SOLID Design Principles

Dr. Rodrigo Spínola

Agenda

- **Motivation:** minimize cost of change

S ingle responsibility principle

O pen/closed principle

L iscov substitution principle

I nterface segregation principle

D ependency inversion principle

Classes are SOLID, Methods are SOFA

- **SOFA:** methods are <u>s</u>hort, do <u>o</u>ne thing, have <u>f</u>ew arguments, single level of <u>a</u>bstraction

SOLID concerns itself with designing classes, assuming the methods are SOFA

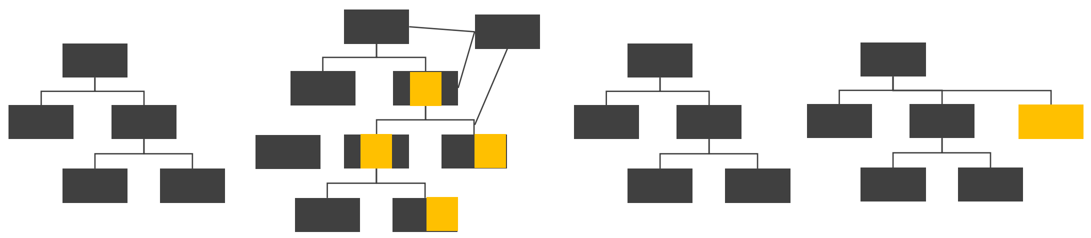
Single Responsibility Principle

- A class should have one and only one reason to change
  - Each responsibility is a possible axis of change
  - Changes to one axis shouldn't affect others
- E.g., a user is a moviegoer, and an authentication principal, and a social network member, … etc. Bad!!!
  - Really big class files are a tipoff
    - Lines of code is a course metric to detect this

Open/Closed Principle

- Classes should be **open** for extension, but **closed** for source modification

Usual way | **OCP**



Starting code base

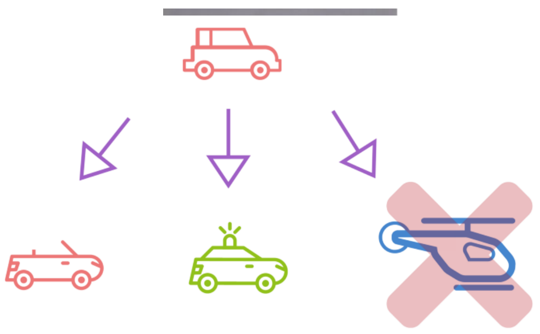
Changes implemented
Yellow = code changed

Starting code base

Change design to make room for new change

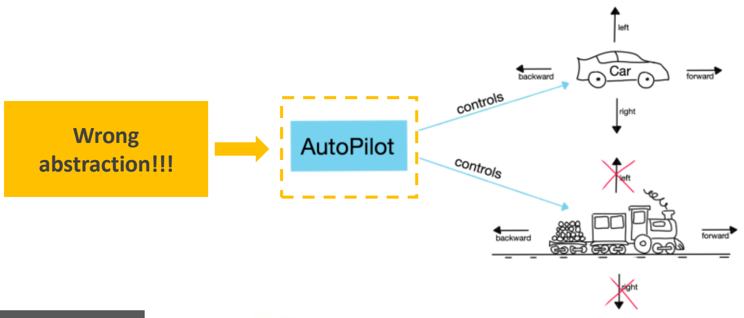
Liskov Substitution Principle

- Attributed to ACM Turing Award winner Barbara Liskov
  - pioneered many ideas in OOP
- “A method that works on an instance of type T, should also work on any subtype of T”

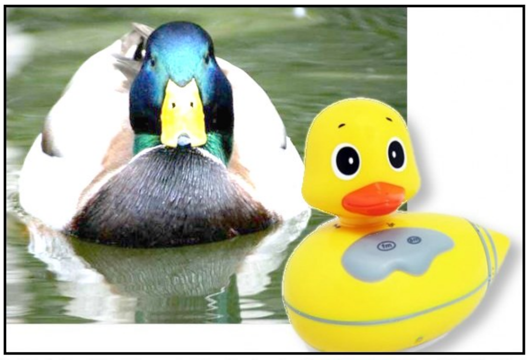


Liskov Substitution Principle

This principle is meant to guide the **design process of classes within hierarchies** with the main concern being **not** to create child classes that **have less and less in common with the parent**, **but** that **only require some methods from the parent class**.



