Reporter: Nguyen Duong Thu Thuy

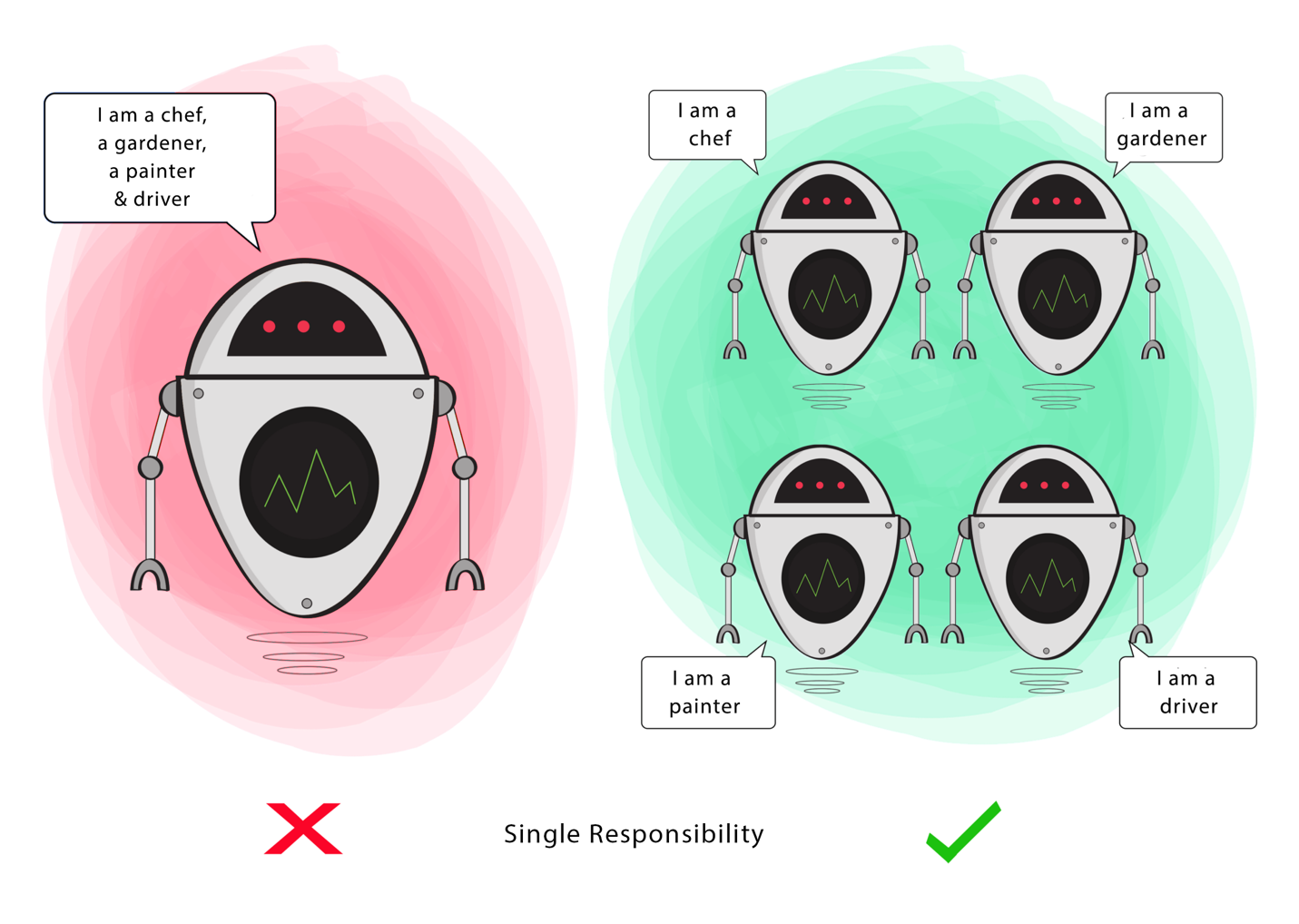
**SOLID PRINCIPLES**

Solid represents 5 principles of **Object Oriented Programming**

* Single Responsibility Principle
* Open-Closed Principle
* Liskov’s Substitution
* Interface’s Segregation
* Dependency Inversion

1. **S – Single Responsibility Principle**

A class should have one and only one reason to change OR A class should have only one responsibility



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.

