COMS W4156 Advanced Software Engineering (ASE)

October 27, 2022

Agenda

1. Bug Finder demo
2. Design Principles

Bug Finder demo: Ria Gupta

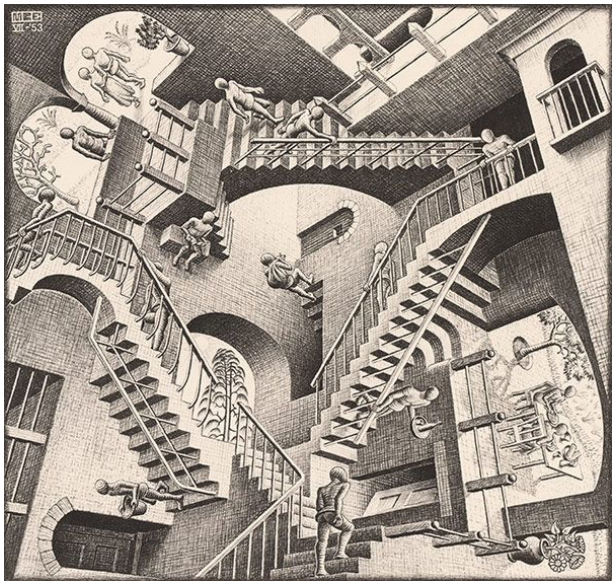
Read the Docs

Free Sonar products for open-source projects

Gitlab community edition

Spotbugs manual (free for everyone but Java-specific)

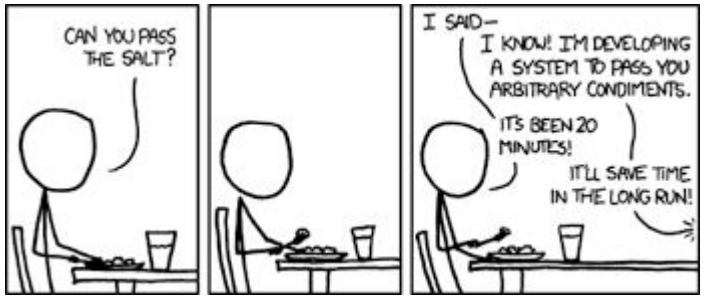
Design



The design manifest in this lithograph is reminiscent of spaghetti code: a great idea for art, a really bad idea for software engineering

Bad Software Design

- Rigid - Hard to change the code because every change affects too many other parts of the system.
- Fragile - When you make a change to some part of the code, unexpected parts of the system break.
- Immobile - Hard to reuse some of the code in another application because it cannot be disentangled from the current application.



You Aren't Gonna Need It (YAGNI): do the simplest design that could possibly work for the functionality needed NOW

Good Software Design

- Functional - Focused on core functionality
- Robust - Resistant to changes in environment and failures
- Measurable - It should be possible to see how well the code is doing outside of a test environment.
- Debuggable - Maintains logs of what its done and is doing.
- Maintainable - Consistent styling, good comments, modular. Easy to make changes to parts of the software without breaking other parts, particularly core functionality.
- Reusable - Only if some/all of the software is realistically plausible for other products.
- Extensible - Should be written with extension in mind (contradicts yagni?).

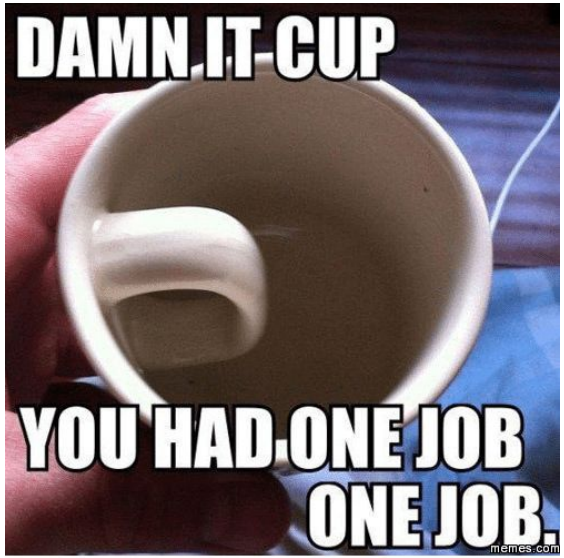
“SOLID” Software Engineering Design Principles

