SOLID Principles

Dart Version

13-06-2024

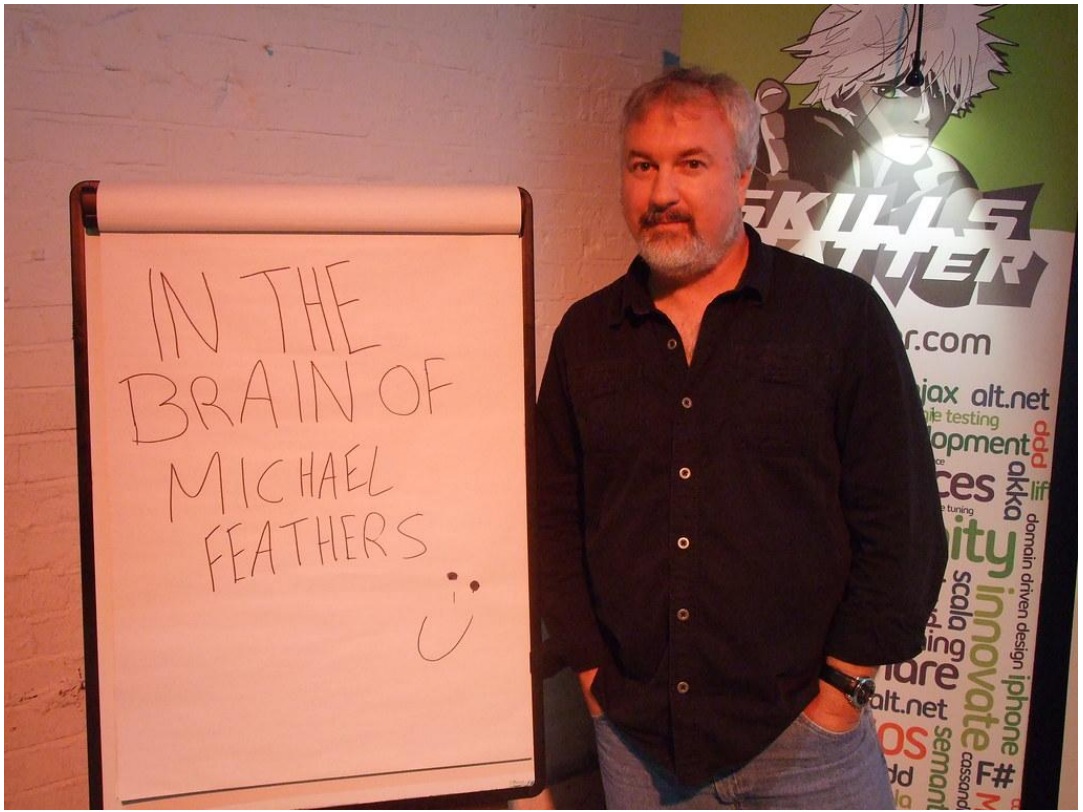
Presented By Benmoussa Younes

Table of content

Introduction

Robert C. Martin
(Uncle Bob)

SOLID
is an acronym
created by
Michael Feathers



Familiarity with OOP is recommended for better understanding

SOLID IS A MUST FOR EVERYONE CODING

1. Why SOLID Principales

- **Good software systems begin with clean code**

- **If the bricks aren't well made, the architecture of the building doesn't matter much.**

- **You can make a substantial mess with well-made bricks.**

This is where the SOLID principles come in.

Understanding the Goal

The Creation of mid-level software structure that

- **Tolerate change**
- **Are easy to understand**
- **Are the basis of components that can be used in many software systems.**

S Single Responsibility (SRP)

O Open-Closed (OCP)

L Liskov Substitution (LSP)

I Interface Segregation (ISP)

D Dependency Inversion (DIP)

SOLID

Single Responsibility (SRP)

2.1 Single Responsibility Principle

SRP might be the least well understood. Programmers assume that it means that every module should do just one thing.

- A function should do one, and only one, thing. **THIS IS NOT SRP**

SINGLE RESPONSIBILITY

- A module should be responsible to one, and only one, actor

- A class should have only one reason to change

Reason to change == Responsibility

The idea is simply to split our codebase into loosely coupled, isolated parts

Let take a look at the symptoms of violating it

SYMPTOM 1: ACCIDENTAL DUPLICATION

- We have coupled each of these actors to the others

- This coupling can cause the actions of the CFO's team to affect the COO's team

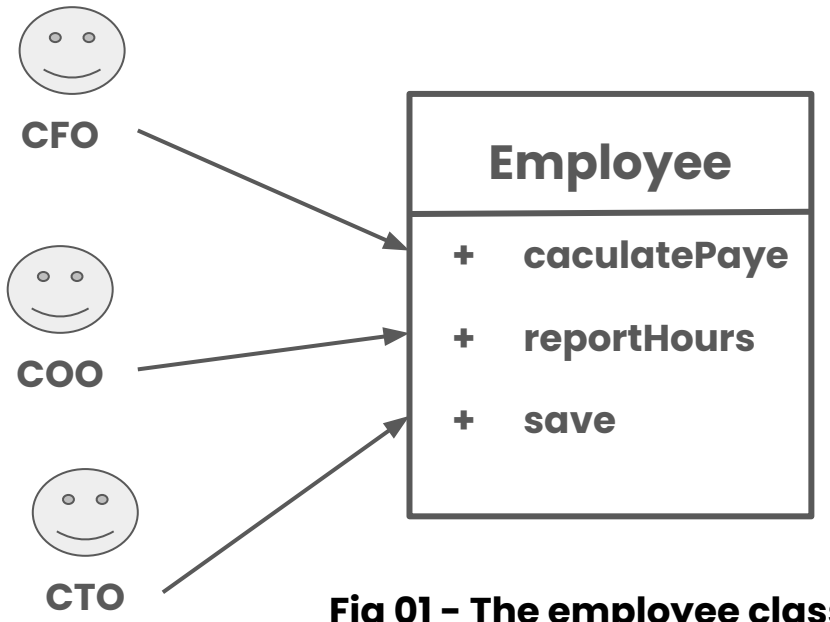


Fig 01 - The employee class

- The CFO's team wants to adjust how non-overtime hours are calculated

- **BUT** the COO's HR team opposes the change because they use non-overtime hours

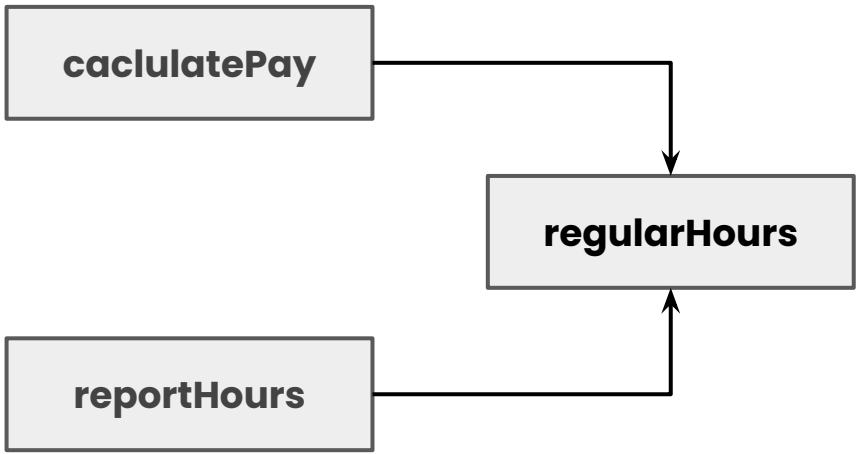


Fig 02 - Shared Function

- A developer sees the regularHours() function used in the calculatePay() method.

- Unfortunately, that developer does not notice that the function is also called by the **reportHours()** function

- **KBOM** HR personnel now use wrong reports generated by the reportHours()

SYMPTOM 2: MERGES

- Merges will be common in source files that contain many different methods (methods are responsible to different actors).

- Suppose that the CTO's team of DBAs decides that there should be a simple schema change to the Employee table of the database

- Suppose also that the COO's team of HR clerks decides that they need a change in the format of the hours report.

- Two different developers, possibly from two different teams, check out the Employee class and begin to make changes.

- Unfortunately their changes collide. The result is a merge.