Eg:



* How many responsibility does this class have ?
* Gets the conversation data form the API
* Parse and create conversation objects from API response
* Saves the conversation array to Core Data Database

If we have to implement NSURLSession for getting Json Data from API, and JSONSerialization to parse and Core Data Stack for saving to Database, this class becomes huge monster class.

we can solve this problem moving the each responsibility down to different classes



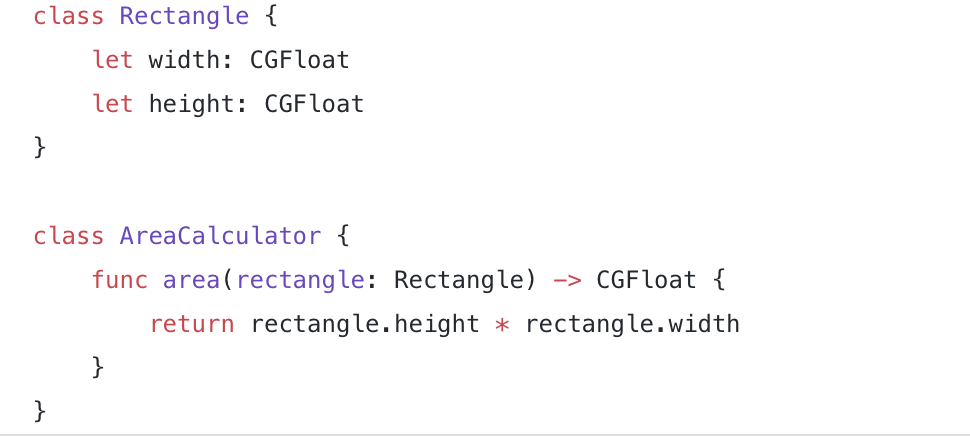
1. **Open – Closed Principle**

Classes and Modules should be open for extension but closed for modification

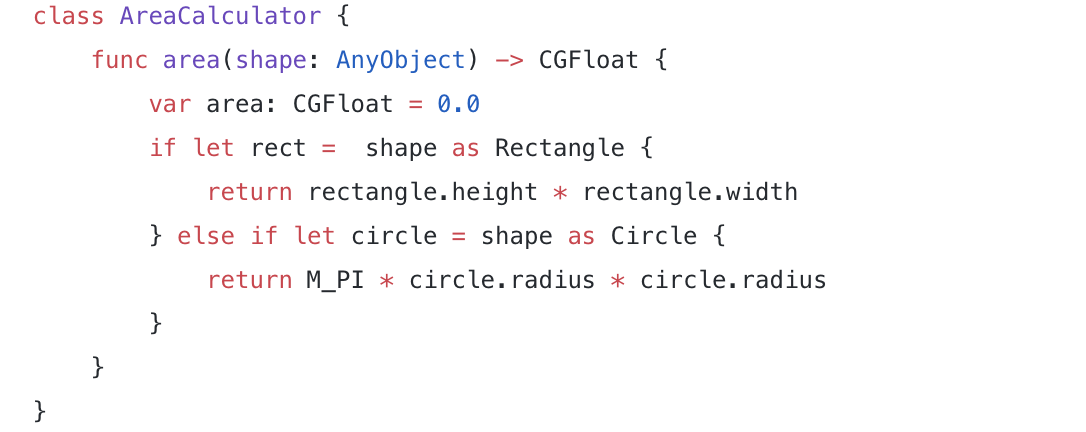
This principle aims to extend a Class’s behaviour without changing the existing behaviour of that Class. This is to avoid causing bugs wherever the Class is being used.

