides the startEngine() method of its superclass.
Let's add another transportation device:

```
class Bicycle extends TransportationDevice
{
   @Override
   void startEngine() /*problem!*/
}
```

Everything isn't going as planned now! Yes, a bicycle **is a** transportation device, however, it does not have an engine and hence, the method startEngine cannot be implemented.
These are the kinds of problems that violation of Liskov Substitution Principle leads to, and they can most usually be recognized by a method that does nothing, or even can't be implemented.
The solution to these problems is a correct **inheritance hierarchy**, and in our case we would solve the problem by differentiating classes of transportation devices with and without engines. Even though a bicycle **is a** transportation device, it doesn't have an engine. In this example our definition of transportation device is wrong. It should not have an engine.
We can refactor our TransportationDevice class as follows:

```
class TrasportationDevice
{
   String name;
   String getName() { ... }
   void setName(String n) { ... }

   double speed;
   double getSpeed() { ... }
   void setSpeed(double d) { ... }
}
```

Now we can extend TransportationDevice for non-motorized devices.

```
class DevicesWithoutEngines extends TransportationDevice
{
   void startMoving() { ... }
}
```

And extend TransportationDevice for motorized devices. Here is is more appropriate to add theEngine object.

```
class DevicesWithEngines extends TransportationDevice
{
   Engine engine;
   Engine getEngine() { ... }
   void setEngine(Engine e) { ... }

   void startEngine() { ... }
}
```

Thus our Car class becomes more specialized, while adhering to the Liskov Substitution Principle.

```
class Car extends DevicesWithEngines
{
   @Override
   void startEngine() { ... }
}
```

And our Bicycle class is also in compliance with the Liskov Substitution Principle.

```
class Bicycle extends DevicesWithoutEngines
{
   @Override
   void startMoving() { ... }
}
```

Conclusion

Object Oriented languages such as Java are very powerful and offer you as a developer a tremendous amount of flexibility. You can misuse or abuse any language. In the Polymorphism post I explained the 'Is-A' test. If you're writing objects which extend classes, but fails the **'Is-A'** test, you're likely violating the Liskov Substitution Principle.

Interface Segregation Principle

Interfaces form a core part of the Java programming language and they are extensively used in enterprise applications to achieve abstraction and to support multiple inheritance of type- the ability of a class to implement more than one interfaces. From coding perspective, writing an interface is simple. You use the interface keyword to create an interface and declare methods in it. Other classes can use that interface with the implements keyword, and then provide implementations of the declared methods. As a Java programmer, you must have written a large number of interfaces, but the critical question is- have you written them while keeping design principles in mind? A design principle to follow while writing interfaces is the Interface Segregation Principle.

The Interface Segregation Principle represents the “I” of the five SOLID principles of object-oriented programming to write well-designed code that are more readable, maintainable, and easier to upgrade and modify. This principle was first used by Robert C. Martin while consulting for Xerox, which he mentioned in his 2002 book, Agile Software Development: Principles, Patterns and Practices. This principle states that “Clients should not be forced to depend on methods that they do not use”. Here, the term “Clients” refers to the implementing classes of an interface.

What the Interface Segregation Principle says is that your interface should not be bloated with methods that implementing classes don’t require. For such interfaces, also called “fat interfaces”, implementing classes are unnecessarily forced to provide implementations (dummy/empty) even for those methods that they don’t need. In addition, the implementing classes are subject to change when the interface changes. An addition of a method or change to a method signature requires modifying all the implementation classes even if some of them don’t use the method.

The Interface Segregation Principle advocates segregating a “fat interface” into smaller and highly cohesive interfaces, known as “role interfaces”. Each “role interface” declares one or more methods for a specific behavior. Thus clients, instead of implementing a “fat interface”, can implement only those “role interfaces” whose methods are relevant to them.

Interface Segregation Principle Violation (Bad Example)

Consider the requirements of an application that builds different types of toys. Each toy will have a price and color. Some toys, such as a toy car or toy train can additionally move, while some toys, such as a toy plane can both move and fly. An interface to define the behaviors of toys is this.

Toy.java

```java
public interface Toy {
    void setPrice(double price);
    void setColor(String color);
    void move();
    void fly();
}
```

A class that represents a toy plane can implement the Toy interface and provide implementations of all the interface methods. But, imagine a class that represents a toy house. This is how the ToyHouse class will look.