AVOID HAVING

Multiple people changing the same source file for different reasons

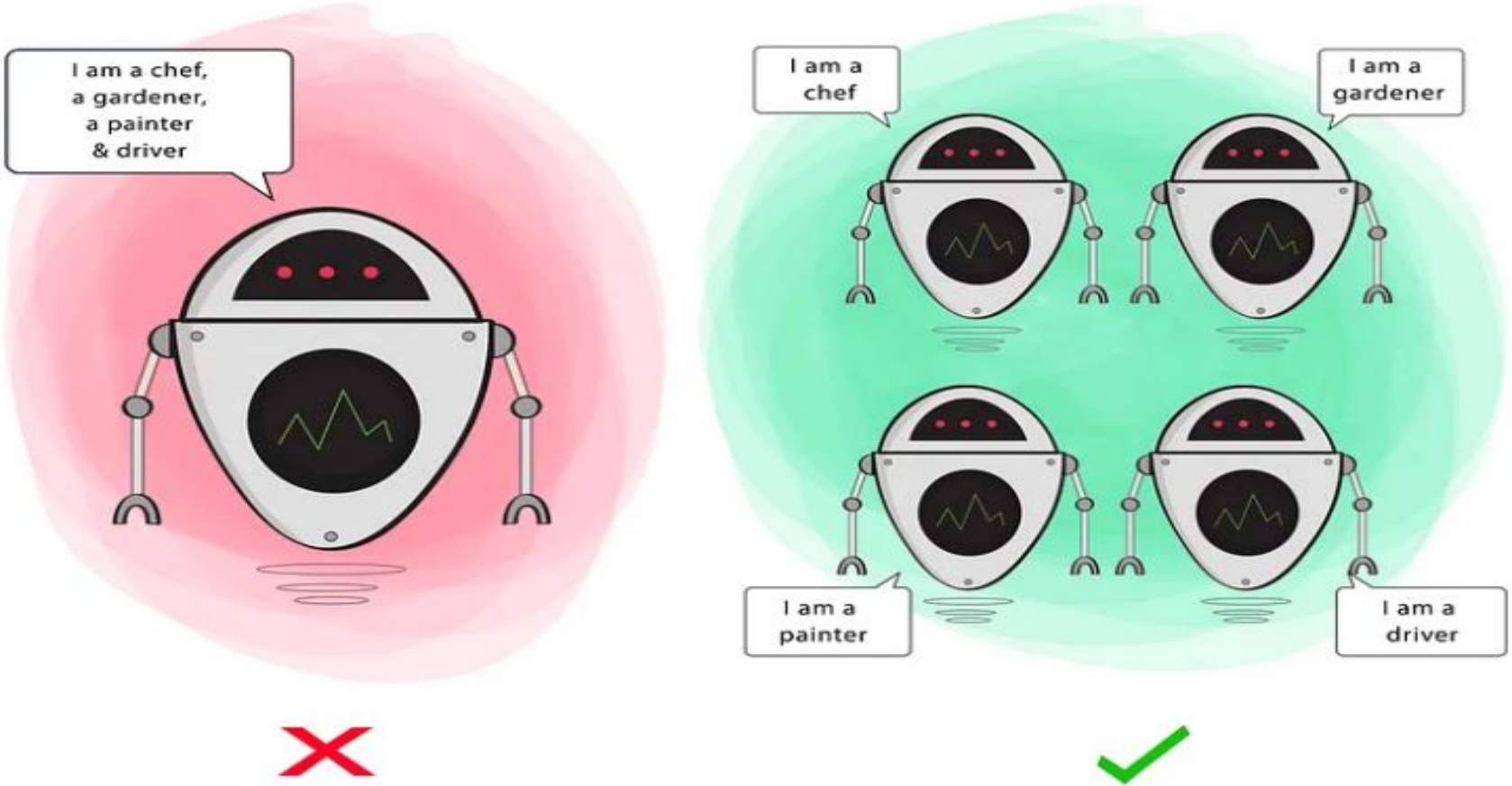


Fig 3 - Single Responsibility Principle Robot Example

SOLUTIONS

SOLUTION 1

The downside of this solution is that the developers now have three classes that they have to instantiate and track

A common solution to this dilemma is to use the Facade pattern

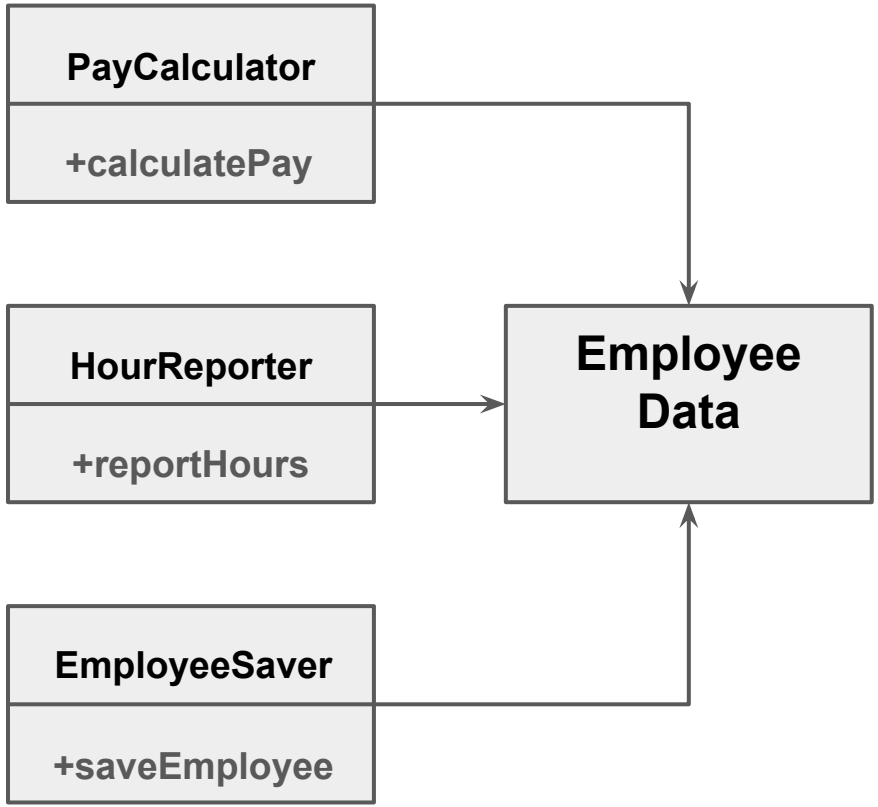


Fig 04- The three classes do not know about each other

SOLUTION 2

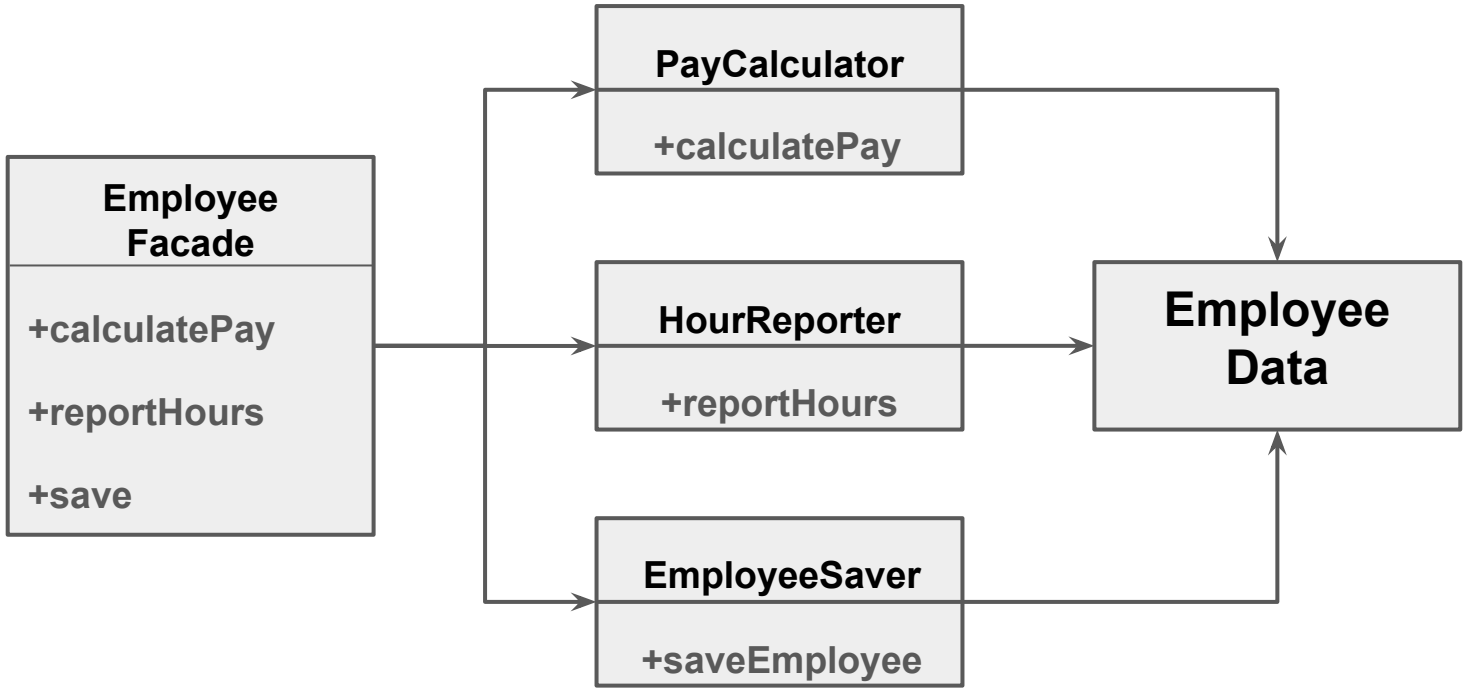


Fig 05 - The Facade pattern

SOLUTION 3

keep the most important business rules closer to the data.

