# 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

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

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.

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.