**SOLID**

Single responsibility

A class should have only one reason to change. They must all have one responsibility!

*Poor example*

class Book {

getTitle()

getAuthor()

turnPage()

printCurrentPage() // This should not belong here

}

*Good example*

Class Book {

getTitle()

getAuthor()

turnPage()

getCurrentPage()

}

Interface Printer {

printPage(page)

}

Class HtmlPrinter implements Printer {

printPage(page)

}

Class PlanTextPrinter implements Printer {

printPage(page)

}

Open closed

Open for extension but closed for modification. This basically means we should write code that doesn’t have to be changed every time the requirements change.

*Poor example*

Class Rectangle {

setWidth(width)

setHeight(height)

}

Class Circle {

setRadius(radius)

}

Class AreaCalculator {

getArea(shape) {

if (shape is Rectangle) {

….

} else if (shape is Circle) {

….

}

}

}

*Good example*

Interface Shape {

getArea()

}

Class Rectangle implements Shape {

setWidth(width)

setHeight(height)

getArea()

}

Class Circle implements Shape {

setRadius(radius)

getArea()

}

Liskov substitution

Subtypes must substitutable for their base types. Don’t just inherit from a class because it shares some common methods and saves you by re-using code.

*Poor example*

Class Rectangle {

setWidth(width)

setHeight(height)

}

Class Square extends Rectangle {

setWidth(width, height) // override to set width and height

setHeight(width, height) // override to set width and height

}

// if the above weren’t overridden, then you would be able to call Square::setWidth by inheriting from the Rectangle class

*Good example*

Class Rectangle {

setWidth(width)

setHeight(height)

}

Class Shape {

setWidth(width)

}

Interface segregation

The dependency of one class to another one should depend on the smallest possible interface. Clients should not be forced to implement interfaces they don’t use. Instead of one massive interface, many smaller interfaces are preferred.

*Poor example*

Interface Animal {

Feed()

Groom()

}

Class Dog implements Animal {

Feed() // implementation

Groom() // implementation

}

Class Tiger implements Animal {

Feed() // implementation

Groom() // Dummy implementation to not violate interface

}

*Good example*

Interface Animal {

Feed()

}

Interface Pet extends Animal {

Groom()

}

Class Dog implements Pet {

Feed()

Groom()

}

Class Tiger implements Animal {

Feed()

}

Dependency inversion

Depend upon abstractions (interfaces), rather than concrete classes. High level modules should not depend on low level modules, but should depend upon abstractions.

*Poor example*

Function print (printer printer) {

If (printer.getType() === ‘html’) {

…

} else …

}

*Good example*

Function print (printerInterface printer) {

//print

}