ToyHouse.java

```java
public class ToyHouse implements Toy {
    double price;
    String color;
    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public void move(){}
    @Override
    public void fly(){}
}
```

As you can see in the code, ToyHouse needs to provide implementation of the move() and fly() methods, even though it does not require them. This is a violation of the Interface Segregation Principle. Such violations affect code readability and confuse programmers. Imagine that you are writing the ToyHouse class and the intellisense feature of your IDE pops up the fly() method for auto complete. Not exactly the behavior you want for a toy house, is it?

Violation of the Interface Segregation Principle also leads to violation of the complementary Open Closed Principle. As an example, consider that the Toy interface is modified to include a walk( ) method to accommodate toy robots. As a result, you now need to modify all existing Toy implementation classes to include a walk method even if the toys don't walk. In fact, the Toy implementation classes will never be closed for modifications, which will lead to a fragile application that is difficult and expensive to maintain.

Following the Interface Segregation Principle

By following the Interface Segregation Principle, you can address the main problem of the toy building application- *The Toy interface forces clients (implementation classes) to depend on methods that they do not use*.

The solution is- *Segregate the Toy interface into multiple role interfaces each for a specific behavior*. Let's segregate the Toy interface, so that our application now have three interfaces:Toy, Movable, and Flyable.

Toy.java

```
package guru.springframework.blog.interfacesegregationprinciple;
public interface Toy {
     void setPrice(double price);
     void setColor(String color);
}
```

Movable.java

```
package guru.springframework.blog.interfacesegregationprinciple;
public interface Movable {
    void move();
}
```

Flyable.java

```
package guru.springframework.blog.interfacesegregationprinciple;
public interface Flyable {
    void fly();
}
```

In the examples above, we first wrote the Toy interface with the setPrice() and setColor() methods. As all toys will have a price and color, all Toy implementation classes can implement this interface. Then, we wrote the Movable and Flyable interfaces to represent moving and flying behaviors in toys. Let's write the implementation classes.

ToyHouse.java

```java
package guru.springframework.blog.interfacesegregationprinciple;
public class ToyHouse implements Toy {
    double price;
    String color;
    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public String toString(){
        return "ToyHouse: Toy house- Price: "+price+" Color: "+color;
    }
}
```

ToyCar.java

```java
package guru.springframework.blog.interfacesegregationprinciple;
public class ToyCar implements Toy, Movable {
    double price;
    String color;
    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
     this.color=color;
    }
    @Override
    public void move(){
        System.out.println("ToyCar: Start moving car.");
    }
    @Override
    public String toString(){
        return "ToyCar: Moveable Toy car- Price: "+price+" Color: "+color;
    }
}
```

ToyPlane.java

```java
package guru.springframework.blog.interfacesegregationprinciple;
public class ToyPlane implements Toy, Movable, Flyable {
    double price;
    String color;
    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public void move(){
        System.out.println("ToyPlane: Start moving plane.");
    }
    @Override
    public void fly(){
        System.out.println("ToyPlane: Start flying plane.");
    }
    @Override
    public String toString(){
        return "ToyPlane: Moveable and flyable toy plane- Price: "+price+" Color: "+color;
    }
}
```

As you can see, the implementation classes now implement only those interfaces they are interested in. Our classes do not have unnecessary code clutters, are more readable, and lesser prone to modifications due to changes in interface methods.
Next, let’s write a class to create objects of the implementation classes.

ToyBuilder.java

```java
package guru.springframework.blog.interfacesegregationprinciple;
public class ToyBuilder {
    public static ToyHouse buildToyHouse(){
        ToyHouse toyHouse=new ToyHouse();
        toyHouse.setPrice(15.00);
        toyHouse.setColor("green");
        return toyHouse;
        }
    public static ToyCar buildToyCar(){
        ToyCar toyCar=new ToyCar();
        toyCar.setPrice(25.00);
        toyCar.setColor("red");
        toyCar.move();
        return toyCar;
    }
    public static ToyPlane buildToyPlane(){
        ToyPlane toyPlane=new ToyPlane();
        toyPlane.setPrice(125.00);
        toyPlane.setColor("white");
        toyPlane.move();
        toyPlane.fly();
        return toyPlane;
    }
}
```

In the code example above, we wrote the ToyBuilder class with three static methods to create objects of the ToyHouse, ToyCar, and ToyPlane classes. Finally, let's write this unit test to test our example.

