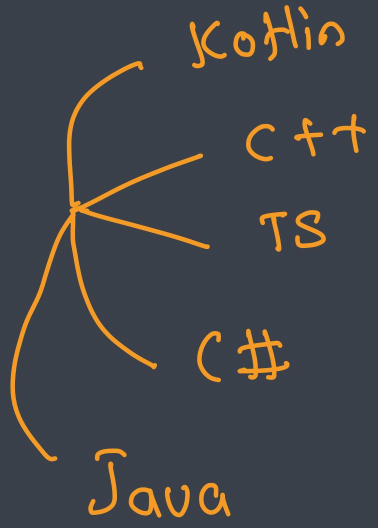
SOLID Principles

Object oriented Programming



Kotlin

C++

TS

C#

Java

Introduction

- The SOLID Principles are five principles of Object-Oriented class design
- They are a set of rules and best practices to follow while designing a class structure
- These five principles help us understand the need for certain design patterns and software architecture in general

→ client-server ✓

→ peer-to-peer ✗

→ microservices →

→ monolithic ✓

History

- The SOLID principles were first introduced by the famous Computer Scientist Robert J. Martin (a.k.a Uncle Bob) in his paper in 2000
- But the SOLID acronym was introduced later by Michael Feathers
- Uncle Bob is also the author of bestselling books Clean Code and Clean Architecture, and is one of the participants of the "Agile Alliance"
- Therefore, it is not a surprise that all these concepts of clean coding, object-oriented architecture, and design patterns are somehow connected and complementary to each other
- They all serve the same purpose:
  - To create understandable, readable, and testable code that many developers can collaboratively work on
- Following the SOLID acronym, they are:
  - The Single Responsibility Principle
  - The Open-Closed Principle
  - The Liskov Substitution Principle
  - The Interface Segregation Principle
  - The Dependency Inversion Principle

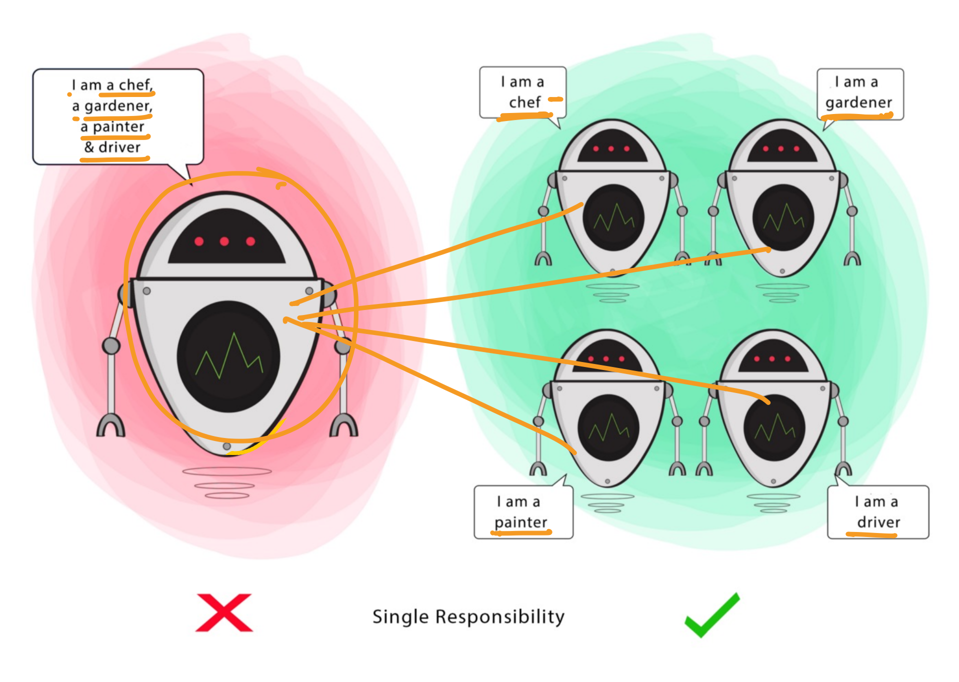
SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