Liskov Substitution Principle

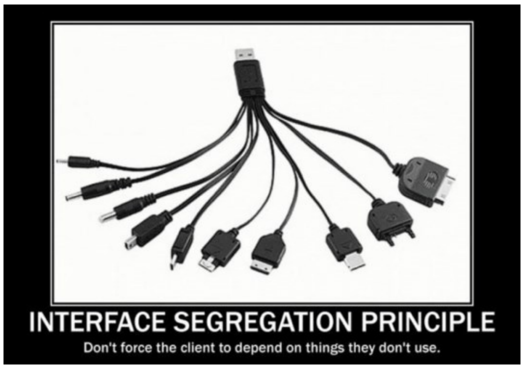


LISKOV SUBSTITUTION PRINCIPLE

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

Interface Segregation Principle

- ISP states that interface creation should be as refined as possible in the sense that all included **method declarations** in interfaces should be logically bound to the **same context** and should contain **only related functionality**
- Breaking this principle is clear when we use interfaces for which classes only implement some methods and leave other methods unimplemented
- You can think of this principle as "the Single Responsibility Principle for Interfaces"


INTERFACE SEGREGATION PRINCIPLE
Don't force the client to depend on things they don't use.

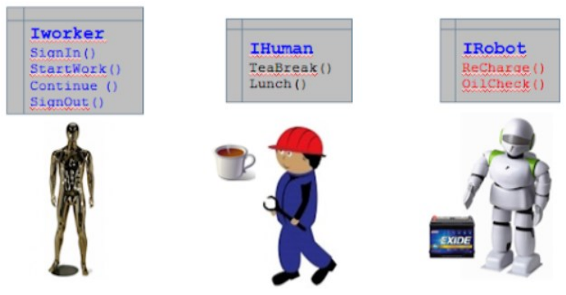
Interface Segregation Principle

- Imagine we have an interface with three methods, a class that provides an implementation for all three methods, and a newer class that only provides implementations for two of the methods
- When other developers will use the second implementation, they would normally assume that all methods are correctly implemented and will have an unpleasant surprise when it turns out they have been using an empty method in their code

Another example…

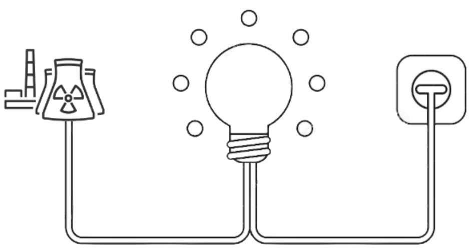
Clients should not be forced to depend on interfaces that they don't use!

Segregate your interfaces!



Dependency Inversion Principle

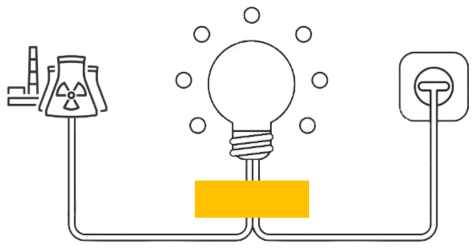
- **Problem:** High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features


Dependency Inversion Principle

When knowing how things work becomes a burden

Dependency Inversion Principle

- **Solution:** introduce an abstraction that decouples the high-level and low-level modules from each other:
    1. High-level modules should not depend on low-level modules. Both should depend on abstractions
    2. Abstractions should not depend on details. Details should depend on abstractions



Dependency Inversion Principle

When knowing how things work becomes a burden

Dependency Inversion Principle

The DIP does not just change the direction of the dependency. It **splits the dependency between the high-level and low-level modules** by introducing an **abstraction between them**.