Design Principle = general technique to avoid bad design (or, phrased positively, create good designs)

1. **S**ingle Responsibility
2. **O**pen/Closed
3. **L**iskov Substitution
4. **I**nterface Segregation
5. **D**ependency Inversion

Design Principle != Design Pattern

Single Responsibility Principle (SRP)



Every module (or class) should have a single responsibility,
and all its methods should contribute to fulfilling that responsibility

Responsibility may be too fine-grained, e.g., the classes of CRC cards have multiple responsibilities. Perhaps better to call it the Single Purpose Principle (SPP)

problem: If a class has multiple purposes, then it is more likely to be affected by a requirements change: It knows too much

solution: Break up the functionality into multiple classes, where each individual class has only one purpose. Leads to more smaller classes

Greater cohesion, lower coupling, fewer test cases

SRP

Sanity check 1: Can you describe the component's purpose in one sentence without using the word "and"?

Sanity check 2: Does it make sense to say "The <class name> <method name> itself" for each method? (after fixing grammar) Example

Sanity check 3: Does the component know/do too much? Recall code smell "bloaters".

Open-Closed Principle (OCP)

Code units should be “open for extension and closed for modification”, to allow change without modifying any existing code that works because change introduces new bugs

Seemingly contradicts refactoring, but most refactoring operations (discussed next week) seek to move out parts of the code to leave behind an OCP code unit

OCP

Backwards compatibility

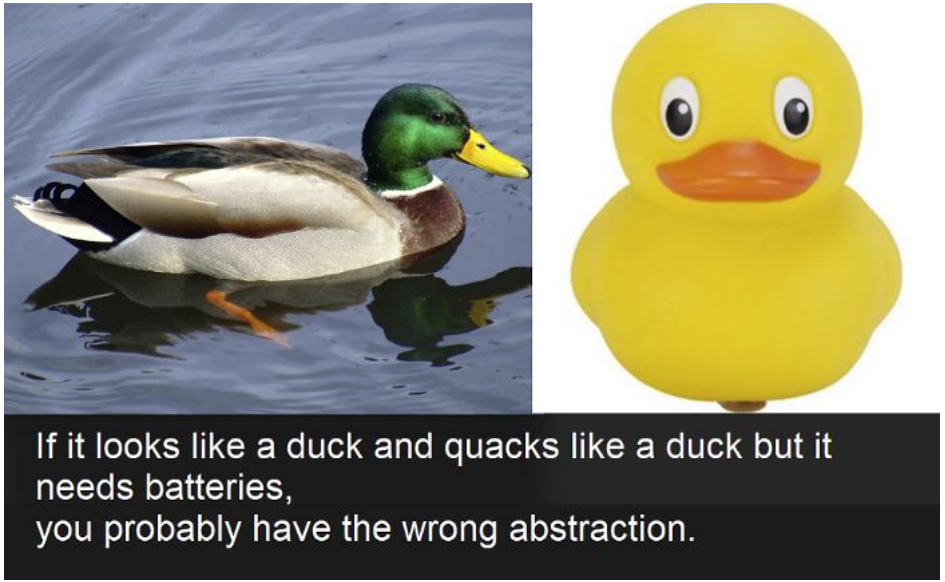
Since requirements change, we need to be able to change existing program behavior

- Change the behavior of an existing class by deriving a new subclass (inheritance)
- Change the behavior of an existing class by composition with a new class (delegation)
- Change the behavior of an existing method by adding a wrapper (adapter) around that function
- Change the behavior of a library or service by adding new API functions or a new implementation of the same interface

Liskov Substitution Principle (LSP)

Extends the inheritance variant of OCP to require that subtypes must be substitutable for their base types (polymorphism)

If class *A* is a subtype of class *B*, then we should be able to replace *B* with *A* without disrupting the behavior of our program