ToyBuilderTest.java

```java
package guru.springframework.blog.interfacesegregationprinciple;
import org.junit.Test;
public class ToyBuilderTest {
    @Test
    public void testBuildToyHouse() throws Exception {
    ToyHouse toyHouse=ToyBuilder.buildToyHouse();
    System.out.println(toyHouse);
    }
    @Test
    public void testBuildToyCar() throws Exception {
    ToyCar toyCar=ToyBuilder.buildToyCar();;
        System.out.println(toyCar);
    }
    @Test
    public void testBuildToyPlane() throws Exception {
    ToyPlane toyPlane=ToyBuilder.buildToyPlane();
        System.out.println(toyPlane);
    }
}
```

The output is:

```
.   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.2.3.RELEASE)
Running guru.springframework.blog.interfacesegregationprinciple.ToyBuilderTest
ToyHouse: Toy house- Price: 15.0 Color: green
ToyPlane: Start moving plane.
ToyPlane: Start flying plane.
ToyPlane: Moveable and flyable toy plane- Price: 125.0 Color: white
ToyCar: Start moving car.
ToyCar: Moveable Toy car- Price: 25.0 Color: red
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec - in guru.springfram
```

Summary of Interface Segregation Principle

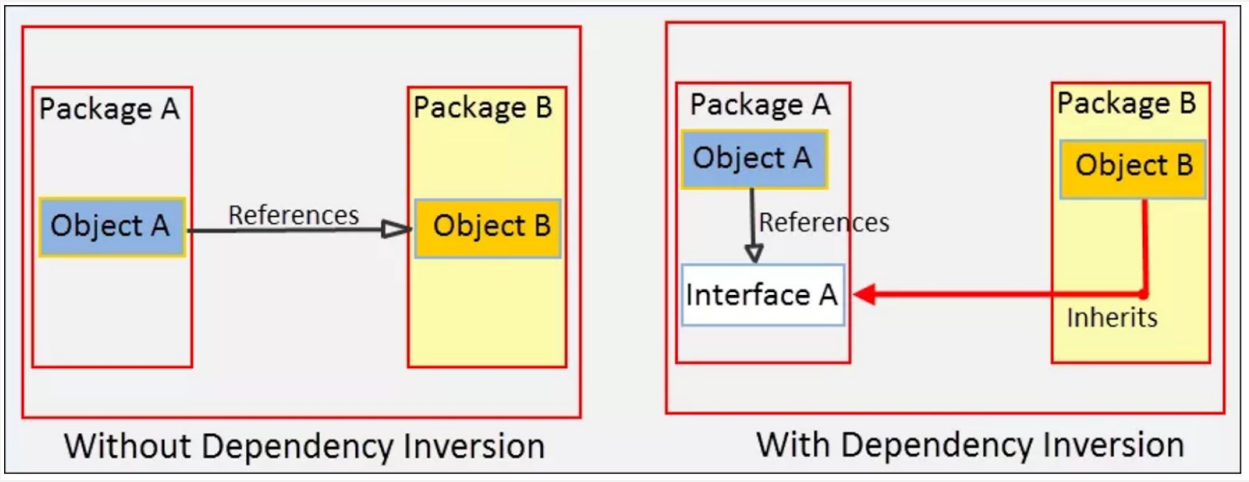
Both the Interface Segregation Principle and Single Responsibility Principle have the same goal: ensuring small, focused, and highly cohesive software components. The difference is that Single Responsibility Principle is concerned with classes, while Interface Segregation Principle is concerned with interfaces. Interface Segregation Principle is easy to understand and simple to follow. But, identifying the distinct interfaces can sometimes be a challenge as careful considerations are required to avoid proliferation of interfaces. Therefore, while writing an interface, consider the possibility of implementation classes having different sets of behaviors, and if so, segregate the interface into multiple interfaces, each having a specific role.

Interface Segregation Principle in the Spring Framework

A number of times on this blog I've written about programming for Dependency Injection when programming using the Spring Framework. The Interface Segregation Principle becomes especially important when doing Enterprise Application Development with the Spring Framework.
As the size and scope of the application you're building grows, you are going to need pluggable components. Even when just for unit testing your classes, the Interface Segregation Principle has a role. If you're testing a class which you've written for dependency injection, as I've written before, it is ideal that you write to an interface. By designing your classes to use dependency injection against an interface, any class implementing the specified interface can be injected into your class. In testing your classes, you may wish to inject a mock object to fulfill the needs of your unit test. But when the class you wrote is running in production, the Spring Framework would inject the real full featured implementation of the interface into your class.
The Interface Segregation Principle and Dependency Injection are two very powerful concepts to master when developing enterprise class applications using the Spring Framework.

