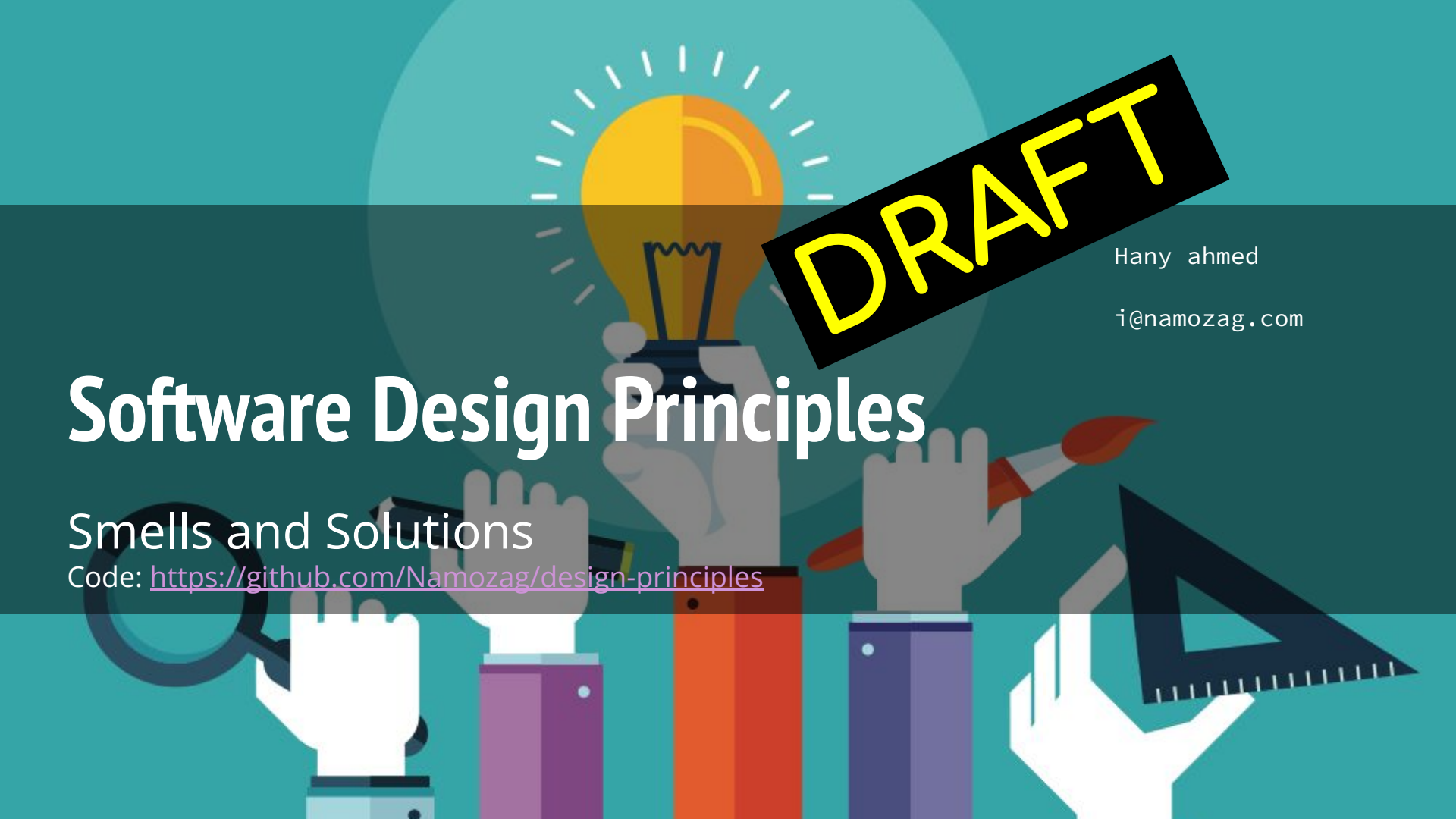


DRAFT

Hany ahmed

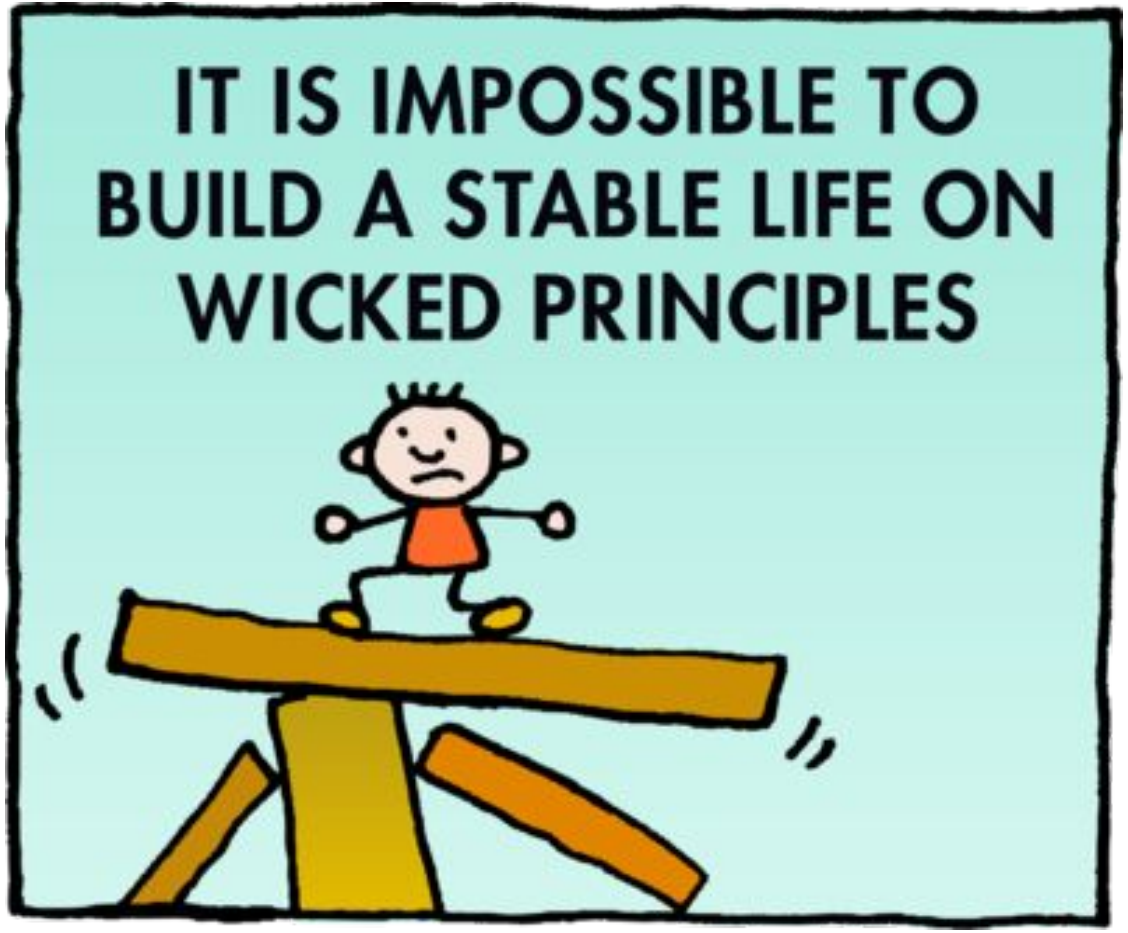
i@namozag.com

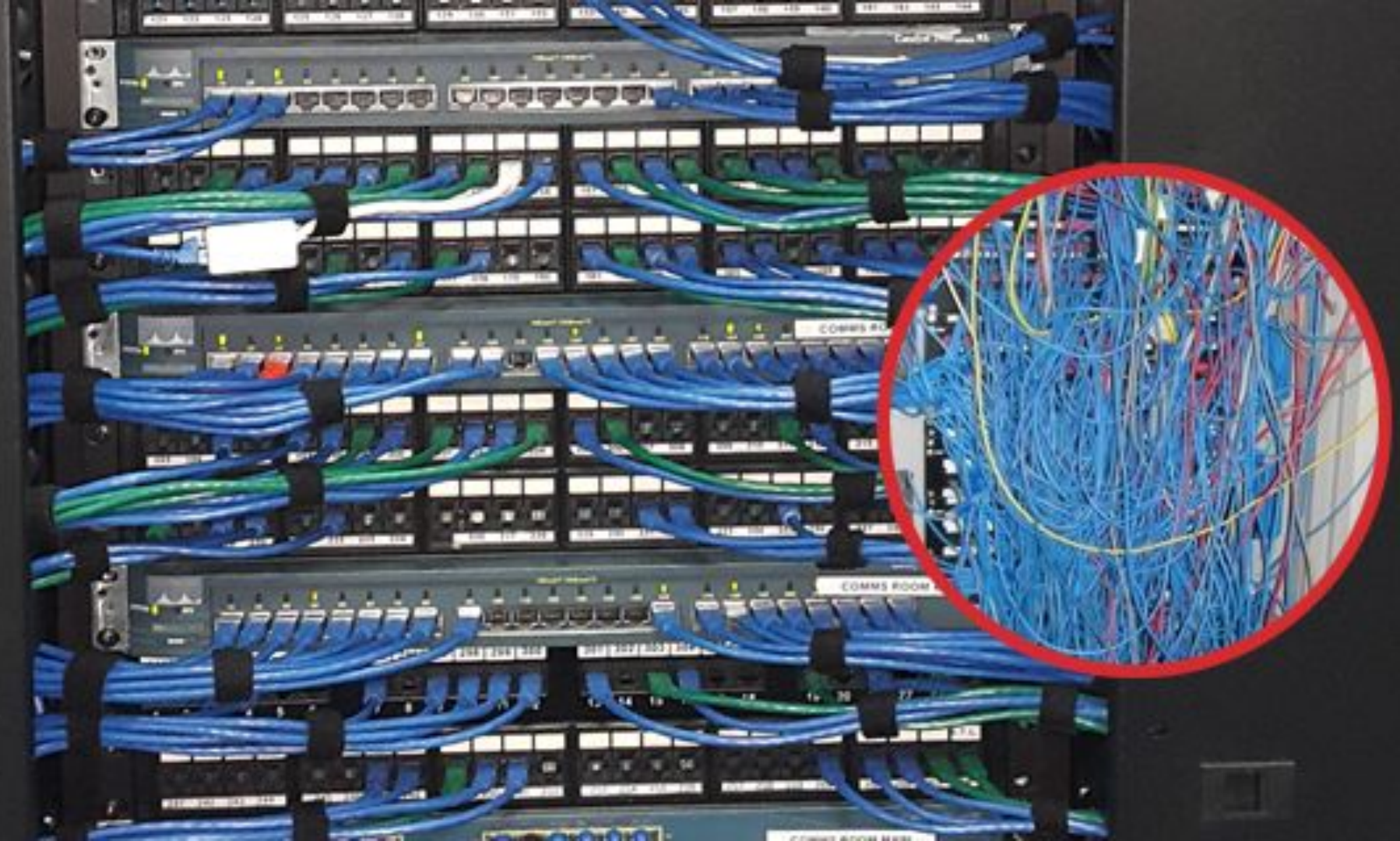
# Software Design Principles

## Smells and Solutions

Code: https://github.com/Namozag/design-principles