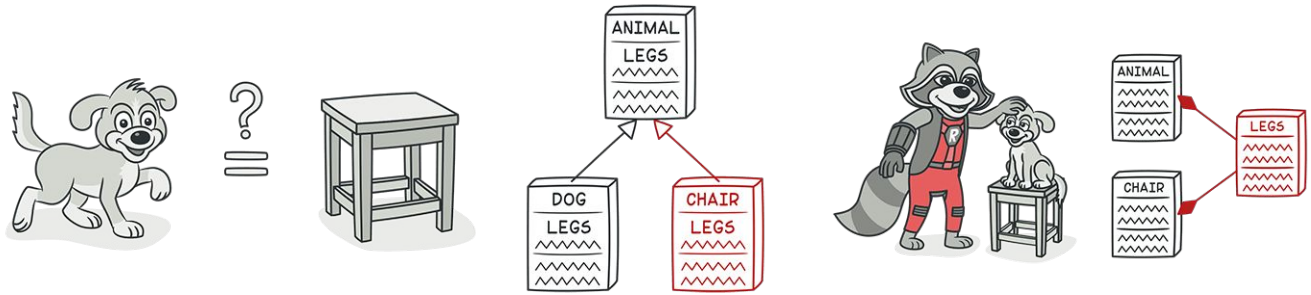
LSP

Do inherited methods make sense for instances of the derived class? Do overridden methods take the same parameter types and return the same types as the base class? Do they have side-effects on the same data?

If not, then we are not actually inheriting - instead should delegate any common functionality, and if there is no common functionality then there's no relationship at all so why pretend?

Code already using references to the base classes should be able to use instances of the derived classes without being aware of the switch

Example



Design by Contract

The contract of a method informs the author of a class about the behaviors that she can safely rely on

The contract could be specified by declaring preconditions and postconditions for each method: The preconditions must be true in order for the method to execute. On completion, after executing the method, the method guarantees that the postconditions are true

Derived classes need to fulfill the same contract.

Interface Segregation Principle (ISP)

Components that implement an interface should only have to provide implementations of the methods relevant to them

Clients should not be forced to depend upon interfaces they do not use

Large interfaces should be split into smaller ones

ISP

Analogous to SRP: If an interface has multiple purposes, then it is more likely to be affected by a requirements change

Break up into multiple interfaces, where each individual interface has only one purpose. Leads to more smaller interfaces

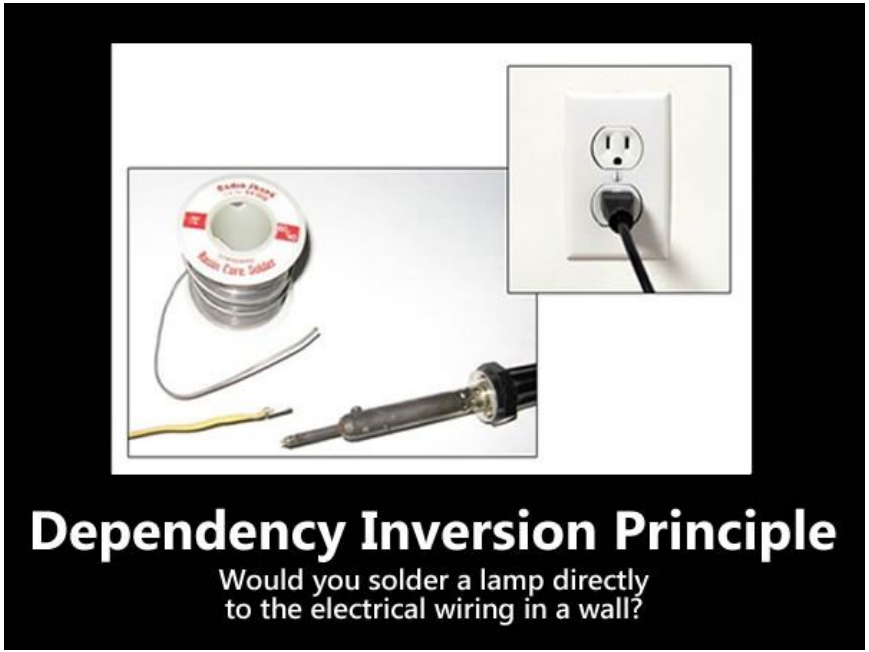
Example

Dependency Inversion Principle (DIP)

