**Architecture**

**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

}

**REP, CCP & CRP**

Components are units of *deployment*. They are the smallest entities that can be deployed as part of a system.

The Reuse/Release Equivalence Principle

People who want to reuse software cannot, and will not, do so unless those components are tracked through a release process and are given release numbers. This reflects that software developers need to know when new releases are coming, and which changes those new releases will bring.

This principle means that the classes and modules that are formed into a component must belong to a cohesive group. The component cannot simply consist of a random bunch of classes and modules; instead there must be some overarching theme that those modules all share.

The Common Closure Principle

Gather into components those classes that change for the same reasons and at the same times. Separate into different components those classes that change at different times and for different reasons. If two classes are so tightly bound, either physically or conceptually, that they always change together, then they belong in the same component. Thus, when a change in requirements comes along, that change has a good chance of being restricted to a minimal number of components.

The Common Reuse Principle

Don’t force users of a component to depend on things they don’t need to. Classes are reused in isolation. More typically, reusable classes collaborate with other classes that are part of the reusable abstraction. The CRP states that these classes belong together in the same component. In such a component we would expect to see classes that have lots of dependencies on each other.

When one component uses another, a dependency is created between the components. Perhaps the *using* component uses only one class within the *used* component, but that still doesn’t weaken the dependency. The *using* component still depends on the *used* component. Because of the dependency, every time the *used* component is changed, the *using* component will likely need corresponding changes. This is true even if the *using* component doesn’t care about the change made in the *used* component. Thus, when we depend on a component, we want to make sure we depend on every class in that component. Put another way we want to make sure that the classes we put in to a component are inseparable – that it is impossible to depend on some and not others.

Therefore, the CRP tells us more about which classes shouldn’t be together than about this classes should be together. The CRP says that classes that are not tightly bound to each other should not be in the same component.

Cohesion vs coupling

Cohesion refers to what the class (or component) can do. Low cohesion would mean that the class does a great variety of actions – it is broad and unfocused on what it should do. High cohesion means that the class is focused on what it should be doing.

|  |
| --- |
| **Document** |
| UploadDocument()  validateDocTitle()  sanitizeData()  checkFileExists() |

|  |
| --- |
| **Document** |
| uploadDocument()  getDocument()  deleteDocument()  checkDocumentExists() |

*Low cohesion high cohesion*

Coupling refers to how related two classes/components are to each other. For low couple classes, changing something in one class should not affect the other. High coupling would make it difficult to change and maintain your code; since classes are closely knit together.

*Good software design has high cohesion and low coupling.*