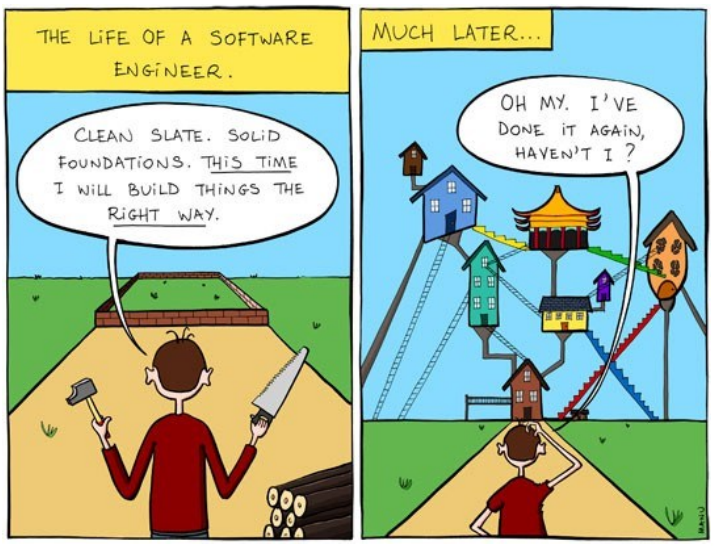
- In the end, you get two dependencies:
    - the high-level module depends on the abstraction, and
    - the low-level depends on the same abstraction.

SOLID is not absolute

- The SOLID design principles provide advice
  - Keep it in mind
  - Know when to refactor code
  - Know when not to refactor, when specific advice may not apply

TDresearchteam
Technical Debt Research Team

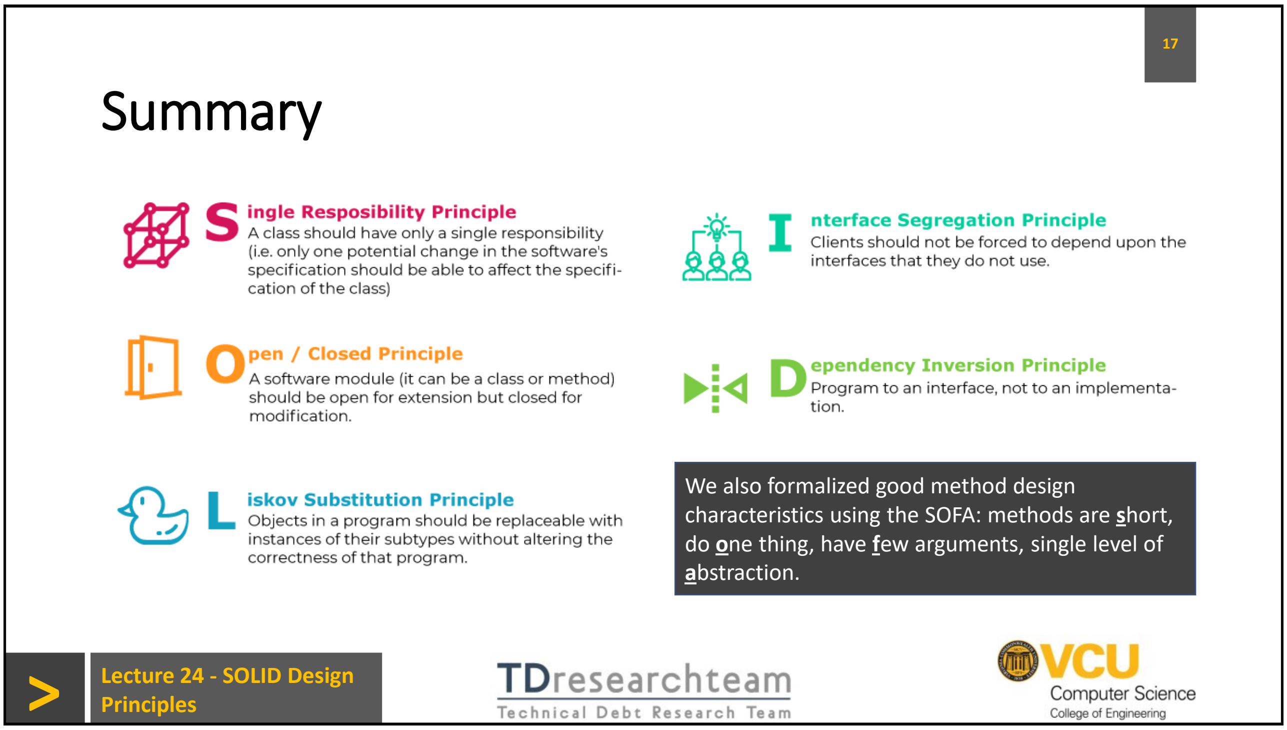
VCU Computer Science
College of Engineering



THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?

