# Liskov Substitution Principle

FUNCTIONS THAT USE POINTERS OR REFERENCES TO BASE CLASSES MUST BE ABLE TO USE OBJECTS OF DERIVED CLASSES WITHOUT KNOWING IT.

liskov-substitution-before is an example of a simple shape drawer. The main program simple outputs what shape is being drawn.

Example one…

ShapeProcesser class violates the solid principles in that the DrawShape function is badly formed. It must know about every possible Shape class, and it must be changed whenever new shapes are created.

Square inherits from rectangle as its fair to say in all intents and purposes that a Square is a rectangle. Can lead to subtle problem.

A Square does not need the Rectangle height and width member variables and the setWidth and SetHeight methods as the width and height of a square are identical.

Example two…

Programmer creates the method create which takes a Rectangle sets the height and width and assumes that the result is 20.

This is fine if the RectangleBase is a Rectangle but if a Square is passed to the method this is not true.

Is the programmer justified in thinking that changing the width of the rectangle will leave the height unchanged?

The Liskov substitution principle highlights these issues and is mainly behaviour driven. A square can be thought of as a Rectangle but a Square is not a rectangle as it’s behaviour is not consistant with that of a Rectangle.

# Interface Segregation

Clients should not have to implement methods that it does not use. Large interfaces into smaller interfaces. ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy.

interface-segregation-before can see that the implementation of the FreePermit class has to implement the Cost method, although the permit is free. This is pollution of interfaces.

# Modify the project to break the interface so that the FreePermit does not have to implement the Cost method.

interface-segregation-after shows splitting the original IPermit interface into a IPermit and ICostPermit. Permit can implement IPermit and ICostPermit and FreePermit just needs to implement IPermit and not provide an implementation for the Cost method.

# Dependency Inversion

https://lostechies.com/derickbailey/2011/09/22/dependency-injection-is-not-the-same-as-the-dependency-inversion-principle/

Decoupling of software modules

Not necessarily the same as dependency injection.

High-level modules should not depend on low-level modules. Both should depend on abstractions.

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

You want to ensure that you can replace the implementation without violating the expectations of that interface, according to LSP

The same principle also applies in software development. Rather than working with a set of classes that are hard wired (tightly coupled) to each other, you want to work with a standard interface.

Code should only work with the interface and not assume any knowledge about the concreate implementation of the interface.

dependency-inversion-before uses the example of the code of the single-responsibility-after code, i.e. the adding a permit class. You can see that the Permits class instantiates the Message sender class when a new Permits is created. This means that to change the contract/implementation of the Message sender we would also need to change the Permits class.

# Using dependency inversion principles modify the project so that the MessageSender class implementation is injected into the Permits class.

dependency-inversion-after shows a new interface IMessageSender which the MessageSender class now implements. The Permits class has a private field IMessageSender that is injected in the constructor rather than instantiating a concreate implementation.

The main program instantiates the MessageSender class and injects into the Permits class.

# Single Responsibility

A class should only have a single responsibility; it should only have one reason to change.

* Robost, changes to a single class or method do not require changes and testing to knock on code.
* Regression

In Sfw.Academy.Solid.SingleResponsibility.Before the Permits class is responsible for adding a permit to the list of permits and to then creating and sending an SMTP email using a defined exchange server. In this example the only reason that the Permits class should change is if the method of adding a permit to the data source requires changing it should not need changing if the email exchange server or method of email sending needs updating.

Change the App.Config to point to your own email address.

# Attempt to modify the project so that modifying the message sending capability is completely separate from the task of adding a permit to the data source.

# Open Closed

“software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification”.

* Regression
* Doesn’t have to be changed each time a requirement changes

Sfw.Academy.Solid.OpenClosed.Before project provides a simple example of a Permit cost calculator, it takes a number of permits calculates the cost and outputs the total cost to the screen. There are two permit types, one that’s cost is calculated by Quantity \* Factor and one by Quantity + Factor.

Looking at the code for the PermitCostCalculator you can see that if we need in the future to add a new permit type that calculates its cost by Quantity – Factor, we will have to add an additional IF statement, this means we are having to modify the PermitCostCalculator class each time a new permit type requires implementing. This is not ideal.

# Attempt to implement PermitTypeThree which calculates the cost of the permit by Quantity –Factor without adding a new IF statement to the PermitCostCalculator class.

We can solve this issue by creating an abstract Permit base class.

The Sfw.Academy.Solid.OpenClosed.After project displays the implementation of the PermitCostCalculator using the Open Closed principle. We can add any number of Permit types without changing the way the total cost of the permits is calculated.

Uncle bob videos

building code that is maintainable and reusable.

The primary mechanisms behind the Open-Closed principle are abstraction and poly-

morphism. In

inheritance. It is by using inheritance that we can

create derived classes that conform to the abstract polymorphic interfaces defined by pure

virtual functions in abstract base classes.