# Introduction

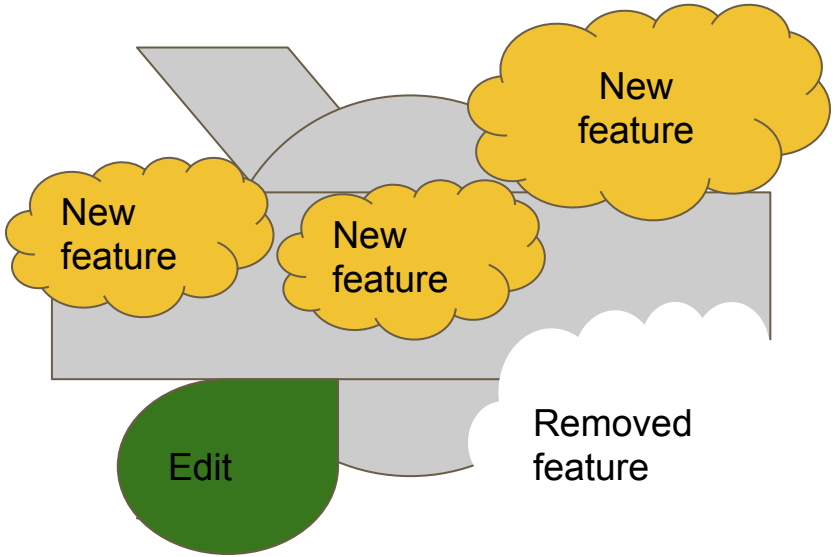

IT IS IMPOSSIBLE TO BUILD A STABLE LIFE ON WICKED PRINCIPLES


COMMS ROOM

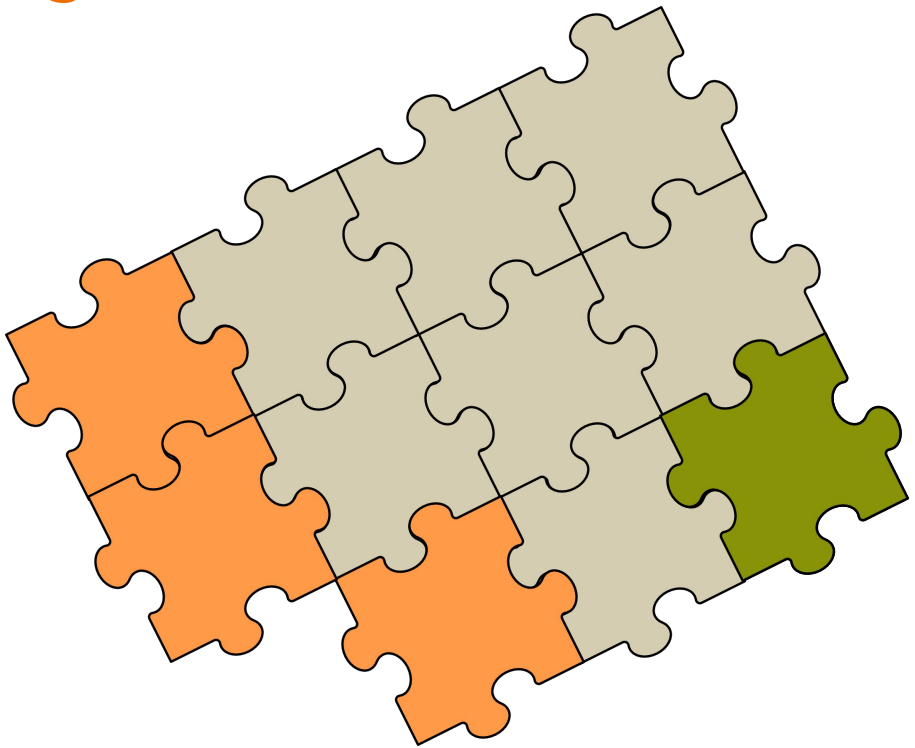
# What are Design principles ?

Set of guidelines that helps us to avoid having a bad design

# What makes the application become ugly overtime ?



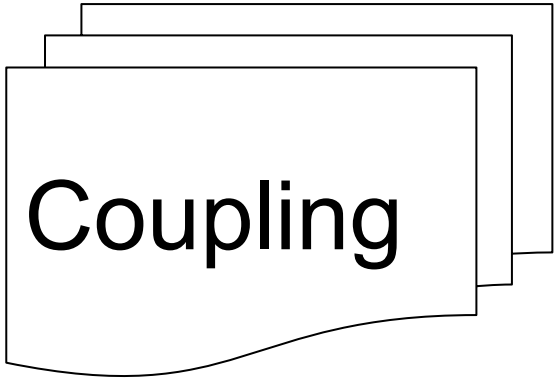
# Changes in a good design



# Why changes make my application become ugly?

Bad design!