We use the Employee class as a Facade

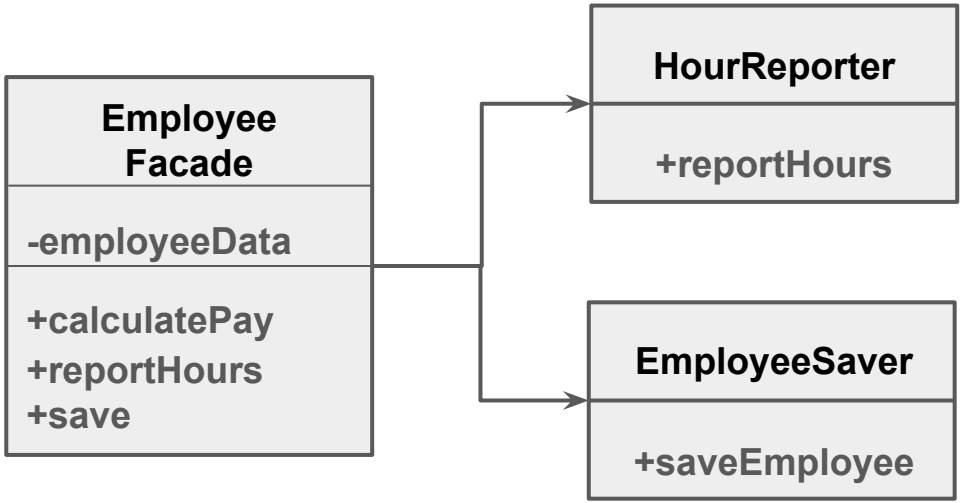


Fig 06- Employee class used as a Facade for the lesser functions

SRP Code Examples

- **The Single Responsibility Principle is about functions and classes**

- **At the level of components, it becomes the Common Closure Principle**

- **At the architectural level, it becomes the Axis of Change responsible for the creation of Architectural Boundaries.**


Single Responsibility Principle
Just because you *can* doesn't mean you *should*.

SOLID

Open-Closed (OCP)

2.2 Open-Closed Principle

- A software artifact should be open for extension but **closed for modification**.

- The behavior of a software artifact ought to be extendible, without having to modify that artifact.

We should be able to add new functionality to existing code **without altering its source**

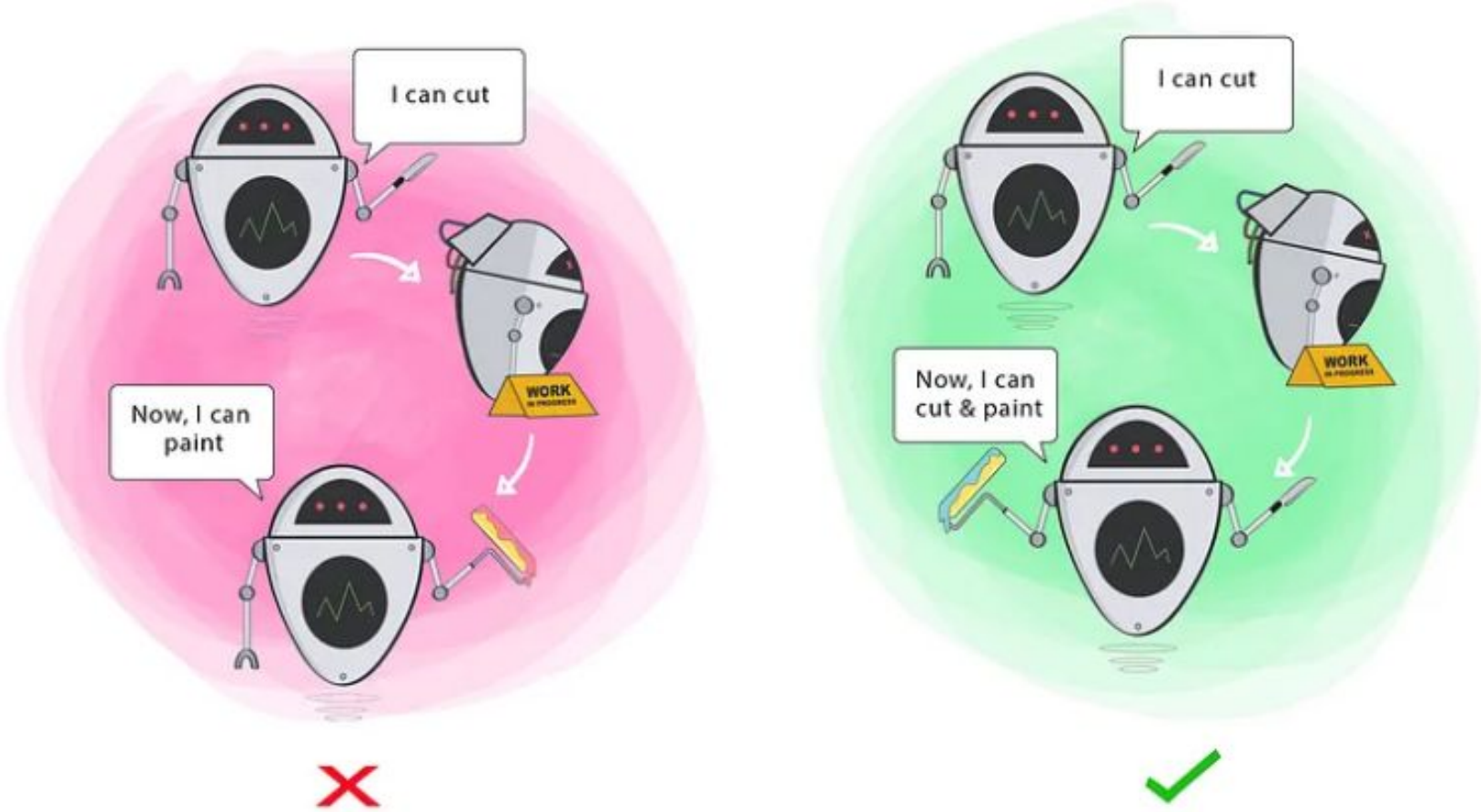


Fig 07 - Open Closed Principle Robot Example

Let take a look at some examples of violating it

OCP Code Examples

- **The goal of OCP is to make the system easy to extend without incurring a high impact of change.**

- **This goal is accomplished by partitioning the system into components, and arranging those components into a dependency hierarchy that protects higher-level components from changes in lower-level components.**


Open-Closed Principle
Open-chest surgery isn't needed when putting on a coat.

SOLID

Liskov Substitution (LSP)

2.3 Liskov Substitution Principle

- In 1988, Barbara Liskov wrote the following as a way of defining subtypes.

"If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T".