Single Responsibility Principle

Wall-E

- The Single Responsibility Principle states that a class should do one thing and therefore it should have only a single reason to change
- To state this principle more technically
  - Only one potential change (database logic, logging logic, and so on.) in the software's specification should be able to affect the specification of the class
- If a Class has many responsibilities, it increases the possibility of bugs because making changes to one of its responsibilities, could affect the other ones without you knowing
- This means that if a class is a data container, like a Book class or a Student class, and it has some fields regarding that entity, it should change only when we change the data model
- Goal
  - This principle aims to separate behaviours so that if bugs arise as a result of your change, it won't affect other unrelated behaviours.



```
class student {

    roll : Number
    name : string
    class : string
          ⋮

    printDetails () {...}

    calculateFees () {...}

    calculateHolidays () {...}
          ⋮

}
```

Responsibilities

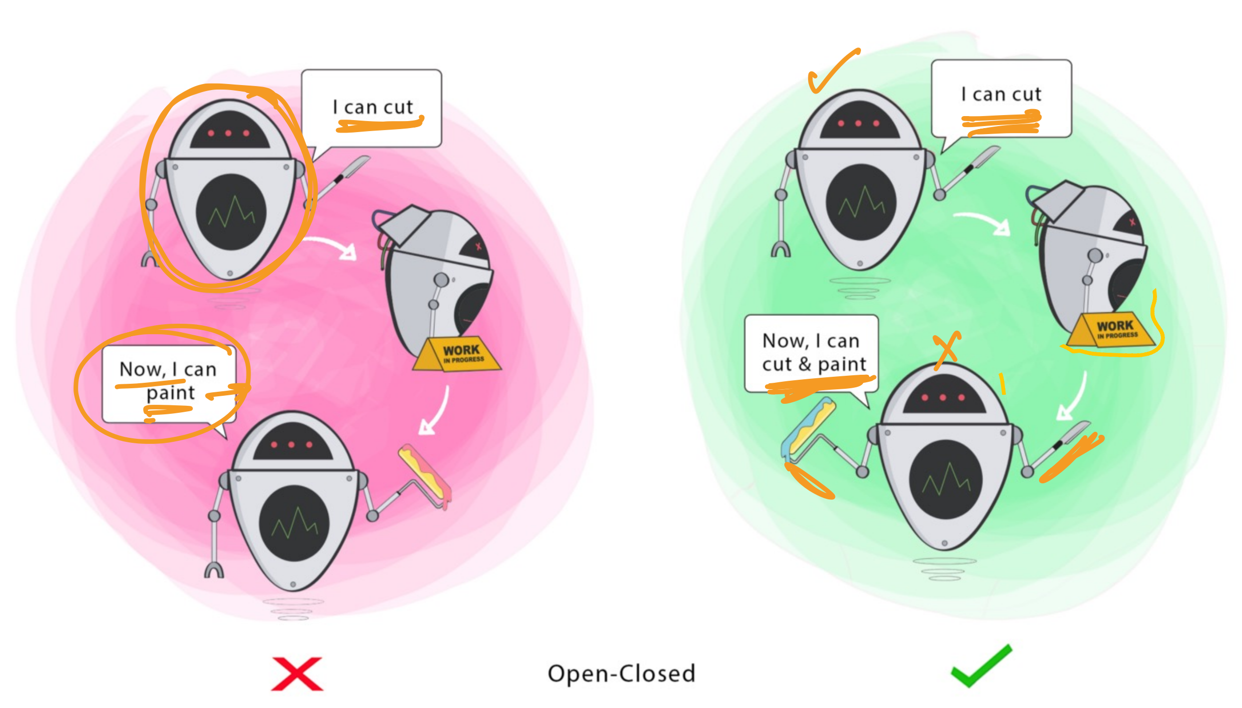
- Requirements changes typically map to responsibilities
- More responsibilities == More likelihood of change
- Having multiple responsibilities within a class couples together these Responsibilities
- The more classes a change affects, the more likely the change will introduce errors