# What makes my design bad ?



# Dependencies

# Dependencies

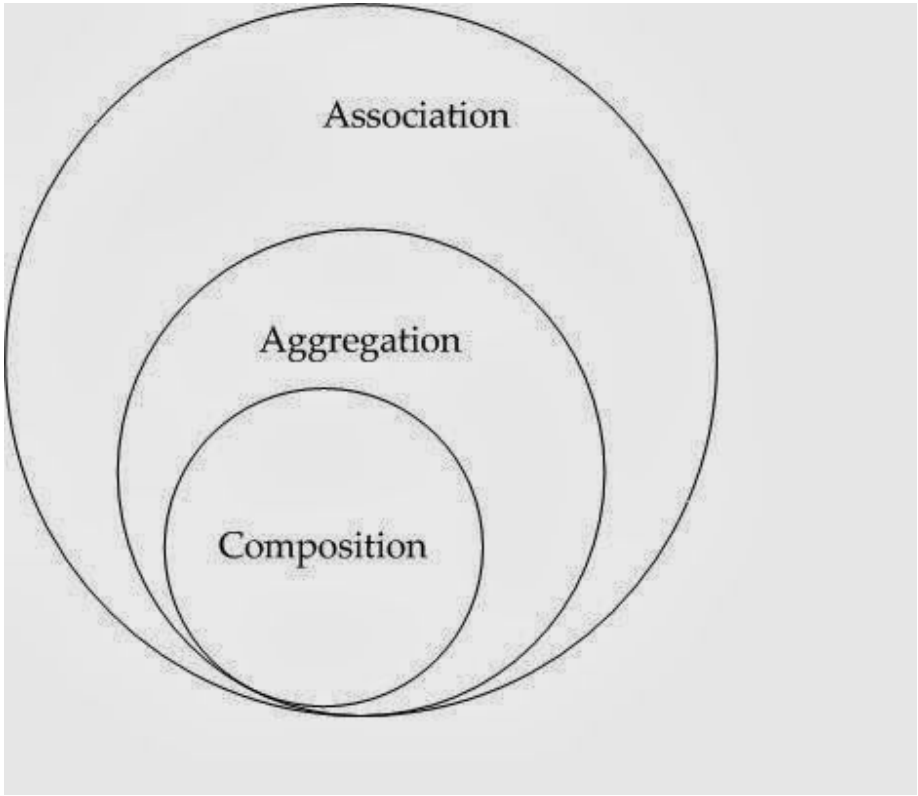
Association

Aggregation

Composition

Realization

Inheritance



# Association

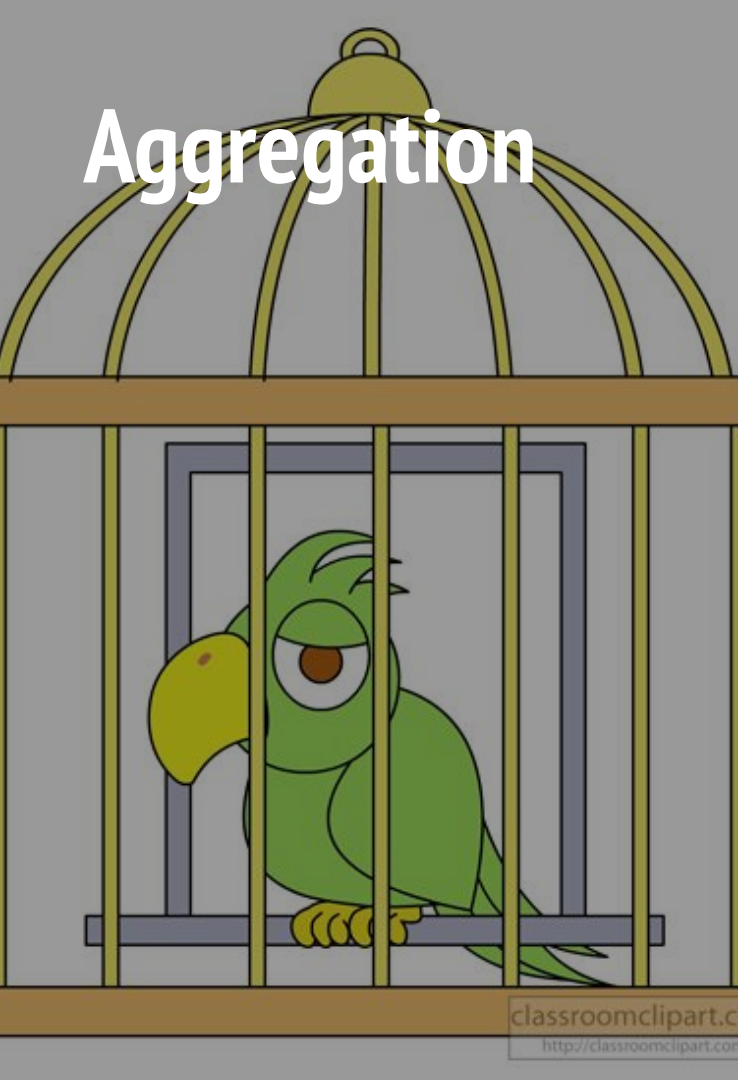

All objects have their own lifecycle

No owner

Implemented with a reference instance variable or a method argument

|Student|---------->|Teacher|

# Aggregation




Whole/part relationship

All objects have their own lifecycle

There is ownership

Instances cannot have cyclic aggregation relationships (part cannot contain its whole)

has a- relationship

**|Cage|<>-------->|Bird|**

# Composition

A strong type of Aggregation.

Child object does not have his own lifecycle

If parent object deleted, child object will also be deleted

part of a- relationship

**|Person|<#>-------->|Head|**

# Symptoms of Rotting Design

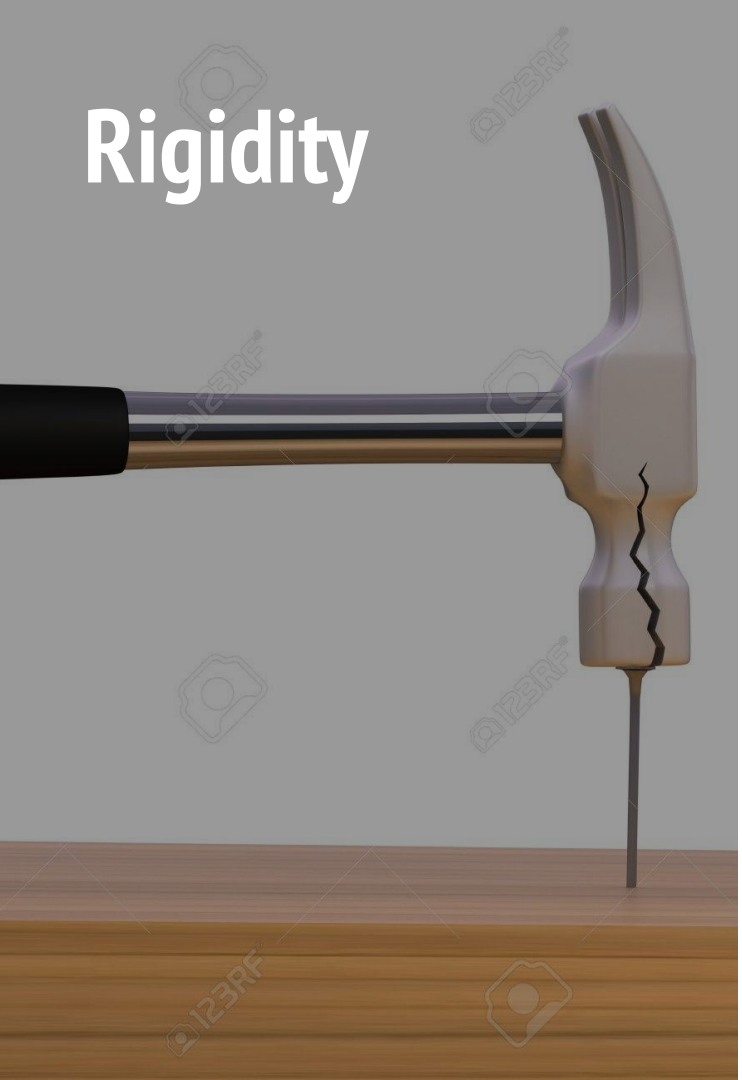
# Symptoms of Rotting Design

Rigidity

Fragility

Immobility

Viscosity




***The tendency for software to be difficult to change***

Every change causes a cascade of subsequent changes in dependent modules

When the manager refuses to allow changes, official rigidity sets in

# Fragility


***The tendency of the software to break in many places every time it is changed***

The breakage may occur in areas that have no conceptual relationship with the area that was changed!

Every fix makes it worse, introducing more problems than are solved

# Immobility

***The inability to reuse software from other projects or from parts of the same project***

The module has too much baggage that it depends upon.

The software is simply rewritten instead of reused

# Viscosity of the Design

***When the design preserving methods are harder to employ than the hacks***

It is easy to do the wrong thing, but hard to do the right thing.