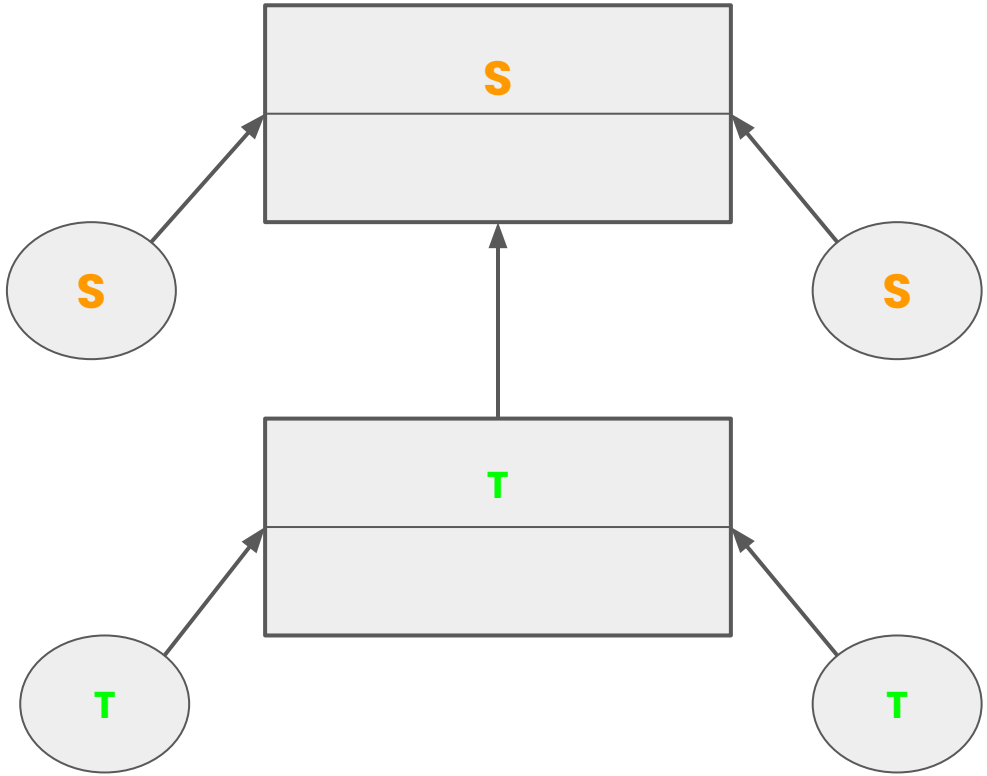


Fig 08 - Barbara Liskov substitution property model 1

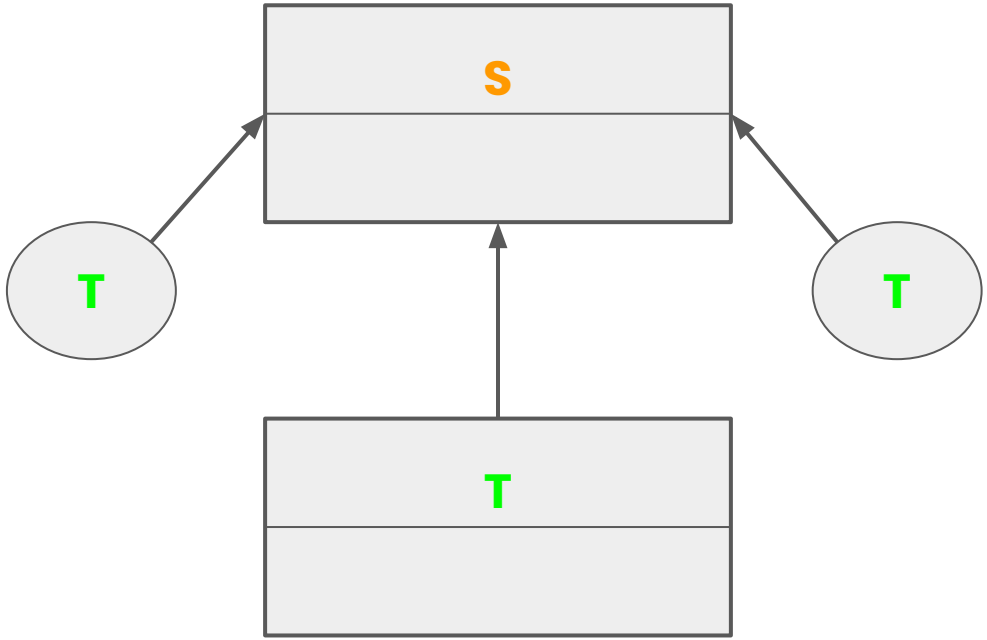


Fig 09 - Barbara Liskov substitution property model 2

Subclasses should behave nicely when used in place of their parent class

GOOD Example of LSP

- Both of the subtypes are substitutable for the License type.

- This design conforms to the LSP

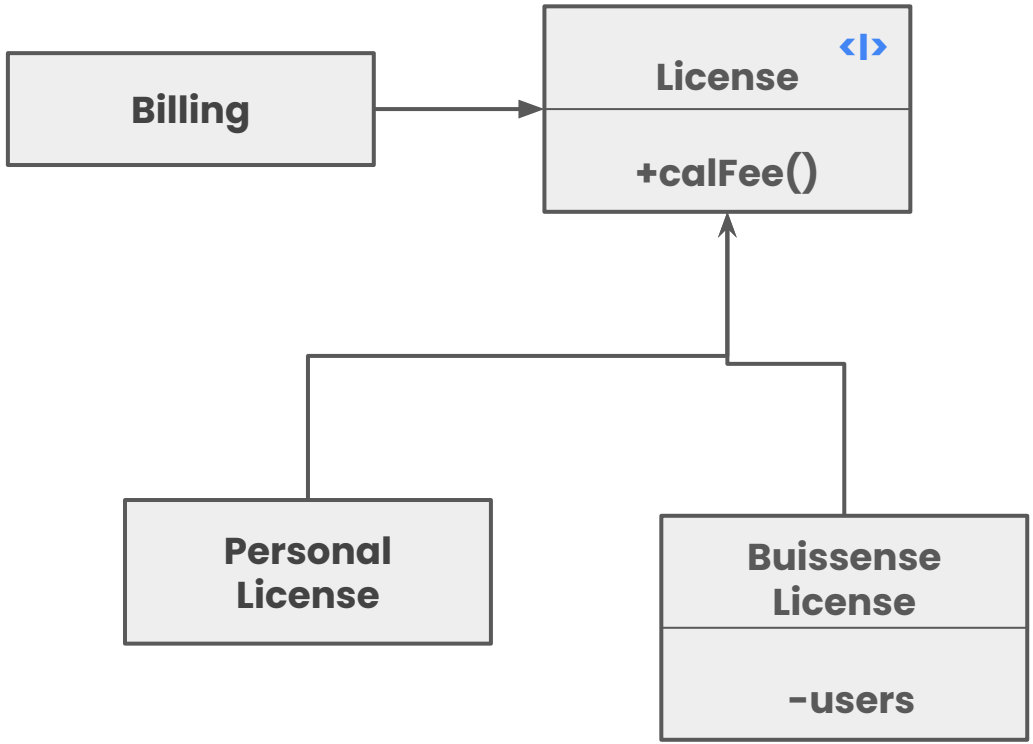
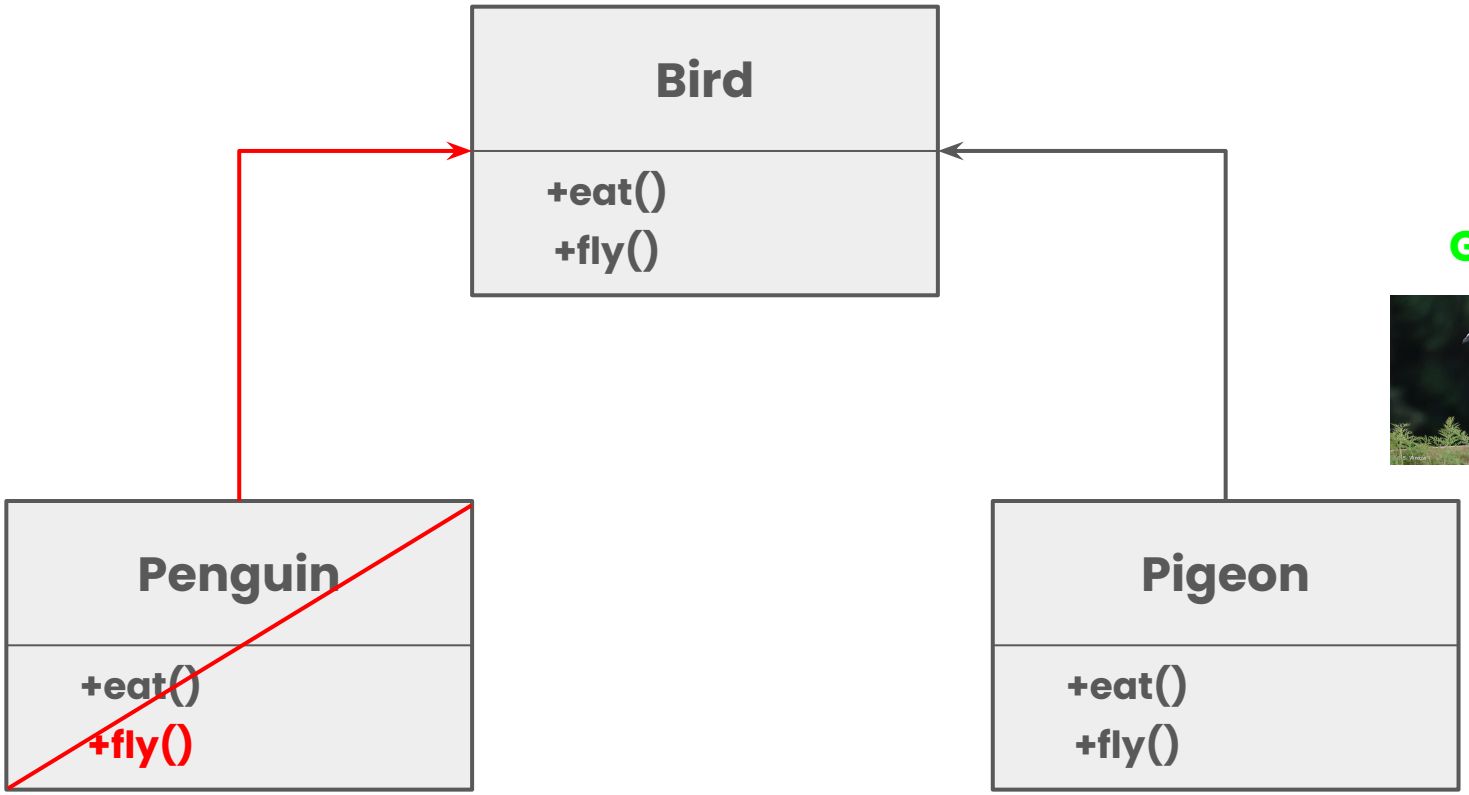