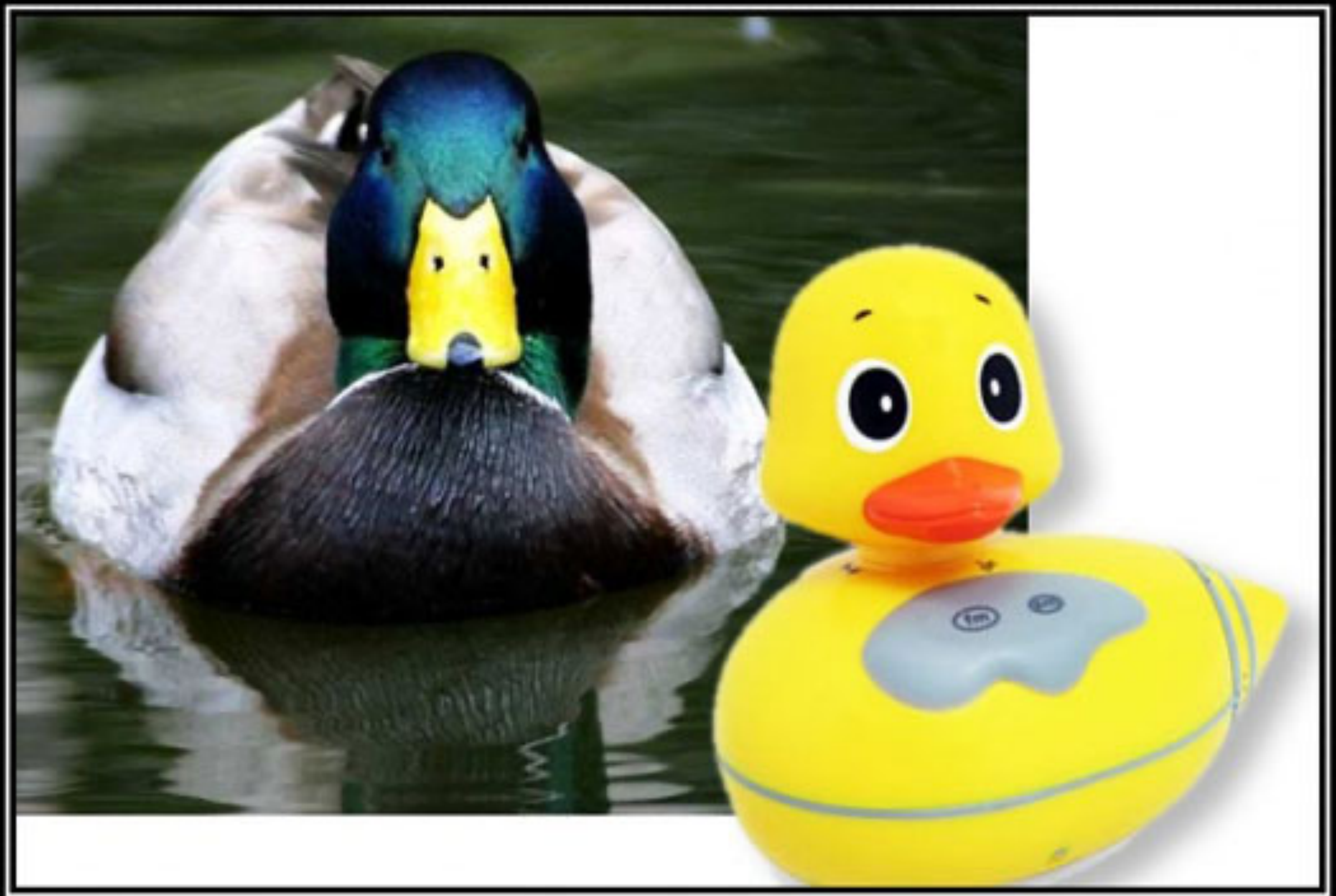
OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