# Viscosity of the Environment

***When the environment make it hard to employ the design***

If compile times are very long, engineers will be tempted to make changes that don't force large recompiles

If the source code control system requires hours to check in just a few files, then engineers will be tempted to make changes that require as few check-ins as possible

# OO Principles

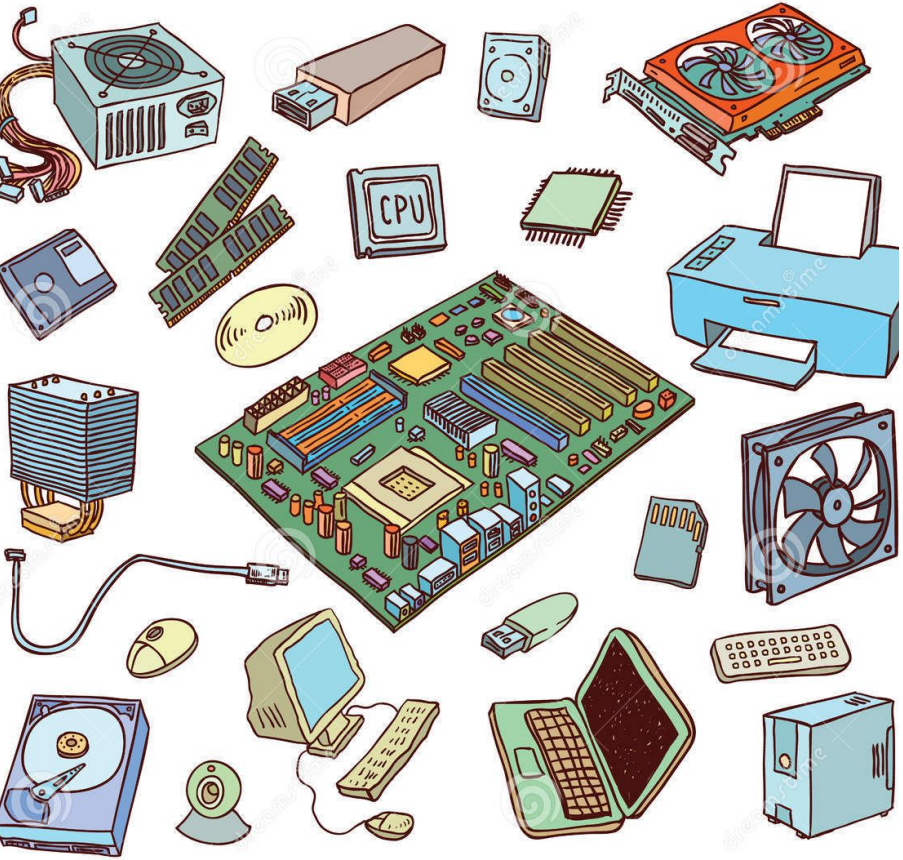
# Object-Oriented Design Principles

Separation of concerns

High Cohesion

Low Coupling

Encapsulation



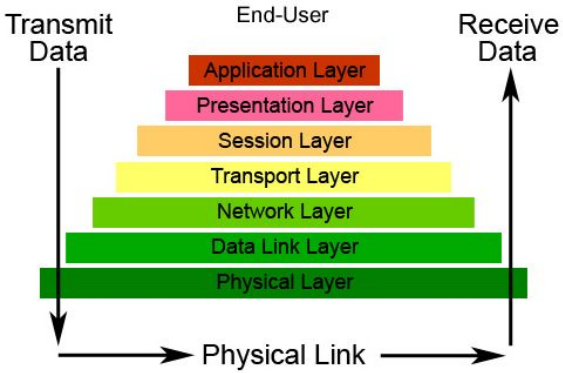


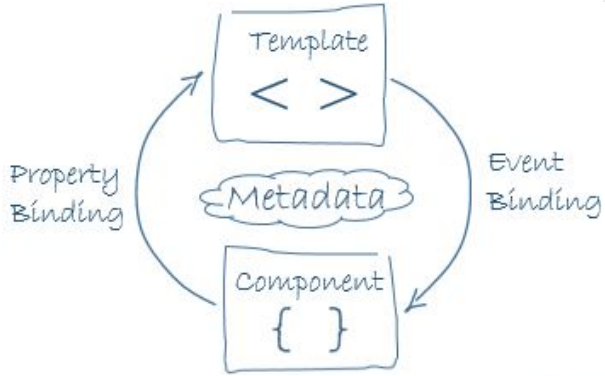

# Separation of concerns

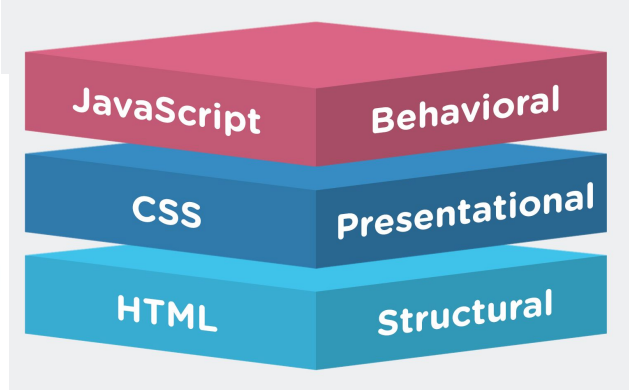
Minimization of interaction points to achieve