Dependency Inversion Principle

As a Java programmer, you've likely heard about code coupling and have been told to avoid tightly coupled code. Ignorance of writing "good code" is the main reason of tightly coupled code existing in applications. As an example, creating an object of a class using the new operator results in a class being tightly coupled to another class. Such coupling appears harmless and does not disrupt small programs. But, as you move into enterprise application development, tightly coupled code can lead to serious adverse consequences.
When one class knows explicitly about the design and implementation of another class, changes to one class raise the risk of breaking the other class. Such changes can have rippling effects across the application making the application fragile. To avoid such problems, you should write "good code" that is loosely coupled, and to support this you can turn to the Dependency Inversion Principle.
The Dependency Inversion Principle represents the last "D" of the five SOLID principles of object-oriented programming. Robert C. Martin first postulated the Dependency Inversion Principle and published it in 1996.
The principle states:
"A. High-level modules should not depend on low-level modules. Both should depend on abstractions.
B. Abstractions should not depend on details. Details should depend on abstractions."
Conventional application architecture follows a top-down design approach where a high-level problem is broken into smaller parts. In other words, the high-level design is described in terms of these smaller parts. As a result, high-level modules that gets written directly depends on the smaller (low-level) modules.
What Dependency Inversion Principle says is that instead of a high-level module depending on a low-level module, both should depend on an abstraction. Let us look at it in the context of Java through this figure.



In the figure above, without Dependency Inversion Principle, Object A in Package A refers Object B in Package B. With Dependency Inversion Principle, an Interface A is introduced as an abstraction in Package A. Object A now refers Interface A and Object B inherits from Interface A. What the principle has done is:

1. Both Object A and Object B now depends on Interface A, the abstraction.
2. It inverted the dependency that existed from Object A to Object B into Object B being dependent on the abstraction (Interface A).

Before we write code that follows the Dependency Inversion Principle, let's examine a typical violating of the principle.

Dependency Inversion Principle Violation (Bad Example)

Consider the example of an electric switch that turns a light bulb on or off. We can model this requirement by creating two classes: ElectricPowerSwitch and LightBulb. Let's write the LightBulb class first.

LightBulb.java

```java
public class LightBulb {
    public void turnOn() {
        System.out.println("LightBulb: Bulb turned on...");
    }
    public void turnOff() {
        System.out.println("LightBulb: Bulb turned off...");
    }
}
```

In the LightBulb class above, we wrote the turnOn() and turnOff() methods to turn a bulb on and off.

Next, we will write the ElectricPowerSwitch class.

ElectricPowerSwitch.java

```java
public class ElectricPowerSwitch {
    public LightBulb lightBulb;
    public boolean on;
    public ElectricPowerSwitch(LightBulb lightBulb) {
        this.lightBulb = lightBulb;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
    public void press(){
        boolean checkOn = isOn();
        if (checkOn) {
            lightBulb.turnOff();
            this.on = false;
        } else {
            lightBulb.turnOn();
            this.on = true;
        }
    }
}
```

In the example above, we wrote the ElectricPowerSwitch class with a field referencing LightBulb. In the constructor, we created a LightBulb object and assigned it to the field. We then wrote a isOn() method that returns the state of ElectricPowerSwitch as a boolean value. In the press() method, based on the state, we called the turnOn() and turnOff() methods.

Our switch is now ready for use to turn on and off the light bulb. But the mistake we did is apparent. Our high-level ElectricPowerSwitch class is directly dependent on the low-level LightBulb class. if you see in the code, the LightBulb class is hardcoded in ElectricPowerSwitch. But, a switch should not be tied to a bulb. It should be able to turn on and off other appliances and devices too, say a fan, an AC, or the entire lightning system of an amusement park. Now, imagine the modifications we will require in the ElectricPowerSwitch class each time we add a new appliance or device. We can conclude that our design is flawed and we need to revisit it by following the Dependency Inversion Principle.

Following the Dependency Inversion Principle

To follow the Dependency Inversion Principle in our example, we will need an abstraction that both the ElectricPowerSwitch and LightBulb classes will depend on. But, before creating it, let’s create an interface for switches.

Switch.java

```
package guru.springframework.blog.dependencyinversionprinciple.highlevel;
public interface Switch {
    boolean isOn();
    void press();
}
```

We wrote an interface for switches with the isOn() and press() methods. This interface will give us the flexibility to plug in other types of switches, say a remote control switch later on, if required. Next, we will write the abstraction in the form of an interface, which we will call Switchable.