**Ex:**

Lets say we have requirement to calculate area of rectangle, so the rectangle and related area calculator class looks like shown below



Now the requirement has come to calculate area of not only Rectangle even for Circle, so our AreaCalculator class will look something like

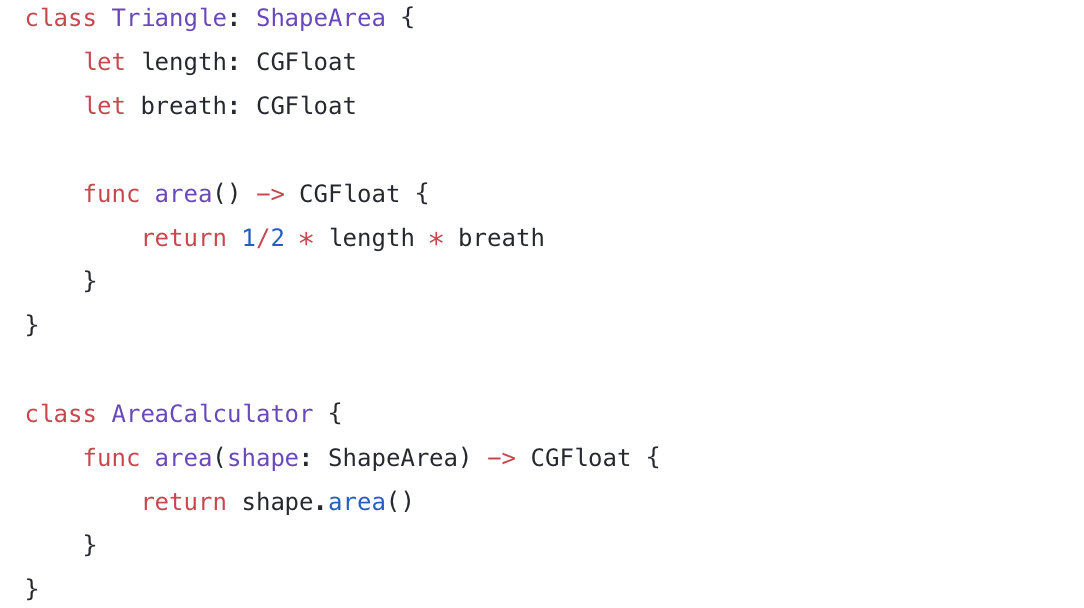


This will work but what if tomorrow one more requirement comes to calculate the area of Triangle, area function inside AreaCalculator class keeps growing with if else conditions, according to Open Close Principle we should not modify the existing class rather extent it.

So we change to:



so if there is requirement tomorrow to calculate the area of even Triangle, we should be able to achieve it without modifying the existing AreaCalculator class just by extending the Shape Protocol for Triangle, This obeys the SOLID Open Close Principle.