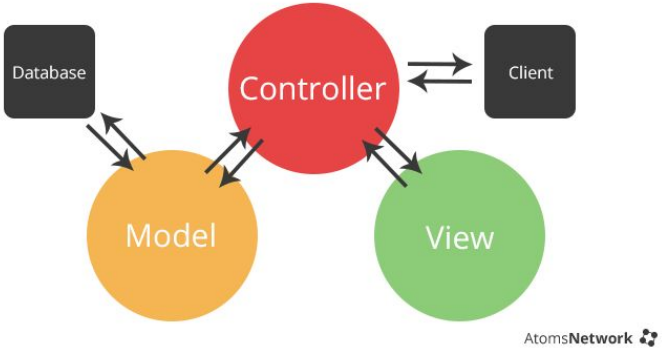
- high cohesion
- low coupling

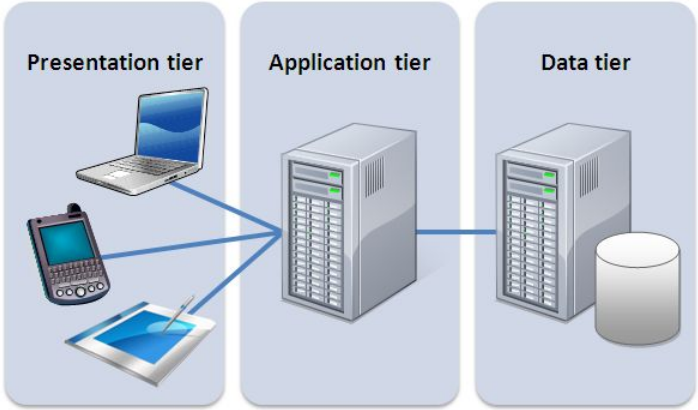
# SoC Examples



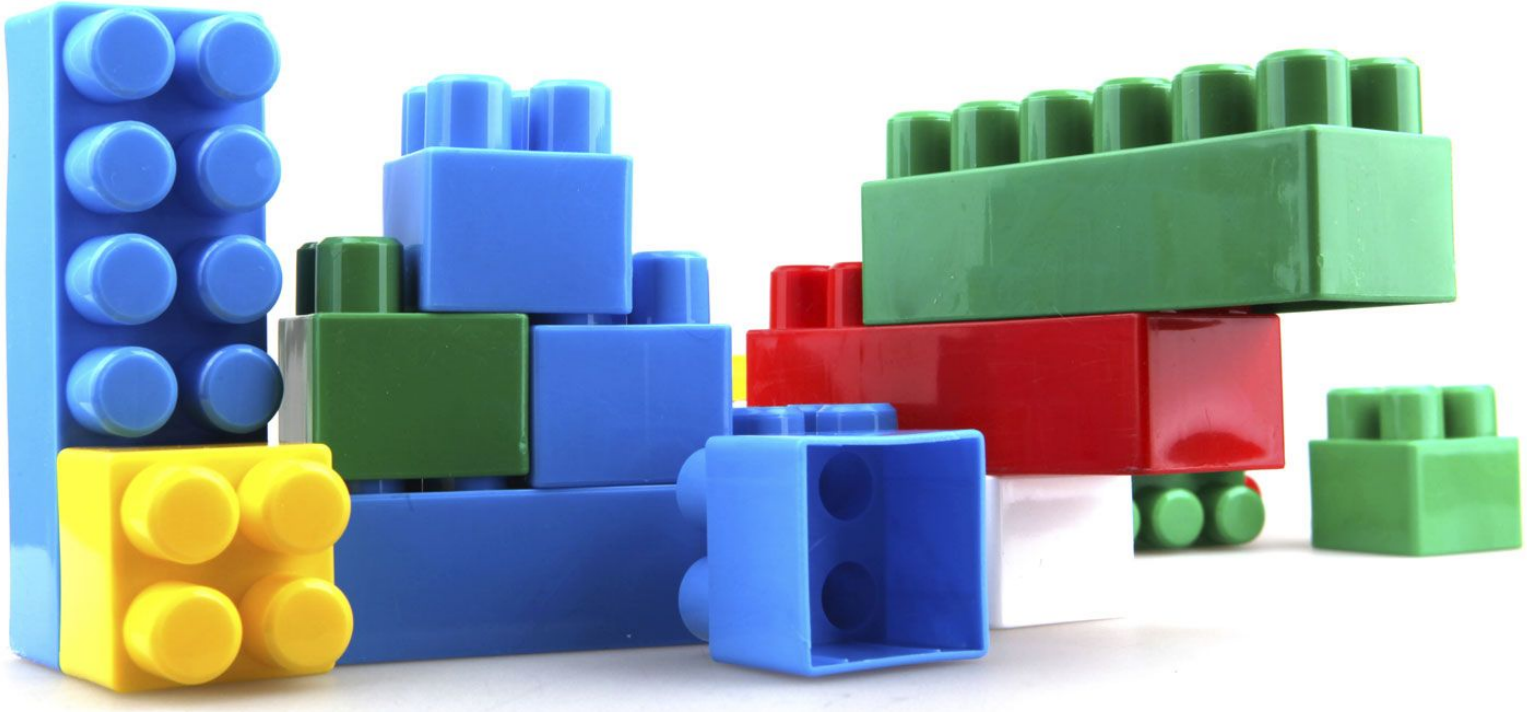



# Modularity



# Abstraction

# Hide details

# Encapsulation Keep changes local

# SOLID Principles

# SOLID Principles

| | |
|---|---|
| Single Responsibility Principle | **SRP** |
| Open Closed Principle | **OCP** |
| Dependency Inversion Principle | **DIP** |
| Interface Segregation Principle | **ISP** |
| Liskov's Substitution Principle | **LSP** |


A class should have only one reason to change.

https://github.com/Namozaq/design-principles/tree/master/src/main/java/dp/srp

# SRP

```
public class EmployeeManager {

    public Employee getEmployee(int id) {
        // do some DB operations
        return new Employee("Hafez Hamdy", 1500d);
    }

    public void printPaySlip(Employee employee) {
        System.out.println("Payslip");
        System.out.println("Name: " + employee.getName());
        System.out.println("Total: " + employee.getSalary());
    }

}
```

# OCP

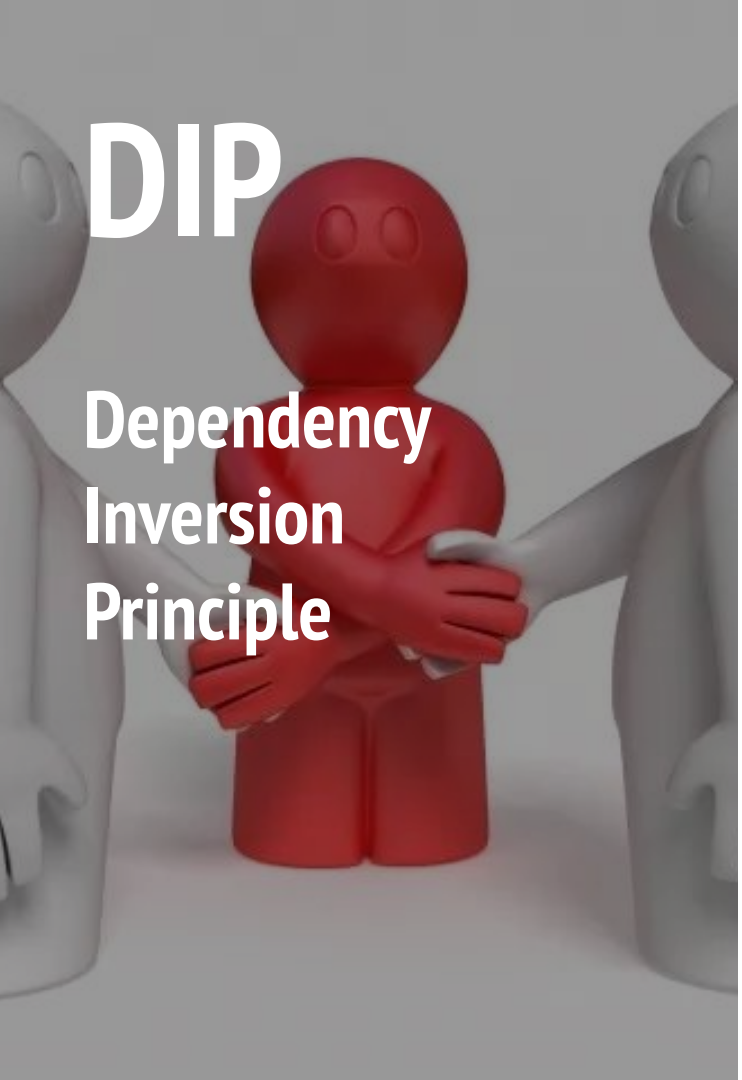
## Open Closed Principle

A module should be open for extension but closed for modification.