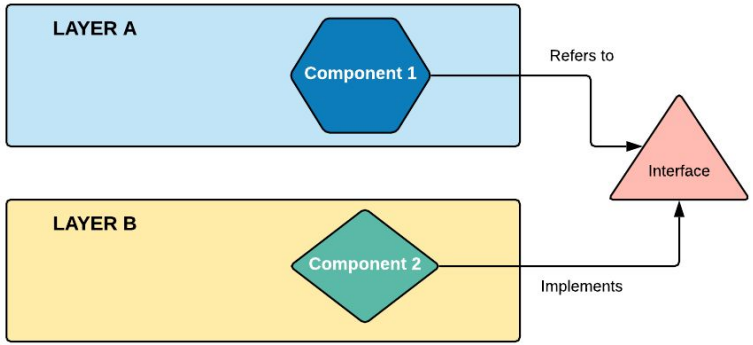
High-level modules should not depend on low-level modules, which makes it hard to reuse the high-level modules

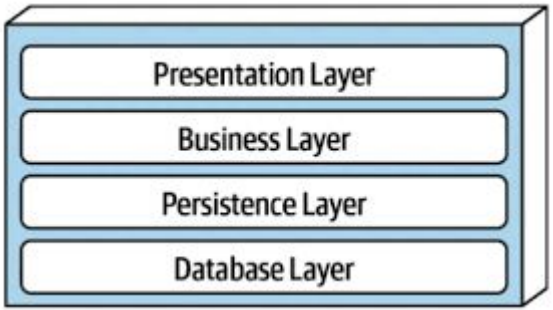
Both should depend on abstractions

Abstractions should not depend on details - details should depend on abstractions



DIP





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

This seemingly contradicts the layered architectural style, where each layer builds on the layer below - the contradiction is resolved by depending on an abstraction of the layer below, not the details of the lower layer

“SOLID” Software Engineering Design Principles

1. **S**ingle Responsibility
2. **O**pen/Closed
3. **L**iskov Substitution
4. **I**nterface Segregation
5. **D**ependency Inversion

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.

Don’t Repeat Yourself principle (DRY)

Another D - SOLIDD?

Avoid duplicate code and redundant work, instead abstract out common code and put in a single location

Make sure that you only implement each feature and requirement once

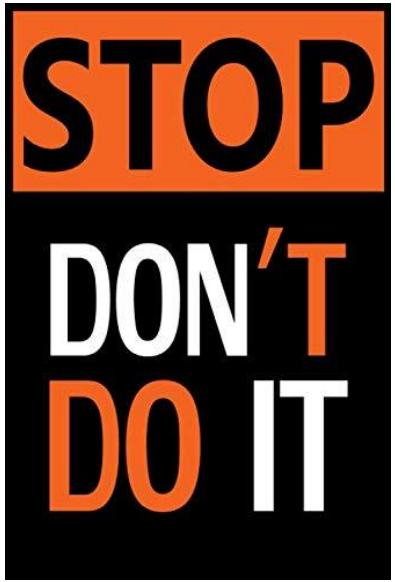
DRY is about putting a piece of functionality in one place vs. SRP is about making sure a class does only one thing

DRY

Sanity check: Are there two or more ways to fulfill the same use case or workflow?

But don’t go too far…

“ARID” = All Readability is Dead



Upcoming Assignments

First Iteration Demo due October 31, next Monday!

➔ You can keep coding and testing between first iteration submission and first iteration demo, but tag both separately! Also see github releases

First Individual Assessment
available 12:01am November 1, due 11:59pm November 4

Next Week

Any questions about Midterm Assessment?

Refactoring

Continuous Integration

github actions demo

Ask Me Anything