Fig 10 - License, and its derivatives, conform to LSP

Let take a look at some examples of violating it

BAD
Penguin
DO
NOT FLY



GOOD

Fig 11 - Barbara Liskov substitution property violated example

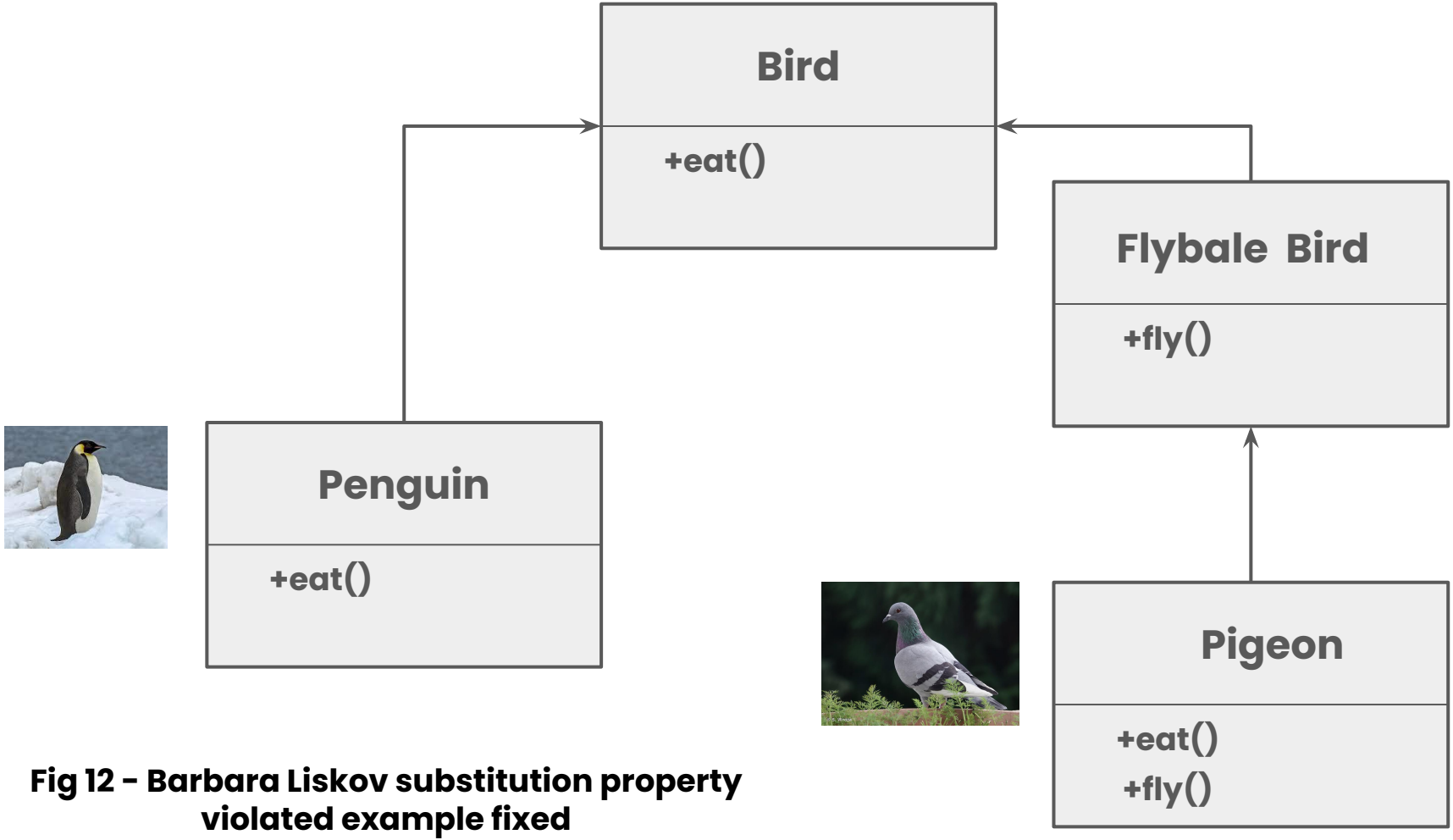


Fig 12 - Barbara Liskov substitution property violated example fixed

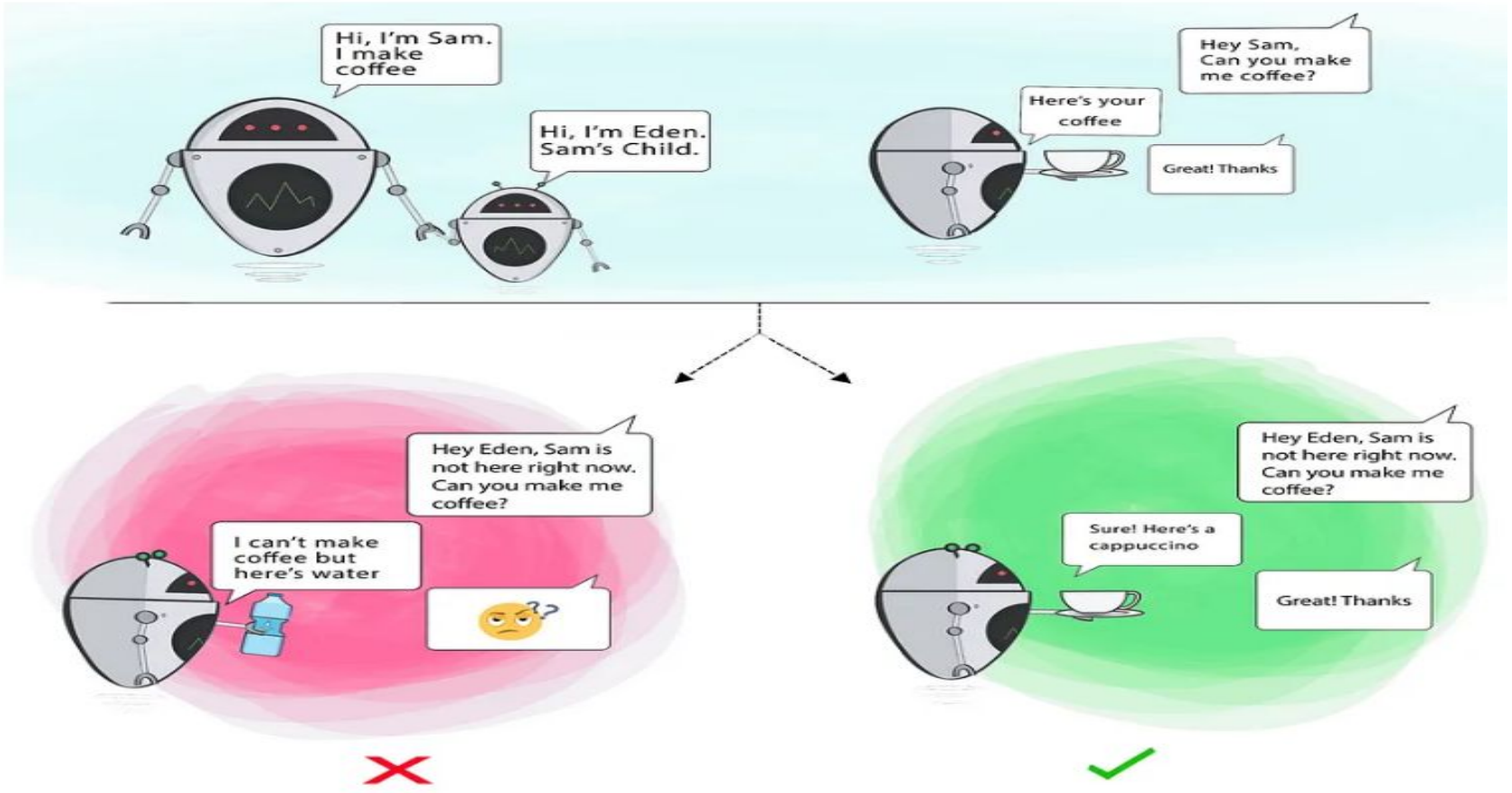


Fig 13 - Barbara Liskov substitution property
Sam - Eden Robots Example

LSP Code Examples

LSP And Architecture

- **The LSP can, and should, be extended to the level of architecture.**

- **A simple violation of substitutability, can cause a system's architecture to be polluted with a significant amount of extra mechanisms.**