https://github.com/Namozaq/design-principles/tree/master/src/main/java/dp/ocp

# OCP

```
public class Painter {
    public void draw(Shape shape) {
        if (shape instanceof Circle) {
            drawCircle((Circle) shape);
        } else if (shape instanceof Rectangle) {
            drawRectangle((Rectangle) shape);
        }
    }
    private void drawCircle(Circle circle) {
        System.out.println("I'm a circle whose radius = " + circle.getRadius());
    }
    private void drawRectangle(Rectangle r) {
        System.out.println("I'm a rectangle whose area = " + r.getArea());
    }
}
```



# DIP

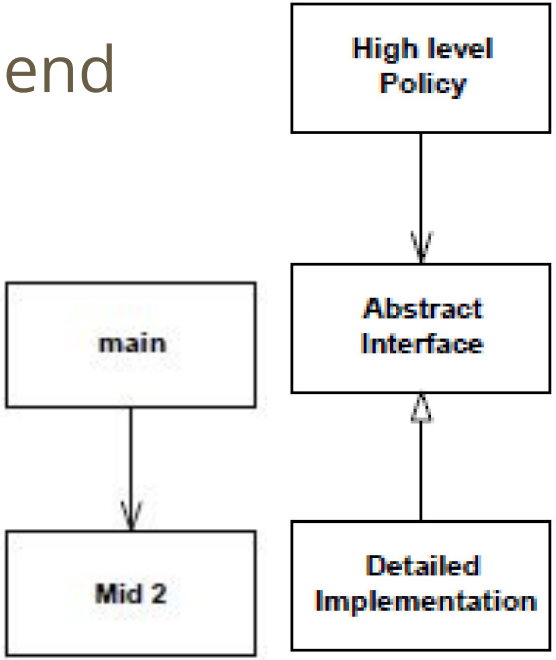
## Dependency Inversion Principle

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

Both should depend

on

abstractions.




https://github.com/Namozag/design-principles/tree/master/src/main/java/dp/dip

# DIP

```
public class Car {
    public void move() {
        println("beep beep");
    }
}

public class Bus {
    public void move() {
        println("toot toot");
    }
}
```

```
public class Driver {
    private Car car;
    public void setCar(Car car) {
        this.car = car;
    }
    public void drive() {
        car.move();
    }
}
```

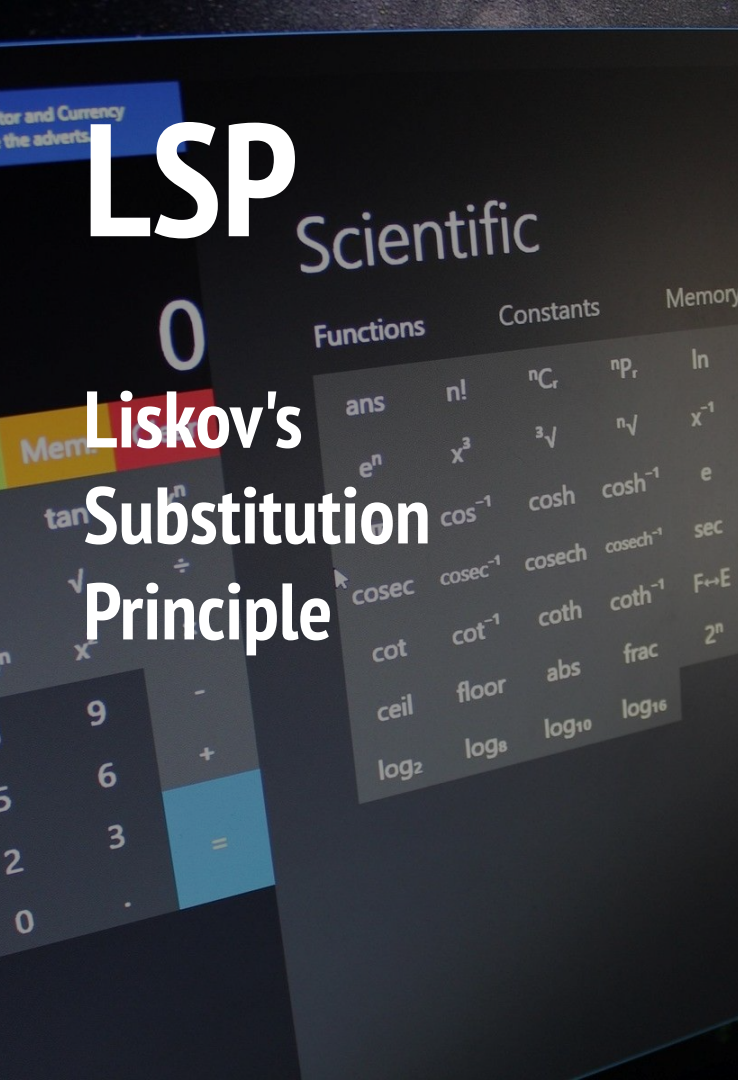
# ISP

## Interface Segregation Principle


Many client specific interfaces are better than one general purpose interface

Clients should not be forced to depend upon interfaces that they don't use.

https://github.com/Namozaq/design-principles/tree/master/src/main/java/dp/isp




# LSP

## Liskov's Substitution Principle

Subclasses should be substitutable for their base classes.

