

SOLID PRInciple

Manish Khadria



February 15, 2023

Khadrias

Bangalore

Contents

[1. Code Smells 2](#_Toc127976312)

[1.1. Rigidity 2](#_Toc127976313)

[1.2. Fragility 2](#_Toc127976314)

[1.3. Immobility 2](#_Toc127976315)

[1.4. Viscosity 2](#_Toc127976316)

[1.5. Needless Complexity 2](#_Toc127976317)

[1.6. Needless Repetition 2](#_Toc127976318)

[1.7. Opacity 2](#_Toc127976319)

[2. EMS (Employee Management System) 2](#_Toc127976320)

[3. SRP (Single Responsibility Principle) 2](#_Toc127976321)

[3.1. Cohesion 2](#_Toc127976322)

[3.2. Coupling 3](#_Toc127976323)

[4. OCP 3](#_Toc127976324)

[5. LSP 3](#_Toc127976325)

[6. ISP 3](#_Toc127976326)

[7. DIP 3](#_Toc127976327)

# Code Smells

Due to requirement changes initial project design degrades. Design failing due to constant changes.

## Rigidity

The system is hard to change because every change forces lot many other changes to the other part.

More Modules Changes = More Rigid Design

## Fragility

Single change break in many places that have no conceptual relationship to the part that was changed.

## Immobility

Hard to reuse the components, due to this keep on duplicate the codes that codebase goes bigger and slower.

## Viscosity

More than one way to do the changes but not doing right changes because of any reason like it impact a lot of places and time consuming so we do hack that is wrong.

It is hard to do the right thing, doing things right is harder than doing things wrongs.

## Needless Complexity

Extra coding for future perspective is good but if that is not useable in future increase complexity by carry extra unnecessary code.

## Needless Repetition

Copy and paste without think is causing a lot of redundant code. We should look for abstraction.

## Opacity

It is hard to read and understand. It does not express its intent well (why it was written the actual requirement) by refactor the code and make it readable and understandable.

# EMS (Employee Management System)

requirements

* + Different Type of Employees.
  + Calculate their salary, bonus and financial component.
  + Log for audit and troubleshooting.
  + Should be extensible, flexible and agile.

# SOLID

All principles are closely connected and interdependent. They are most effective when they combined together.

SOLIC principles complement each other and work together in unison, to achieve the common purpose of well-designed software.

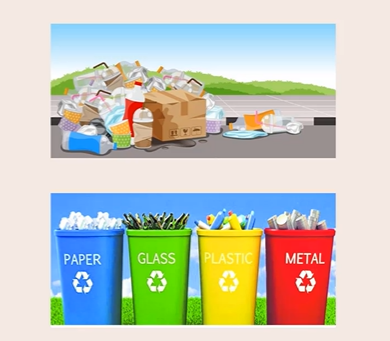
# SRP (Single Responsibility Principle)

Every software component should have one and only one responsibility (reason to change).

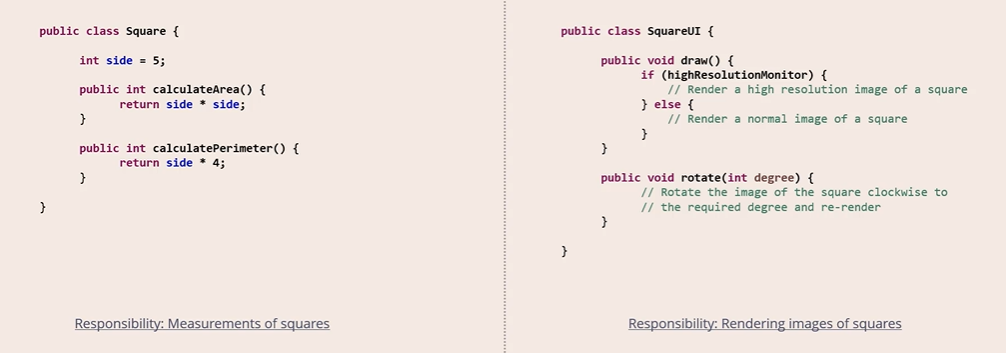
Component can be a class, a method or a module.

## Cohesion

Is the degree to which the various parts of a software component are related.



**Higher Cohesion helps achieve better adherence to the SRP.**



## Coupling

Is defined as the level of inter dependency between various software components.

**Loose Coupling helps achieve better adherence to the SRP.**

****

# OCP (Open Closed Principle)

Software component should be closed for modification but open for extension.

Software component should be extendable to add new feature or behaviour but without Modifying existing code. Sometimes we have redesign or revamp component as well.

Very easy to add new features. And required minimum development and testing as only new feature is required to test only as it is not changing to existing code and features.

OCP mostly required decoupling, which automatically follow SRP.

Do not follow blindly OCP sometime for resolving bug, it is better to made existing code changes over the revamp code design. if only if same type of bug occurrence then go with revamp if it is suitable otherwise we will have a lot of classes that complicate overall design.

It is subjective decision rather than objective to apply OCP.

# LSP (Liskov Substitution Principle)

Object should be replaceable with their subtypes without affecting the correctness of the program.

Car is interface and hatchback is a car implementation. But racing car does not fully implement car interface so it not **replaceable**. as racing car does not have cabin as it has cockpit only. So better create new class Vehicle who is parent of both car and racing car. By rule **Break the hierarchy** if it fails the substitution test.

Second Rule is ‘**Tell, Don’t Ask’**

# ISP (Interface Segregation Principle)

No client should not be forced to depend on methods it does not use.

Avoid below because they lead to violation of ISP  
Fat interfaces

Interfaces with low cohesion

Empty Method Implementation

# DIP (Dependency Inversion Principle)

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

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