Example

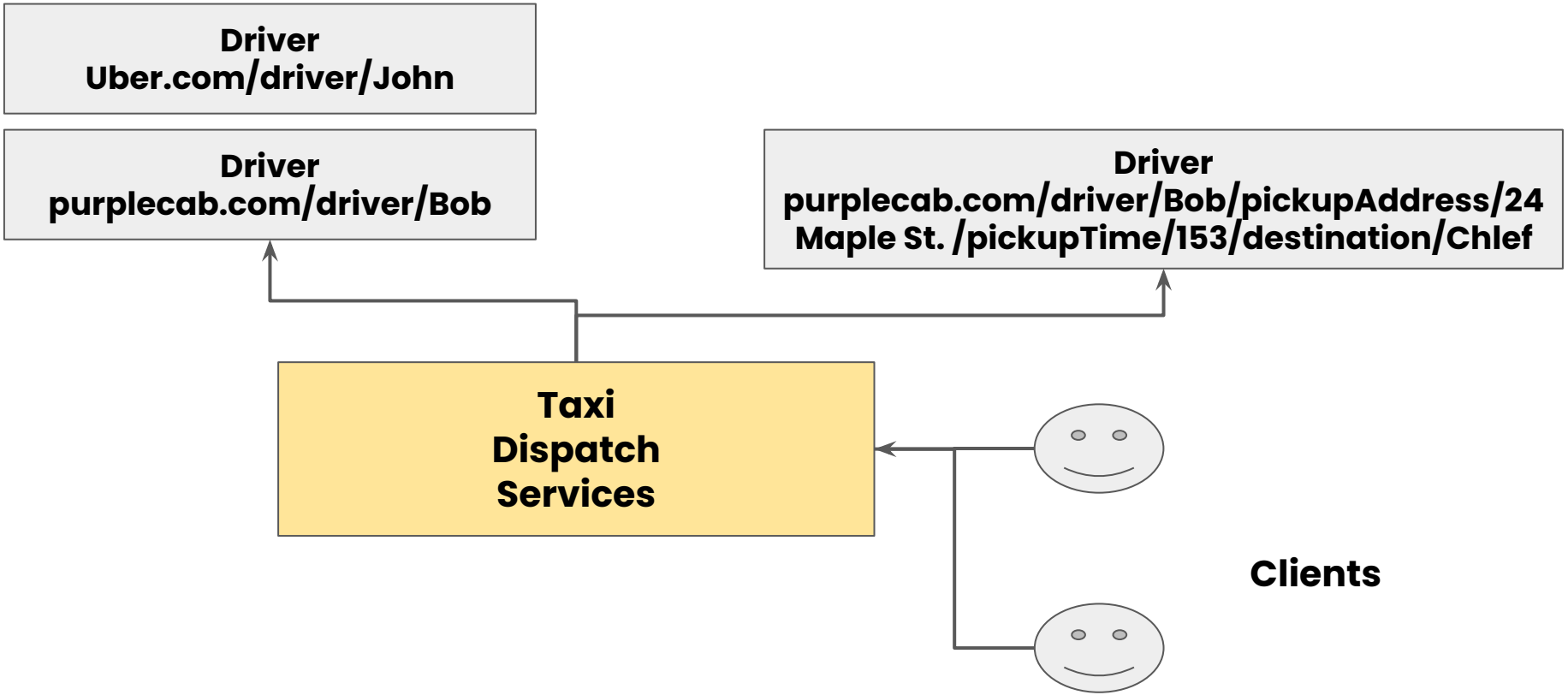


Fig 14 - building an aggregator for many taxi dispatch services

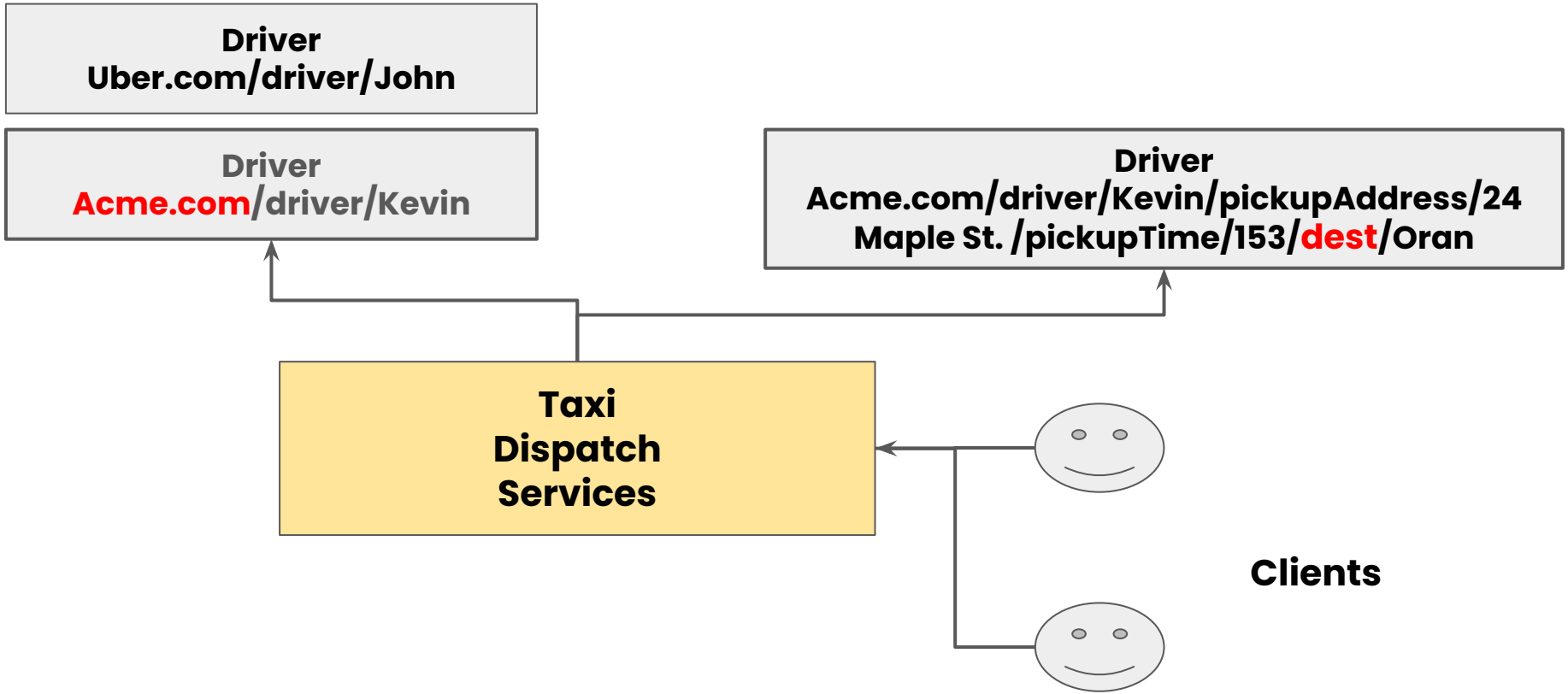
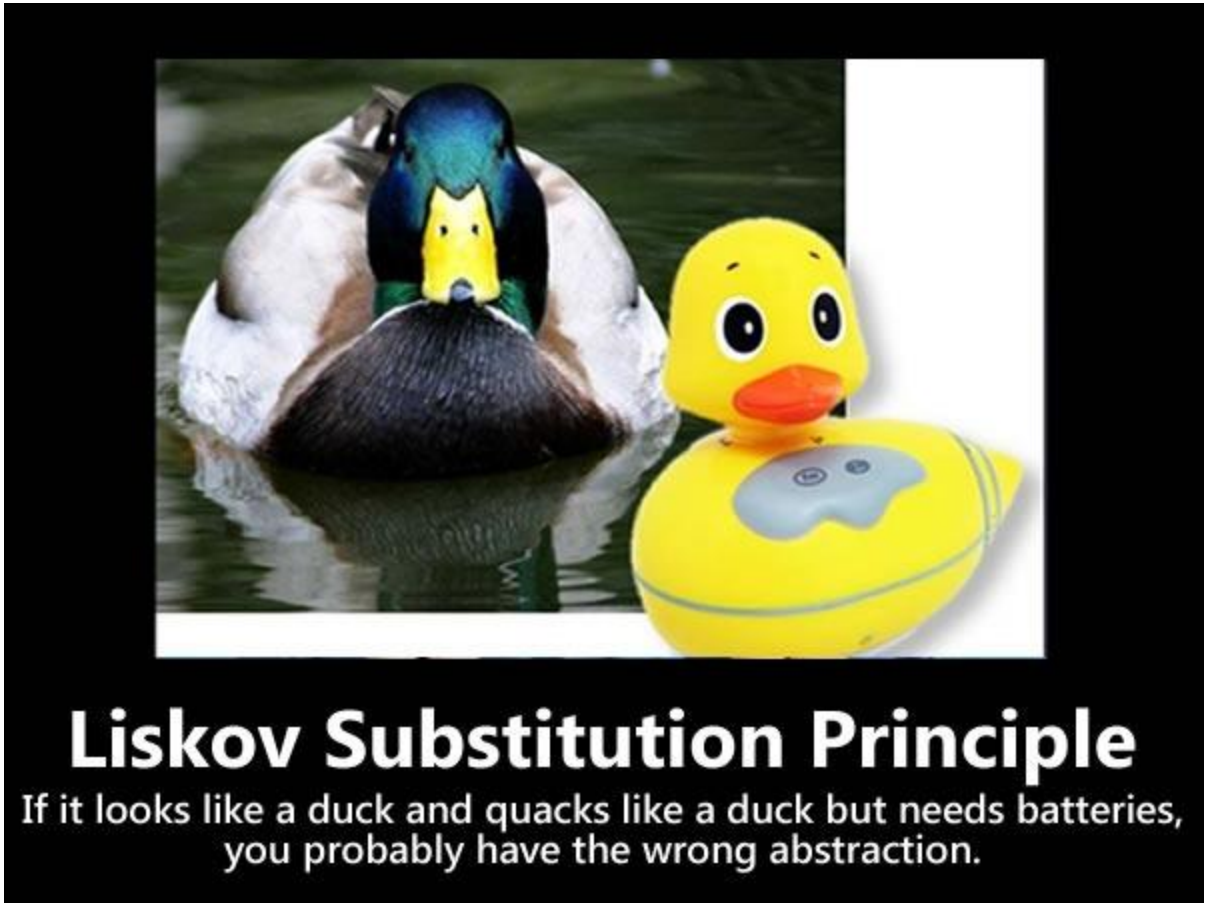


Fig 15 - Violation of LSP taxi dispatch services

Result

```
if (driver.getDispatchUri().startsWith("acme.com"))...
```

Our architect will have to add a significant and complex mechanism to deal with the fact that the interfaces of the restful services are not all substitutable.


