# Single Responsibility Principle

#### Object-Oriented Terminology

In object-oriented programming (Java, among other languages, follows this paradigm), you will often hear terms such as robustness, cohesion, coupling etc. [**Cohesion**](https://en.wikipedia.org/wiki/Cohesion_%28computer_science%29) 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**](https://en.wikipedia.org/wiki/Coupling_%28computer_programming%29), 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.

#### 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](http://springframework.guru/dependency-injection-example-using-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

Section 3, Lecture 43

#### 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](http://springframework.guru/solid-principles-object-oriented-programming/) software engineering principles to write well-designed code that are more readable, maintainable, and easier to upgrade and modify. [Bertrand Meyer](https://en.wikipedia.org/wiki/Bertrand_Meyer) coined the term Open Closed Principle, which first appeared in his book [Object-Oriented Software Construction](https://www.amazon.com/Object-Oriented-Software-Construction-CD-ROM-Edition/dp/0136291554), 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](http://springframework.guru/polymorphism-java/) 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](http://springframework.guru/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

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

#### ClaimApprovalManager.java

* package guru.springframework.blog.openclosedprinciple;
* 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

* package guru.springframework.blog.openclosedprinciple;
* public class ClaimApprovalManager {
* 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

* package guru.springframework.blog.openclosedprinciple;
* public abstract class InsuranceSurveyor {
* public abstract boolean isValidClaim();
* }

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

#### HealthInsuranceSurveyor.java

* package guru.springframework.blog.openclosedprinciple;
* 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

* package guru.springframework.blog.openclosedprinciple;
* 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

* package guru.springframework.blog.openclosedprinciple;
* 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.springframework.blog.openclosedprinciple.ClaimApprovalManagerTest

#### 

#### 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.

# Liskov Substitution Principle

Section 3, Lecture 44

#### Liskov Substitution Principle

The Liskov Substitution Principle is one of the SOLID principles of object-oriented programming ([Single responsibility](http://springframework.guru/single-responsibility-principle/), [Open-closed](http://springframework.guru/open-closed-principle/), Liskov Substitution, [Interface Segregation](http://springframework.guru/interface-segregation-principle/) and Dependency Inversion). We have already written about the [single responsibility principle](http://springframework.guru/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**](http://springframework.guru/polymorphism-java/): 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](http://springframework.guru/polymorphism-java/). 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](https://en.wikipedia.org/wiki/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](http://springframework.guru/gang-of-four-design-patterns/) 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](http://springframework.guru/polymorphism-java/) 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.