Open Closed Principle

- The Open-Closed Principle requires that classes should be open for extension and closed to modification
- Modification means changing the code of an existing class, and extension means adding new functionality
  ↳ inheritance
- So what this principle wants to say is
  - We should be able to add new functionality without touching the existing code for the class
  - This is because whenever we modify the existing code, we are taking the risk of creating potential bugs
  - So we should avoid touching the tested and reliable (mostly) production code if possible
- Goal
  - This principle aims to extend a Class's behaviour without changing the existing behaviour of that Class
  - This is to avoid causing bugs wherever the Class is being used



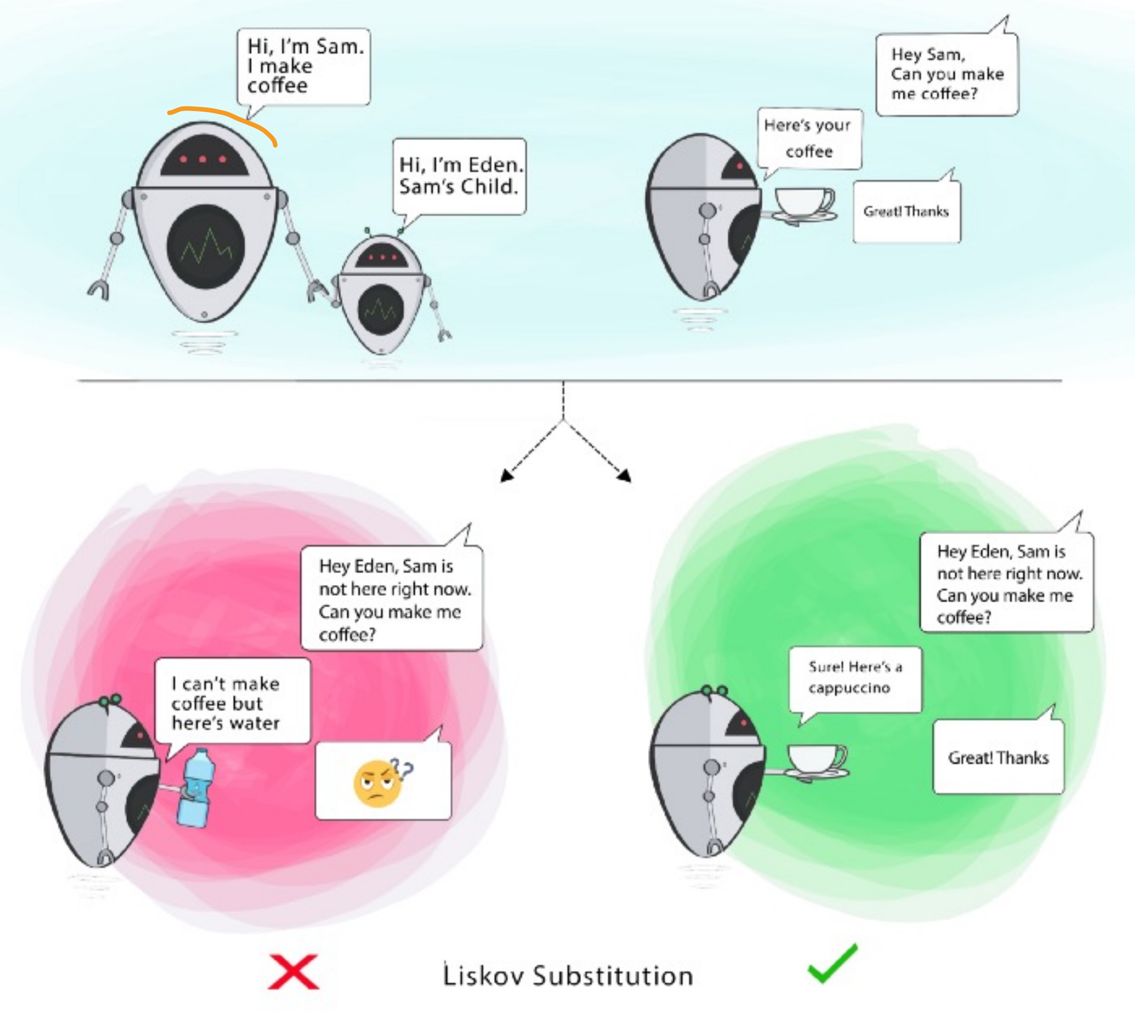


LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Liskov Substitution Principle

- The Liskov Substitution Principle states that subclasses should be substitutable for their base classes
- This means that, given that class B is a subclass of class A, we should be able to pass an object of class B to any method that expects an object of class A and the method should not give any weird output in that case