Liskov Substitution Principle
If it looks like a duck and quacks like a duck but needs batteries,
you probably have the wrong abstraction.

SOLID

Interface Segregation (ISP)

2.4 Interface Segregation Principle

- The Interface Segregation Principle (ISP) derives its name from the diagram problem shown in Figure 16

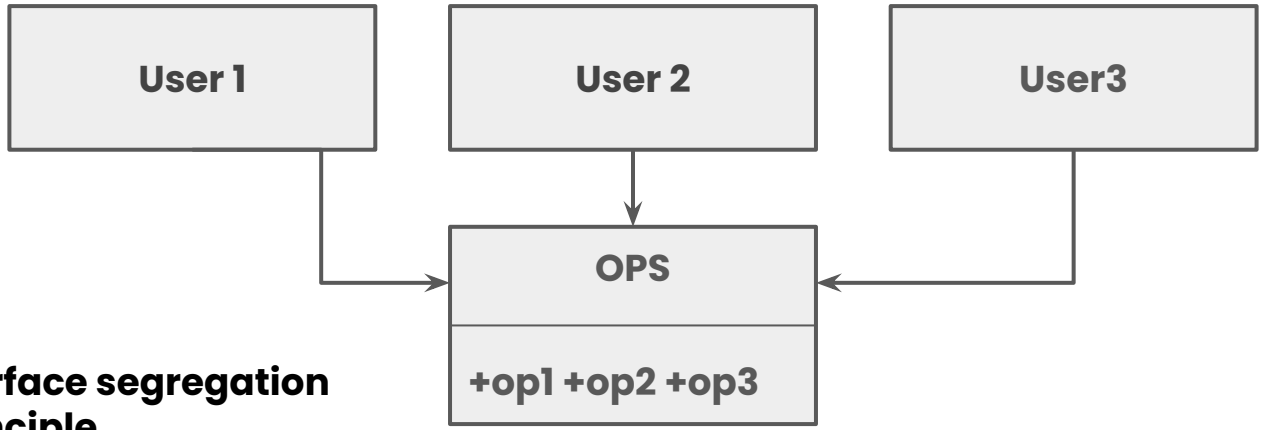


Fig 16 - The interface segregation principle

- Let’s assume that User1 uses only op1, User2 uses only op2, and User3 uses only op3.

- The source code of User1 will probably inadvertently depend on op2 and op3, even though it doesn’t call them.

- In language like Java this dependence means that a change to the source code of op2 in OPS will force User1 to be recompiled and redeployed, even though nothing that it cared about has actually changed.

ISP splits interfaces that are very large into smaller and more specific ones so client will only have to know about the methods that are of interest to them

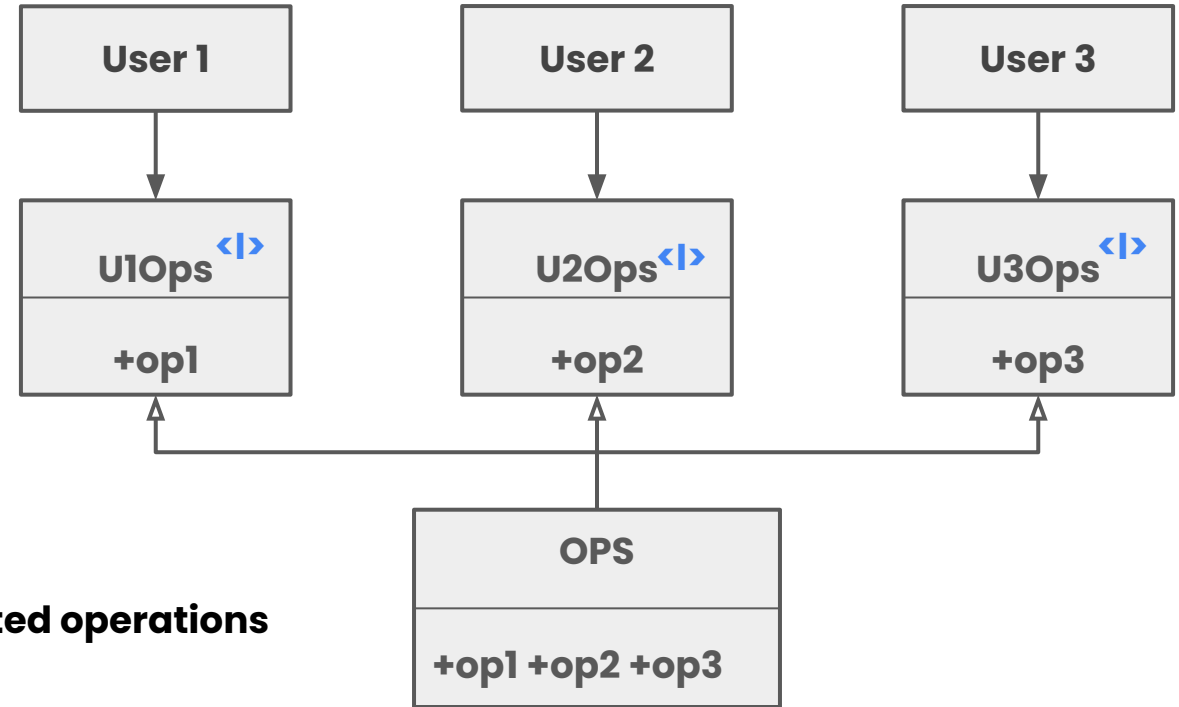
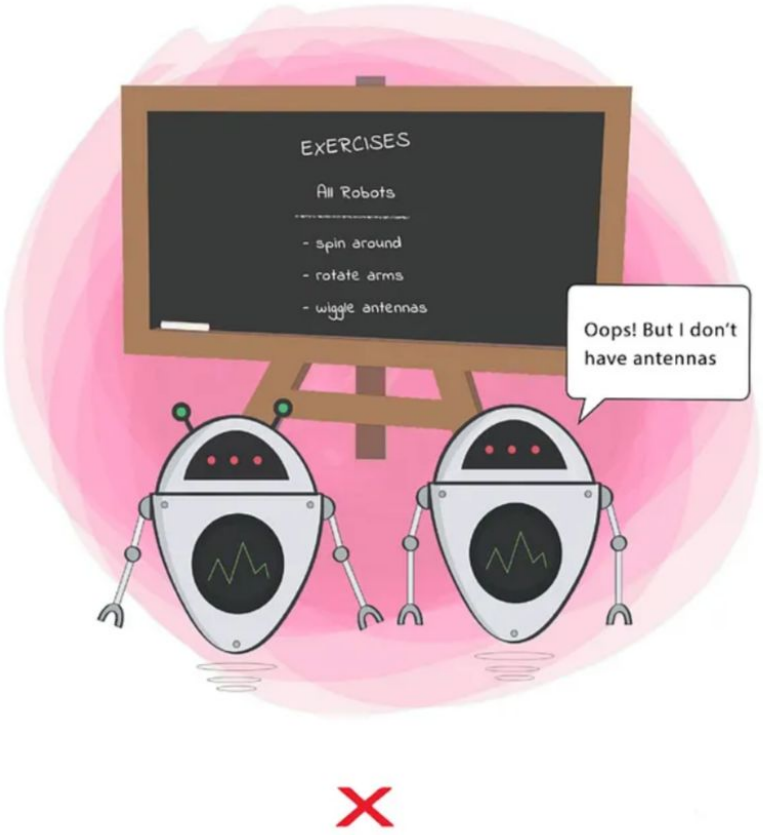


Fig 17 - Segregated operations

Let take a look at some examples of violating it



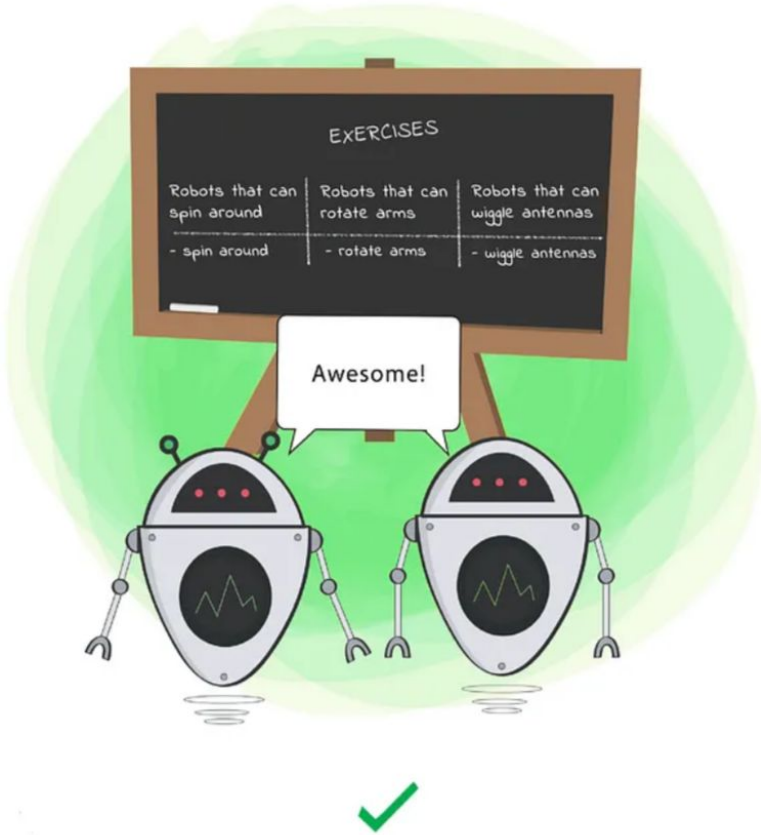


Fig 18 - Interface segregation principle Robots Example

Fig 19 – Interface segregation principle
USB Example

We do not want to force our clients to use an interface that contains functions or methods that they don’t use easily enough.