TDresearchteam
Technical Debt Research Team

VCU Computer Science
College of Engineering



17

Summary

S ingle Resposibility Principle
A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)

O pen / Closed Principle
A software module (it can be a class or method) should be open for extension but closed for modification.

L iskov Substitution Principle
Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

I nterface Segregation Principle
Clients should not be forced to depend upon the interfaces that they do not use.

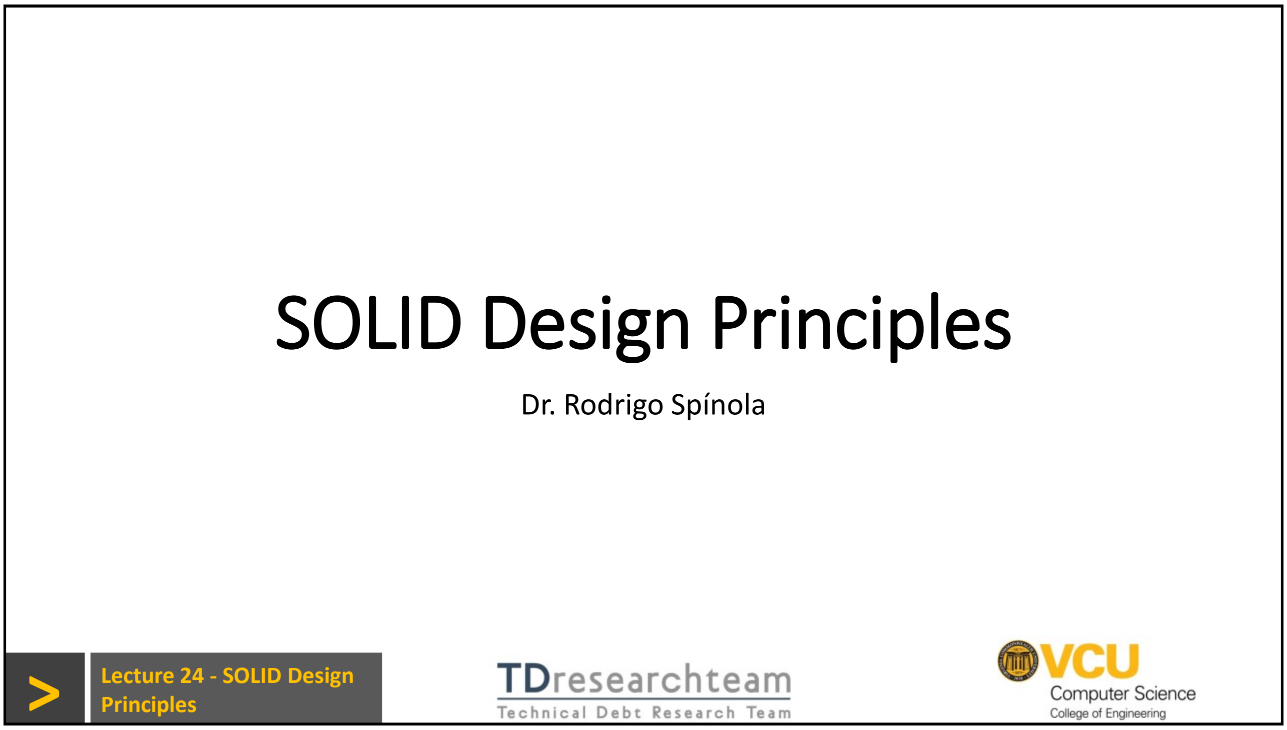
D ependency Inversion Principle
Program to an interface, not to an implementation.

We also formalized good method design characteristics using the SOFA: methods are <u>s</u>hort, do <u>o</u>ne thing, have <u>f</u>ew arguments, single level of <u>a</u>bstraction.

Lecture 24 - SOLID Design Principles

TDresearchteam Technical Debt Research Team

VCU Computer Science College of Engineering



SOLID Design Principles

Dr. Rodrigo Spínola

> Lecture 24 - SOLID Design Principles

TDresearchteam
Technical Debt Research Team

VCU
Computer Science
College of Engineering