Switchable.java

```
package guru.springframework.blog.dependencyinversionprinciple.highlevel;
public interface Switchable {
    void turnOn();
    void turnOff();
}
```

In the example above, we wrote the Switchable interface with the turnOn() and turnoff() methods. From now on, any switchable devices in the application can implement this interface and provide their own functionality. Our ElectricPowerSwitch class will also depend on this interface, as shown below:

ElectricPowerSwitch.java

```java
package guru.springframework.blog.dependencyinversionprinciple.highlevel;
public class ElectricPowerSwitch implements Switch {
    public Switchable client;
    public boolean on;
    public ElectricPowerSwitch(Switchable client) {
        this.client = client;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
   public void press(){
       boolean checkOn = isOn();
       if (checkOn) {
           client.turnOff();
           this.on = false;
       } else {
             client.turnOn();
             this.on = true;
       }
   }
}
```

In the ElectricPowerSwitch class we implemented the Switch interface and referred the Switchable interface instead of any concrete class in a field. We then called the turnOn() and turnoff() methods on the interface, which at run time will get invoked on the object passed to the constructor. Now, we can add low-level switchable classes without worrying about modifying the ElectricPowerSwitch class. We will add two such classes: LightBulb and Fan.

LightBulb.java

```java
package guru.springframework.blog.dependencyinversionprinciple.lowlevel;
import guru.springframework.blog.dependencyinversionprinciple.highlevel.Switchable;
public class LightBulb implements Switchable {
    @Override
    public void turnOn() {
        System.out.println("LightBulb: Bulb turned on...");
    }
    @Override
    public void turnOff() {
        System.out.println("LightBulb: Bulb turned off...");
    }
}
```

Fan.java

```java
package guru.springframework.blog.dependencyinversionprinciple.lowlevel;
import guru.springframework.blog.dependencyinversionprinciple.highlevel.Switchable;
public class Fan implements Switchable {
    @Override
    public void turnOn() {
        System.out.println("Fan: Fan turned on...");
    }
    @Override
    public void turnOff() {
        System.out.println("Fan: Fan turned off...");
    }
}
```

In both the LightBulb and Fan classes that we wrote, we implemented the Switchable interface to provide their own functionality for turning on and off. While writing the classes, if you have missed how we arranged them in packages, notice that we kept the Switchable interface in a different package from the low-level electric device classes. Although, this did not make any difference from coding perspective, except for an import statement, by doing so we have made our intentions clear- We want the low-level classes to depend (inversely) on our abstraction. This will also help us if we later decide to release the high-level package as a public API that other

applications can use for their devices. To test our example, let's write this unit test.

ElectricPowerSwitchTest.java

```java
package guru.springframework.blog.dependencyinversionprinciple.highlevel;
import guru.springframework.blog.dependencyinversionprinciple.lowlevel.Fan;
import guru.springframework.blog.dependencyinversionprinciple.lowlevel.LightBulb;
import org.junit.Test;
public class ElectricPowerSwitchTest {
    @Test
    public void testPress() throws Exception {
     Switchable switchableBulb=new LightBulb();
     Switch bulbPowerSwitch=new ElectricPowerSwitch(switchableBulb);
     bulbPowerSwitch.press();
     bulbPowerSwitch.press();
    Switchable switchableFan=new Fan();
    Switch fanPowerSwitch=new ElectricPowerSwitch(switchableFan);
    fanPowerSwitch.press();
    fanPowerSwitch.press();
    }
}
```

The output is:

```
.   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.2.3.RELEASE)
Running guru.springframework.blog.dependencyinversionprinciple.highlevel.ElectricPowerSwitchTe
LightBulb: Bulb turned on...
LightBulb: Bulb turned off...
Fan: Fan turned on...
Fan: Fan turned off...
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.016 sec - in guru.springfram
```

Summary of the Dependency Inversion Principle

Robert Martin equated the Dependency Inversion Principle, as a first-class combination of the Open Closed Principle and the Liskov Substitution Principle, and found it important enough to give its own name. While using the Dependency Inversion Principle comes with the overhead of writing additional code, the advantages that it provides outweigh the extra effort. Therefore, from now whenever you start writing code, consider the possibility of dependencies breaking your code, and if so, add abstractions to make your code resilient to changes.

Dependency Inversion Principle and the Spring Framework

You may think the Dependency Inversion Principle is related to Dependency Injection as it applies to the Spring Framework, and you would be correct. Uncle Bob Martin coined this concept of Dependency Inversion before Martin Fowler coined the term Dependency Injection. The two concepts are highly related. Dependency Inversion is more focused on the structure of your code, its focus is keeping your code loosely coupled. On the other hand, Dependency Injection is how the code functionally works. When programming with the Spring Framework, Spring is using Dependency Injection to assemble your application. Dependency Inversion is what decouples your code so Spring can use Dependency Injection at run time.