1. **L – Liskov’s Substitution**

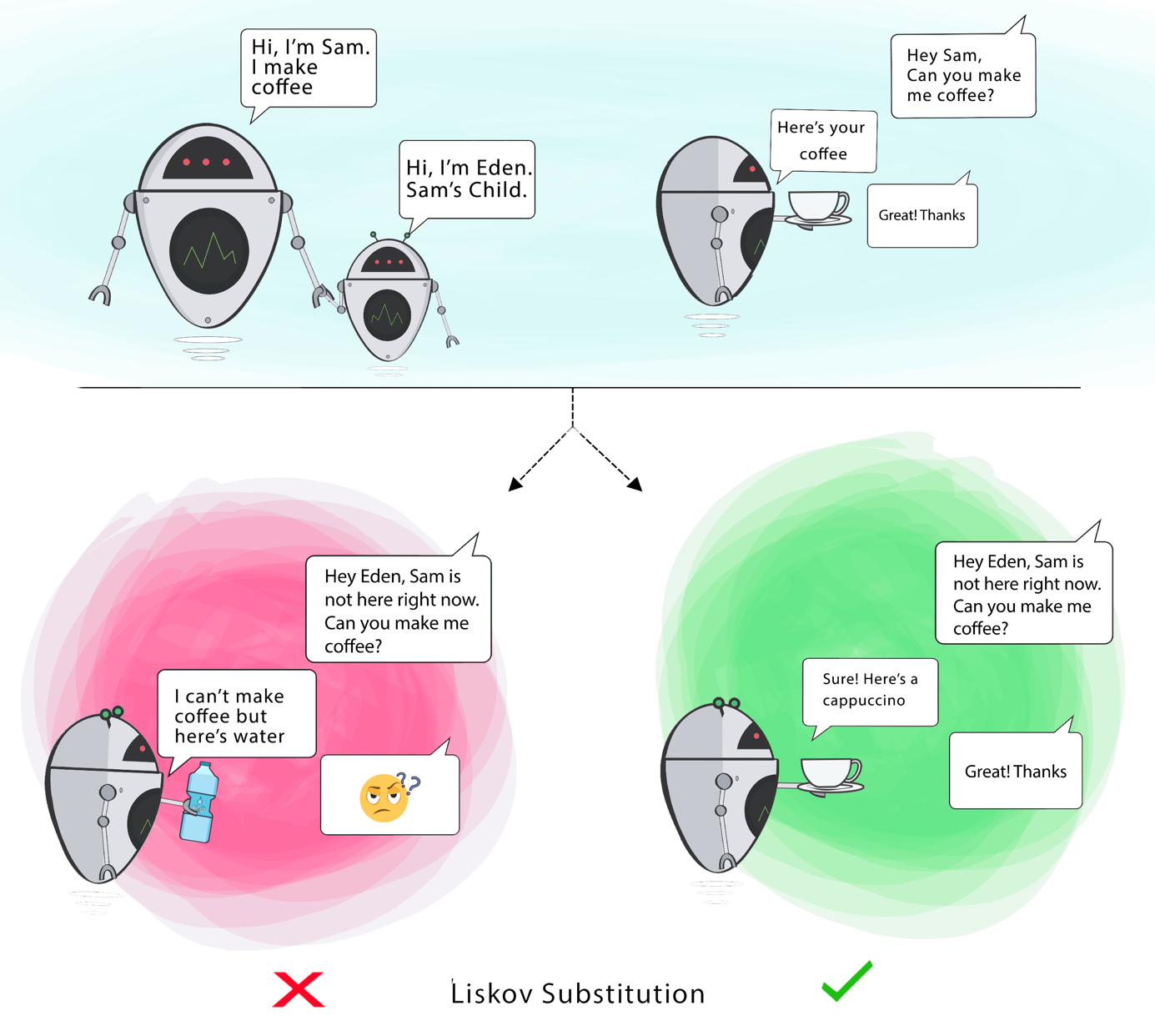
**Child classes** should **never break** the **parent class** type definitions

If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program.

When a child Class cannot perform the same actions as its parent Class, this can cause bugs.

If you have a Class and create another Class from it, it becomes a parent and the new Class becomes a **child**. The **child** Class should be able to do everything the **parent** Class can do. This process is called **Inheritance**.

The child Class should be able to process the same requests and deliver the same result as the **parent** Class or it could deliver a result that is of **the same type**.