- This is the expected behavior, because when we use inheritance we assume that the child class inherits everything that the superclass has
- The child class extends the behavior but never narrows it down
- Therefore, when a class does not obey this principle, it leads to some nasty bugs that are hard to detect
- Goal
  - This principle aims to enforce consistency so that the parent Class or its child Class can be used in the same way without any errors


Hi, I'm Sam. I make coffee
Hi, I'm Eden. Sam's Child.
Hey Sam, Can you make me coffee?
Here's your coffee
Great! Thanks
Hey Eden, Sam is not here right now. Can you make me coffee?
I can't make coffee but here's water
??
Hey Eden, Sam is not here right now. Can you make me coffee?
Sure! Here's a cappuccino
Great! Thanks
✗
Liskov Substitution
✓

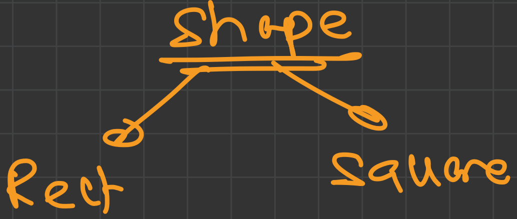
```
class Rectangle{
    width: number;
    height: number

    setWidth(..){.. }
    setHeight(..){...}
    calcArea(){...}
}
```

Shape{ }

Square extends Shape {.

Rectangle → → ..