New derived classes are extending the base classes without changing their behavior

https://github.com/Namozaq/design-principles/tree/master/src/main/java/dp/lsp

# Other Principles

# Other Common Principles

DRY

KISS

YAGNI

LoD

Encapsulate what varies

Favor Composition over Inheritance


Every piece of knowledge must have a

- single
- unambiguous
- authoritative

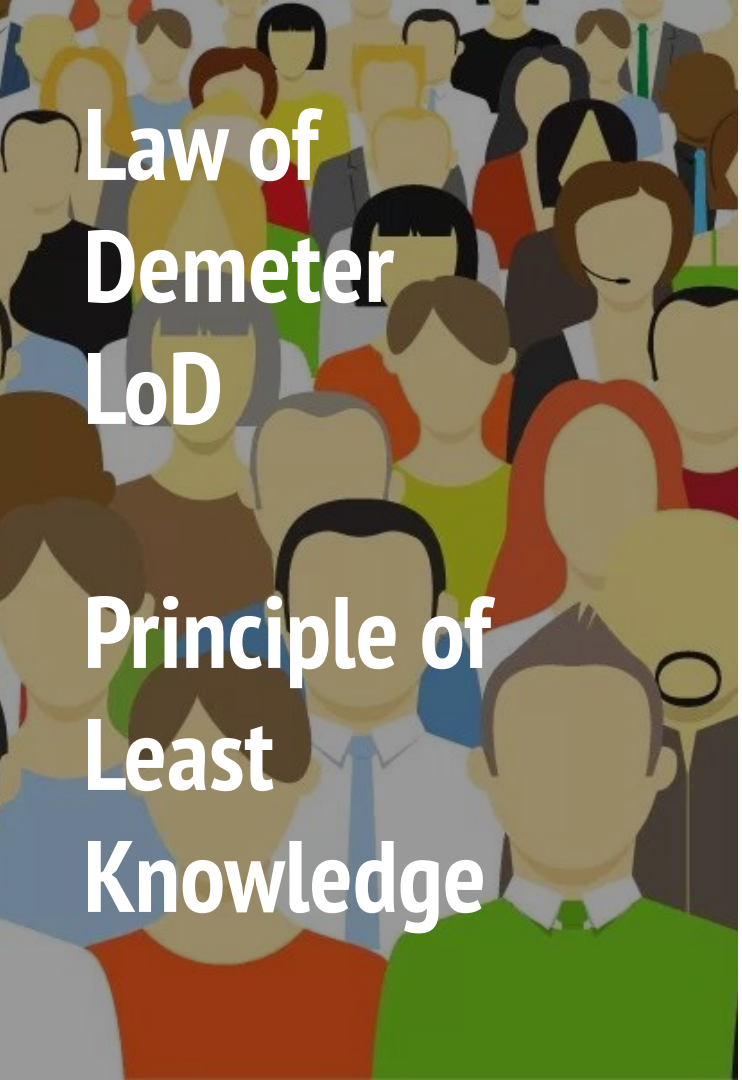
representation within a system

# DRY

```
void addUser(User u) {
     if (!u.mobile.matches("/^(\+\d{1,3}[- ]?)?\d{10}$/") {
          throw new RuntimeException("Invalid email");
     }
     return repo.addUser(u);
}

void updateUser(User u) {
     if (!u.mobile.matches("/^(\+\d{1,3}[- ]?)?\d{10}$/") {
          throw new RuntimeException("Invalid email");
     }
     return repo.updateUser(u);
}

<input type="text" pattern="/^(\+\d{1,3}[- ]?)?\d{10}$/" >
```



# Law of Demeter
# LoD

# Principle of Least Knowledge

A component or object should not know about internal details of other components or objects

# LoD

```
class Car {
    private Engine engine;
}

class Engine {
    public void start() {...}
}
```

```
class Driver {
    drive() {
        car.getEngine().start();
    }
}
```


Most systems work best if they are kept simple rather than made complicated


Programmer should not add functionality until deemed necessary

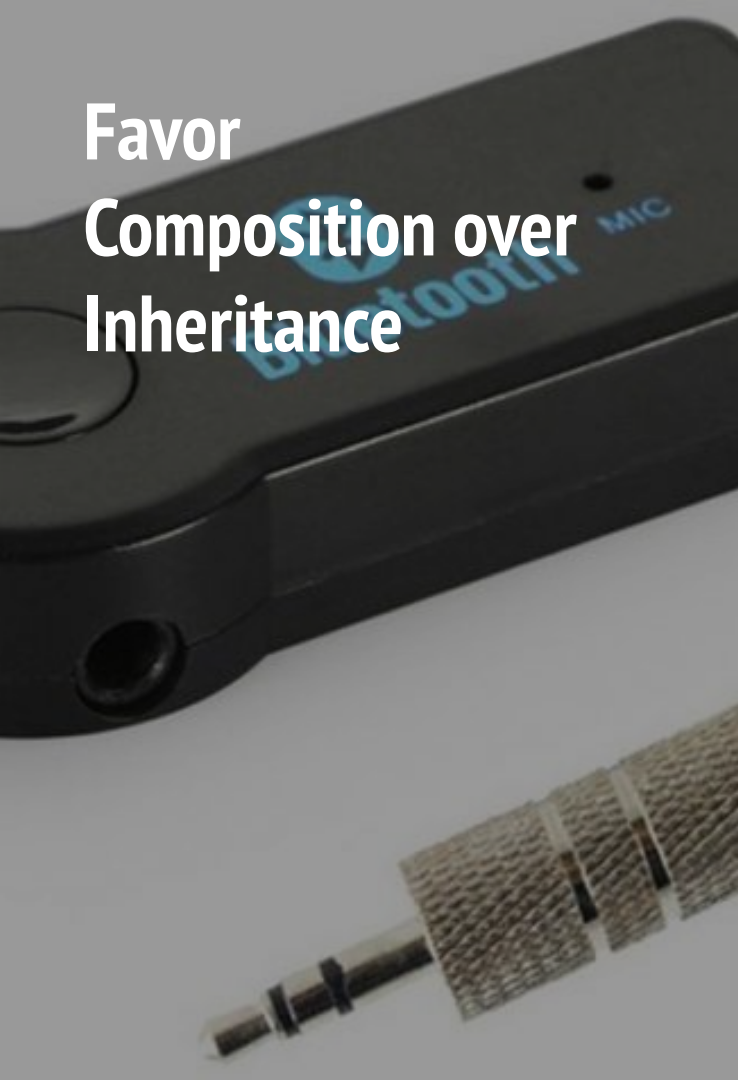
# YAGNI

```
UserService {
    User findUser(int id) {
        return repo.findUser(id);
    }
    void addUser(User u) {
        return repo.addUser(u);
    }
}
```

```
UserRepo {
    findUser(int id) {...}
    addUser(User user) {...}
    UpdateUser(User user) {...}
    deleteUser(int id) {...}
}
```

# Encapsulate what varies

Encapsulate the code you expect or suspect to be changed in future




# Favor Composition over Inheritance

Identifying system object behaviors in separate interfaces instead of creating a hierarchical relationship

# References


The Principles of OOD

Key Principles of Software Architecture

Principles of Software Design