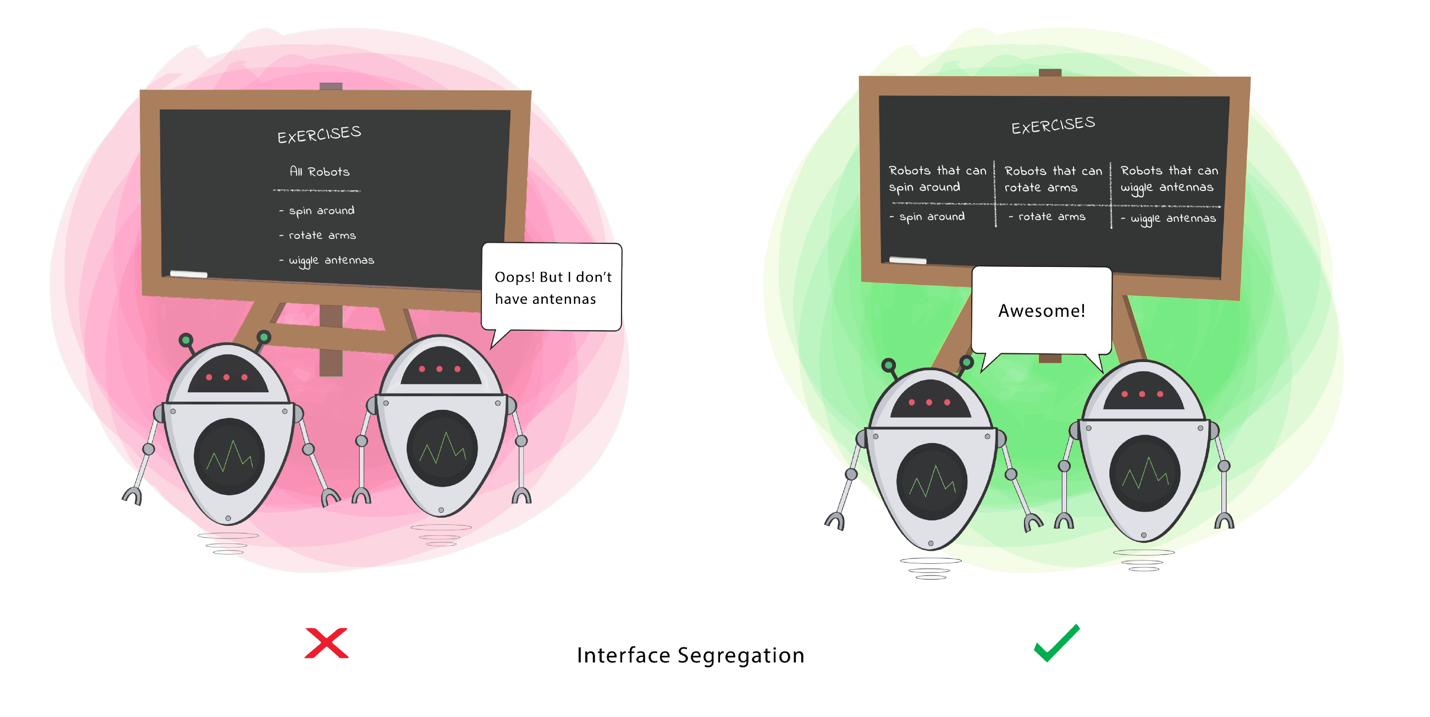
Goal: This principle aims to enforce consistency so that the parent Class or its child Class can be used in the same way without any errors.

1. **I – Interface Segregation Principle**

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.

When a Class is required to perform actions that are not useful, it is wasteful and may produce unexpected bugs if the Class does not have the ability to perform those actions.

A Class should perform only actions that are needed to fulfil its role. Any other action should be removed completely or moved somewhere else if it might be used by another Class in the future.