~~class Square extends Rectangle{~~

~~setWidth(){ this.width = newW.. this.height = newW}~~

~~setHeight(){ this.width = newH; this.hei = neH; }~~

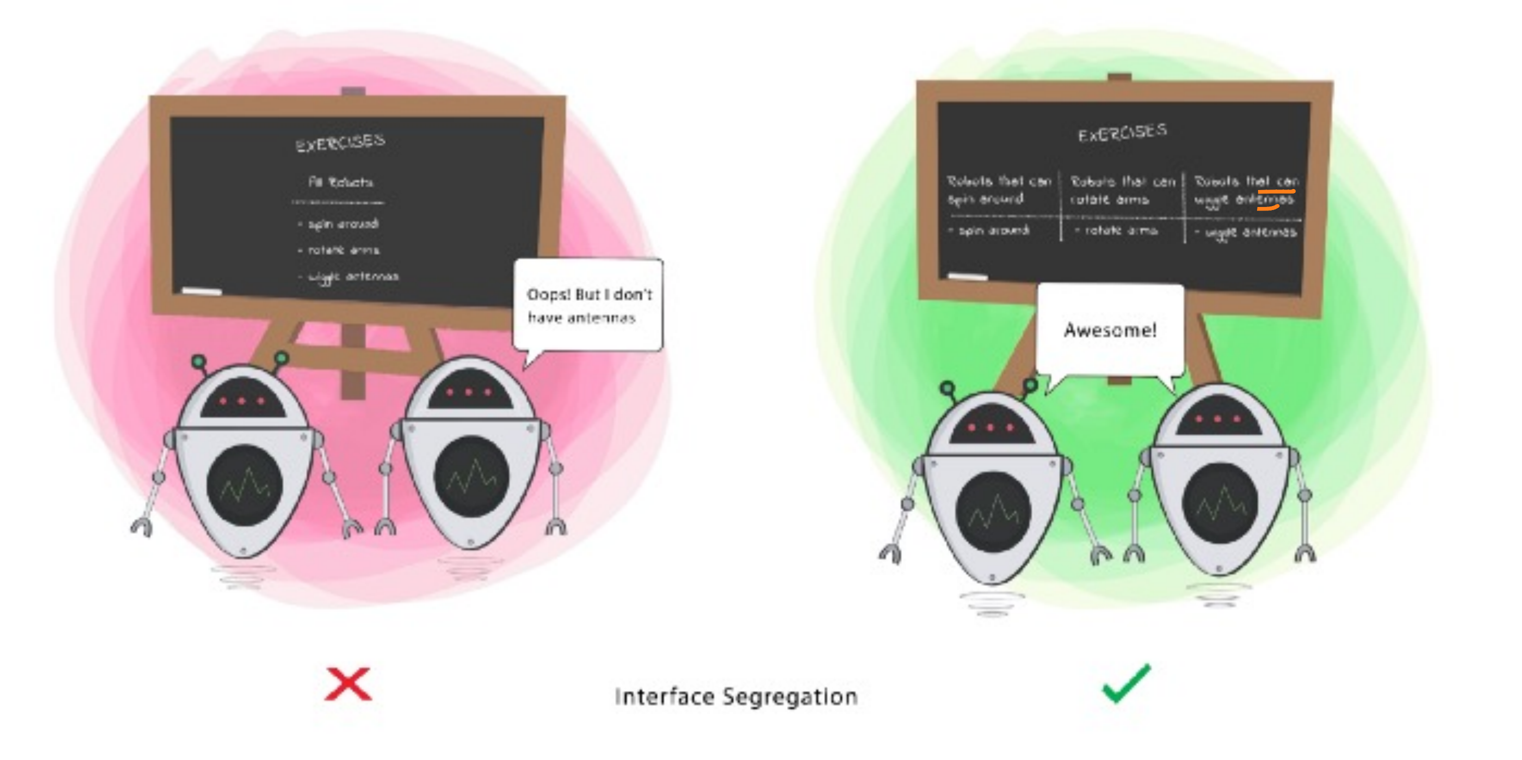
~~}~~

INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

Interface Segregation Principle

- Segregation means keeping things separated, and the Interface Segregation Principle is about separating the interfaces
- The principle states that many client-specific interfaces are better than one general-purpose interface
- Clients should not be forced to implement a function they do no need
- Goal
  - This principle aims at splitting a set of actions into smaller sets so that a Class executes ONLY the set of actions it requires



