|  |  |
| --- | --- |
| **Principle** | **Examples** |
| Single Responsibility Principle | <https://github.com/htuzel/SolidPrinciples/blob/master/1.SingleResponsibility/after.js> |
| Open / Closed Principle | <https://github.com/htuzel/SolidPrinciples/blob/master/2.OpenClose/after.ts> |
| Liskov Substitution Principle | <https://github.com/htuzel/SolidPrinciples/blob/master/3.LiskovSubstitution/after.js> |
| Interface Segregation Principle | <https://github.com/htuzel/SolidPrinciples/blob/master/4.InterfaceSegregation/after.ts> |
| Dependency Inversion Principle | <https://github.com/htuzel/SolidPrinciples/blob/master/5.DependencyInversion/after.js> |

**SOLID “S” violation**



In this code snippet it is violating the SRP because there are three functions responsible for three different actors. To fix this problem is to separate code that support different actors.

* Create three different methods for calculatePay, reportHours and save, that receive as parameter the employee data.

The employData object to save a shared simple data structure, it used by all three actors.



PayCalculator



HourReporter



EmployeeServer



**SOLID “O” violation**

Text

Description automatically generated

In this code snippet is violating the Open / Close principle, because if we need a new role, it should modify the method. To fix it, the following change can be done.

Text

Description automatically generated

In this case there are two methods the first one is in charge to check if the employee has the privilege and the second one is in charge to add a new role. So, the problem was fixed. Because if we need a new role the only thing that is need it is call the method addNewRole.

**SOLID “L” violation**

The Liskov Substitution Principle says, if we have a base class, it should not extend the method to classes that cannot or won’t accept its parent class methods. The following code snippet is an example of what not to do.

Text

Description automatically generated

The problem here is that we have base class Bird and as we know there are birds can fly and others can not fly, so if we use the base class Bird in a bird that can not fly, would be a method not related to the class. The following code snippet explain the solution to this situation.

Text

Description automatically generated

Now we have a base class and other subclasses, for every bird.

**SOLID “I” violation**

The Interface Segregation Principle says that no client should be forced to depend on method it does not use.

In the following code snippet, there are a lot of methods and attributes that may not be used in a class, so, to fix it we must create other interfaces with the methods and attributes that are related.

Text

Description automatically generated

Example of the solution:

Text

Description automatically generated

**SOLID “D” violation**

To understand the Dependency Inversion Principle there are two keys:

1. High-Level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions.

This means that an abstraction (interface or abstract class) should not depend on a detail (concrete classes).

The goal of the DIP is to decouple high-level modules from low-level modules. This safeguards the higher-level modules from possibly breaking changes in lower-level modules.

Example:

Text

Description automatically generated

In this example we use a constructor function to provide a CourseService to our CourseController class. The CourseService then gets used in the get method of the CourseController.

This means that a high-level module depends on a low-level module.

This a problem because, a change in the low-level module (CourseService) could break modules that depend on it.

To fix it we must create an interface.

Text

Description automatically generated

We changed the CourseController in such a way that it only refers to an abstraction of the CourseService, the interface ICourseService, not a concrete class.