ISP Code Examples

ISP And Architecture

Fig 20 - A problematic architecture

Depending on something that carries baggage that you don’t need can cause you troubles that you didn’t expect.

SOLID

Dependency Inversion (DIP)

2.5 Dependency Inversion Principle

- **The Dependency Inversion Principle (DIP) tells us that the most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions.**

- **this means that the use, import, and include statements should refer only to source modules containing interfaces, abstract classes, or some other kind of abstract declaration. Nothing concrete should be depended on.**

- **Clearly, treating this idea as a rule is unrealistic, because software systems must depend on many concrete facilities. For example, the String class of java**

- **It is the volatile concrete elements of our system that we want to avoid depending on. Those are the modules that we are actively developing, and that are undergoing frequent change.**

DIP means that high level modules should not depend on low level modules. Both should depend on abstraction

Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

Let take a look at some examples of violating it

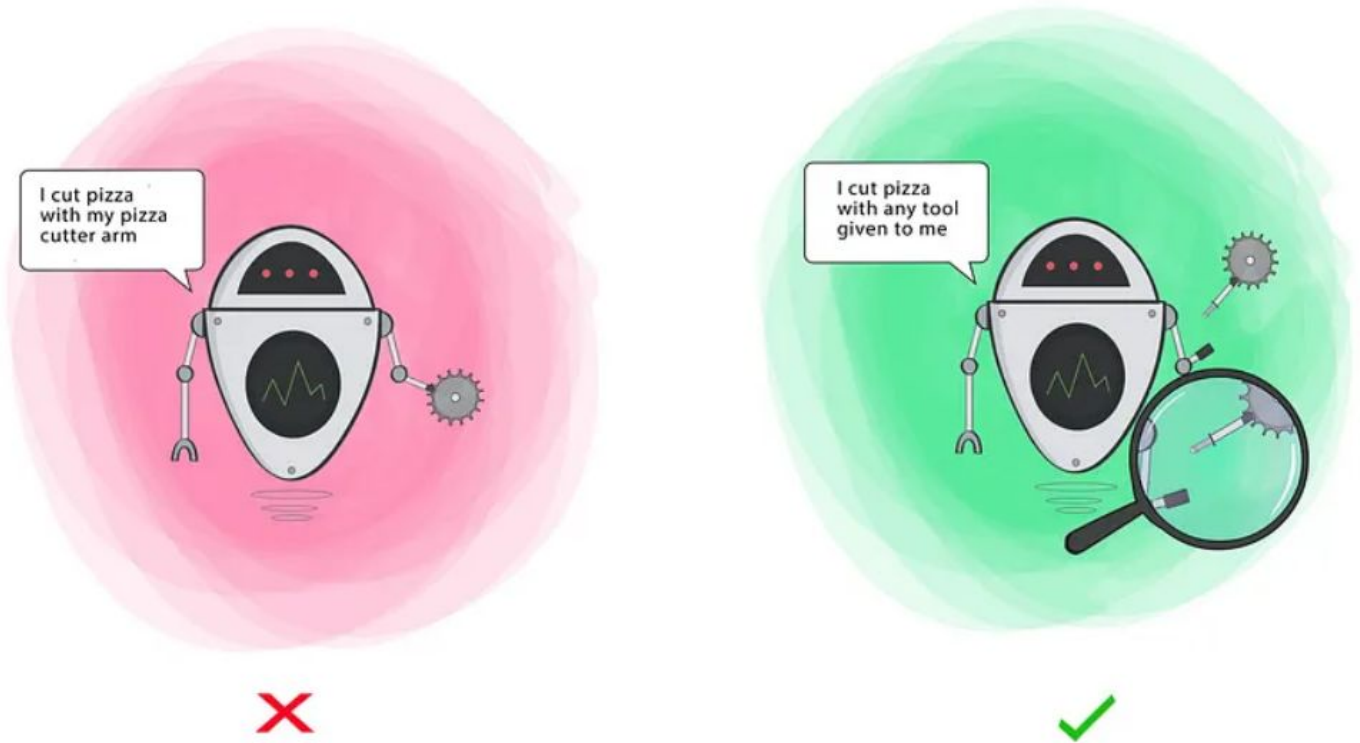


Fig 21 - Dependency Inversion principle Robots Example

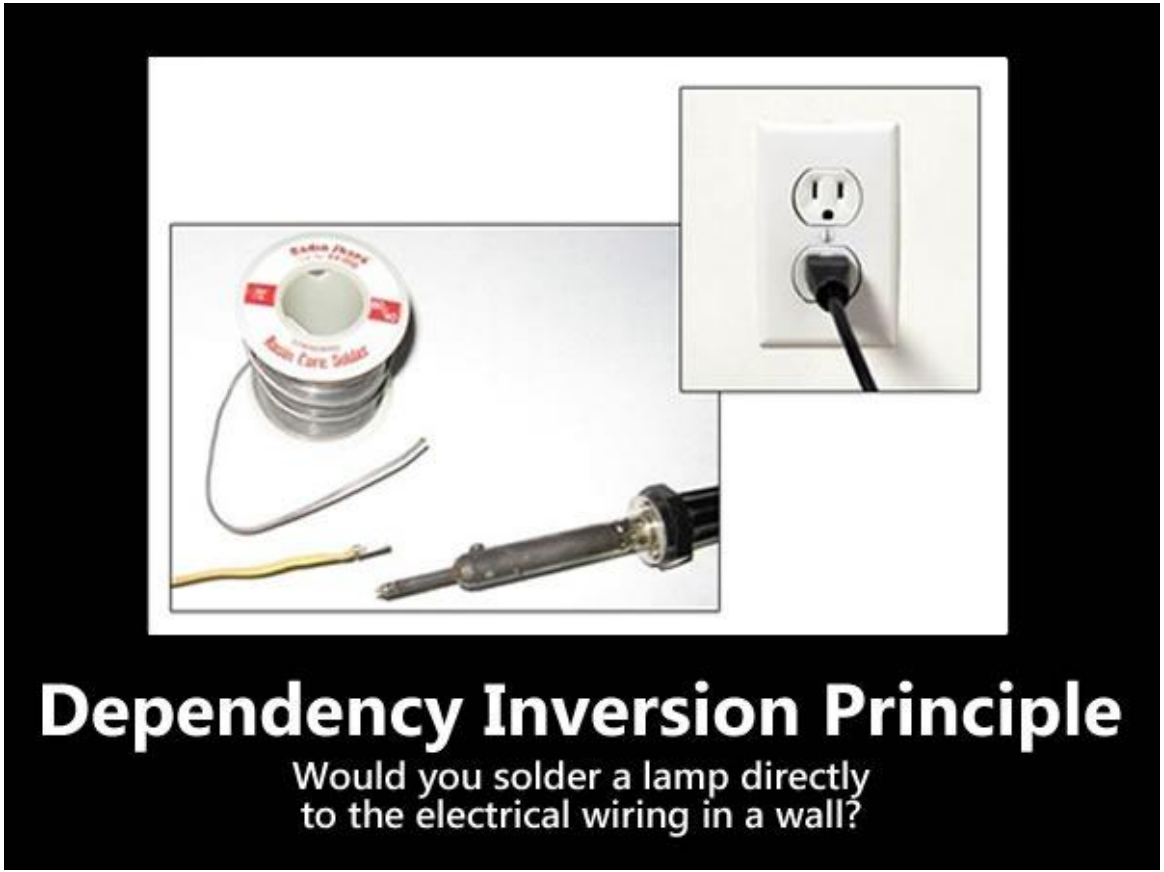


Fig 22 - Dependency Inversion principle
Lamp example

DIP Code Examples

- **Every change to an abstract interface corresponds to a change to its concrete implementations. Conversely, changes to concrete implementations do not always, or even usually, require changes to the interfaces that they implement.**

- **Indeed, good software designers and architects work hard to reduce the volatility of interfaces. They try to find ways to add functionality to implementations without making changes to the interfaces**

The implication, then, is that stable software architectures are those that avoid depending on volatile concretions, and that **favor the use of stable abstract interfaces**

Conclusion

Thank you
Any Questions ?