interface

service Service

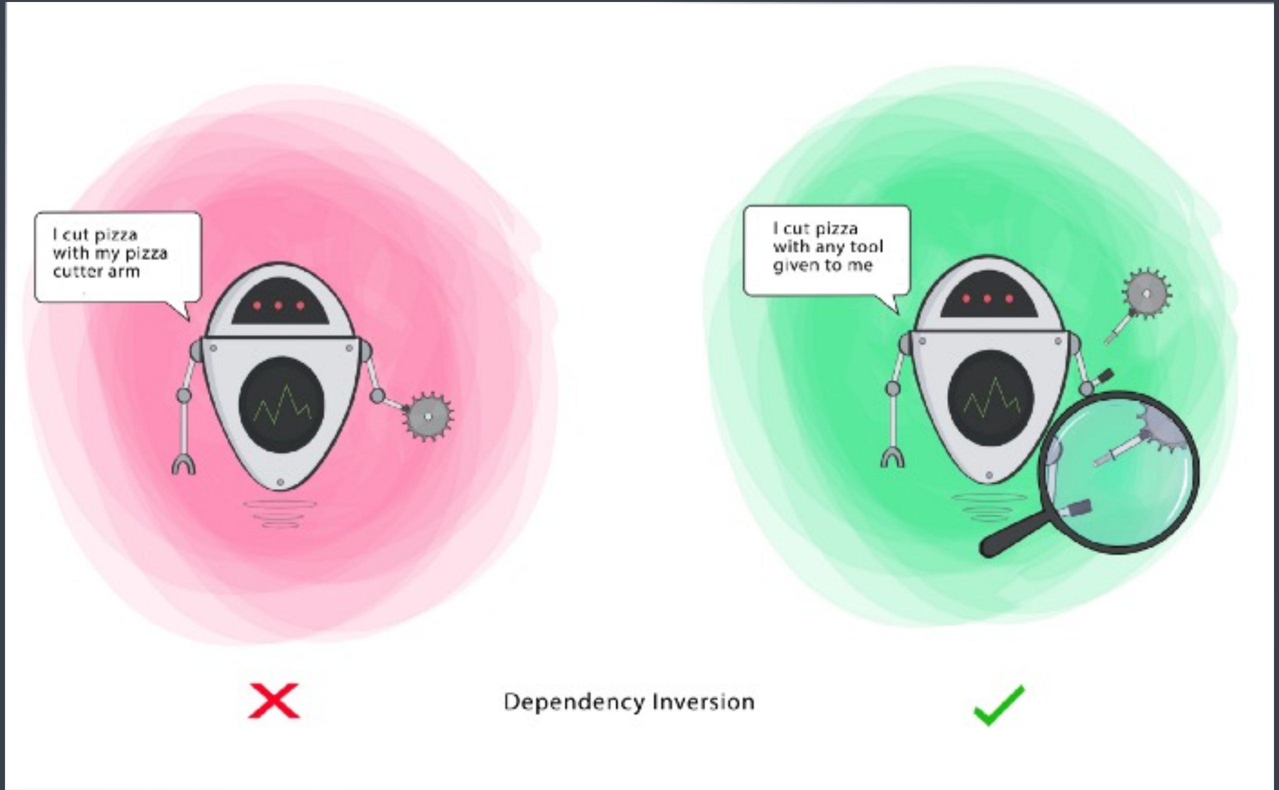
↳ contract between provider & consumer

DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Dependency Inversion Principle

- The Dependency Inversion principle states that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions
- In his article (2000), Uncle Bob summarizes this principle as follows:
  - If the OCP states the goal of OO architecture, the DIP states the primary mechanism
- We want our classes to be open to extension, so we have reorganized our dependencies to depend on interfaces instead of concrete classes
- Goal
  - This principle aims at reducing the dependency of a high-level Class on the low-level Class by introducing an interface


I cut pizza with my pizza cutter arm
I cut pizza with any tool given to me
✗
Dependency Inversion
✓

programming methodologies

- → procedural programming → C
  function
- → OOP → TS
- → scripting → JS, Bash scripting
- → functional programming → JS ← map, filter, forEach

Development methodologies

- → SDLC
- → Waterfall
- → Iterative
- → Agile →
  - → Scrum ← * * * *
  - → Kanban

e-commerce

- user ←
- product ←
- order
- cart

e-commerce → project

Product **Backlog**

Story

Task

→ user module → ① signup

- → create database & table
- → create an API – backend
- → create UI (frontend)

② signin

- → create an API (backend)
- → create UI (frontend)

③ change password

scrum

client

- roles → scrum master, product (manager)

scrum team (dev, tester, UI/UX ..)

→ events → sprint : time bound event

1 w to 4 w

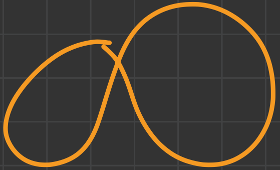
2w

↓

stories

⇓

Sprint backlog



Jira

events

→ story mapping event →

→ Daily scrum → every 24 hrs

→ Sprint Review → Demo of working s/f

→ Sprint Retrospective