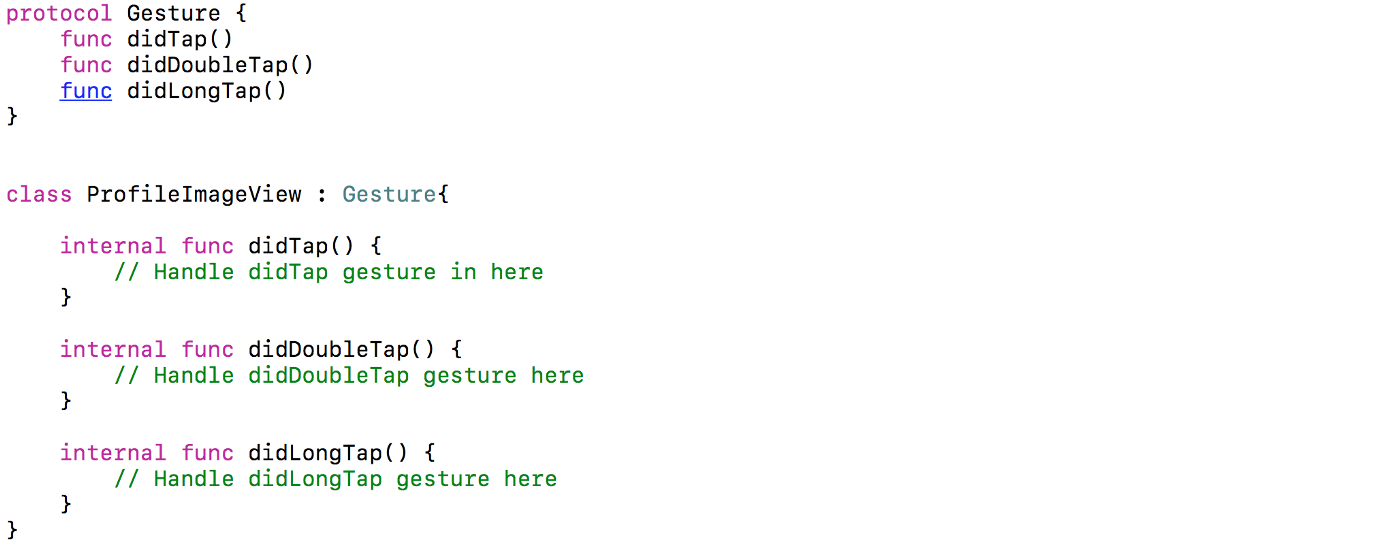
This principle solves FAT interface problems of Object Oriented Programming

A interface is called FAT when it has too many methods which contains more information than we really want.

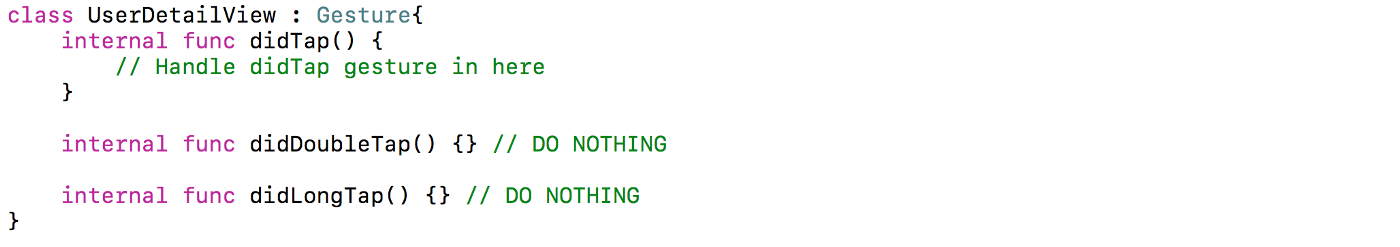
Lets understand Fat Interface in Protocols with an example, we have a GestureProtocol with the method didTap: which a View would be using it to handle the Gesture



After sometime as usual the requirement came to handle more gesture like didDoubleTap: and didLongPress: so our GestureProtocol becomes



Again as usual not all the Views in our app requires all the gestures some View would require only didTap: gesture in that case our UserDetailView class which would need only didTap: gesture would look like



We can solve the problem using little protocols instead of a big ones, the below code snippet shows segregation of Gesture protocols into small ones



1. D – Dependency Inversion Principle

High level modules should not depend on low level modules both should depend on Abstractions

Firstly, let’s define the terms used here more simply:

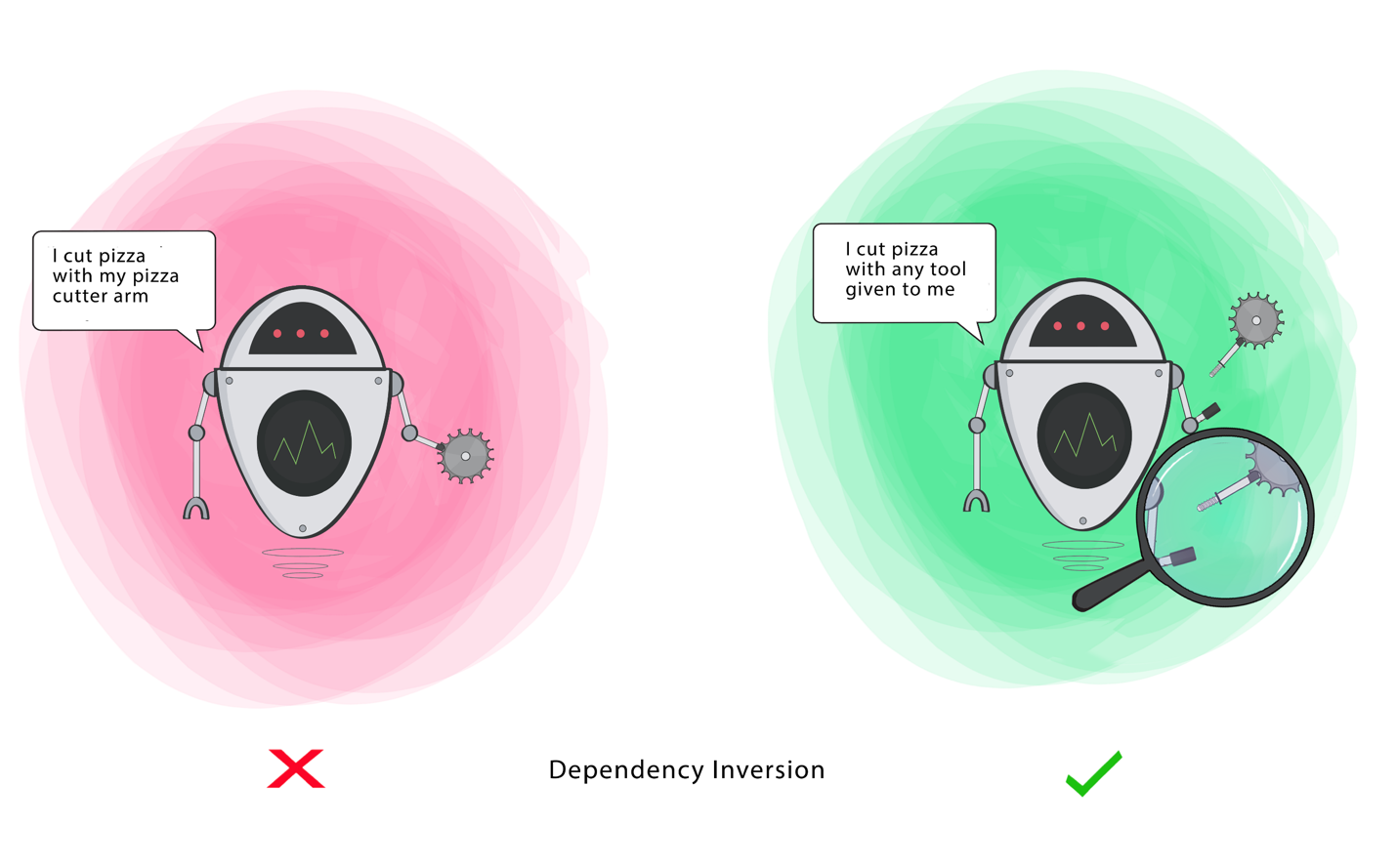
**High-level Module**(or Class): Class that executes an action with a tool.

**Low-level Module** (or Class): The tool that is needed to execute the action

**Abstraction:** Represents an interface that connects the two Classes.

**Details**: How the tool works

Goal: This principle aims at reducing the dependency of a high-level Class on the low-level Class by introducing an interface.



Ex:



Here the CoreDataController is a low level module, its easy to reuse in other projects, the problem is with high level module ConversationDataController, its tightly coupled with CoreDataController.

What if later, when a different database is needed? What are we going to do?

We can solve this dependency using a Database Protocol. In this way ConversationDataController can use abstract protocol without caring for type of database used, lets see how the ConversationDataController will look after applying this.

