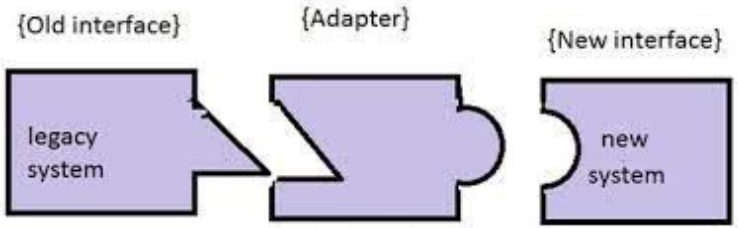
COMS W4156 Advanced Software Engineering (ASE)

October 7, 2021

shared google doc for discussion during class

How to get from CRC Cards to Code

First check whether any CRC “classes” can be implemented by already available components, including libraries and services. This might require adapters (more on adapters later on).

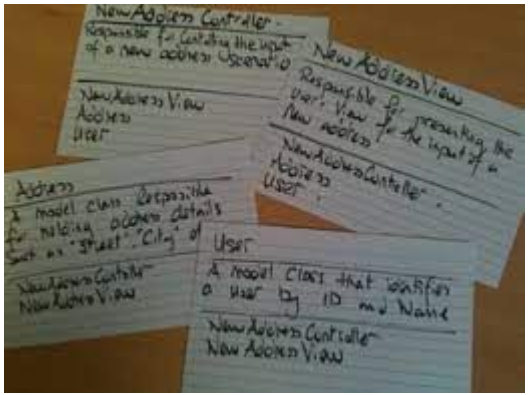


Then other CRC classes will typically be implemented by a class, module, file, etc. - the programming language’s modular unit

The responsibilities correspond to the modular unit’s methods, functions, procedures, subroutines, etc., with access to the collaborators via parameters, object or class fields, global variables, shared data structures, etc.

Note updating fields, global variables, shared data structures = “side-effects”

Turn over your CRC card



On the back, list:

Class or other component name (not necessarily same as on front, e.g., if reusing existing software)

Variables maintained by instances of the class - data as well as references to collaborators

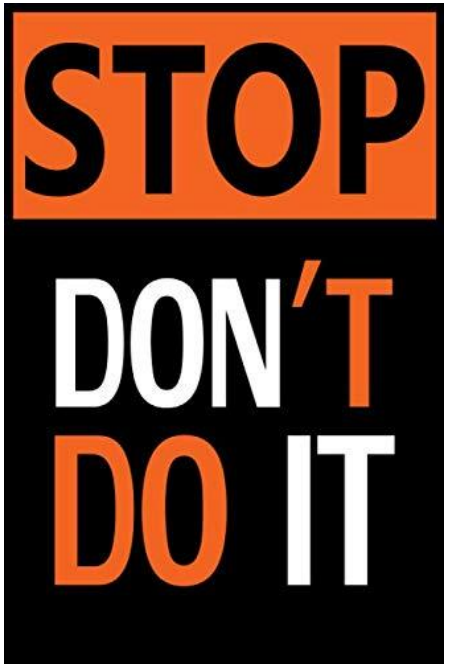
Methods that carry out the responsibilities - signature with name, parameter list, return types, *make sure to include each method's "side-effect" input/output data beyond parameters and return*

More on Side-Effects

Avoid when possible

But if your units have, or might have, side-effects (particularly beyond an object's own fields) - then unit testing needs to consider inputs from shared state and, even more significantly, outputs to shared state

Mocking of external shared state for unit and integration testing covered later on

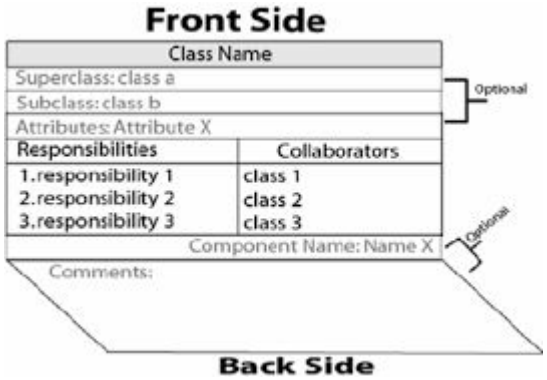


Backs of CRC Cards

The back of a CRC card is also commonly used for [entity-relationship (E-R) data modeling concepts]

Generalization (a kind of), Aggregation (has parts), other Associations

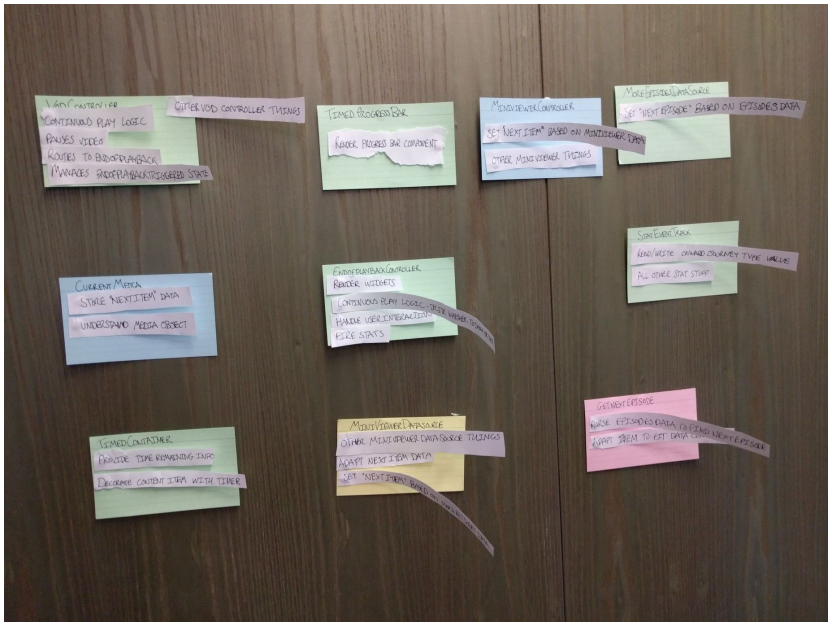
Also commonly used for comments



Debugging CRC Cards, Revisited

Again, animate the use cases - move the cards around on a table or board - to make sure every trigger is accounted for, every precondition is checked, every postcondition indeed becomes true, and every step in all basic, alternative and exception flows is accounted for

May help to represent actors that trigger or interact with responsibilities as “fake” CRC cards



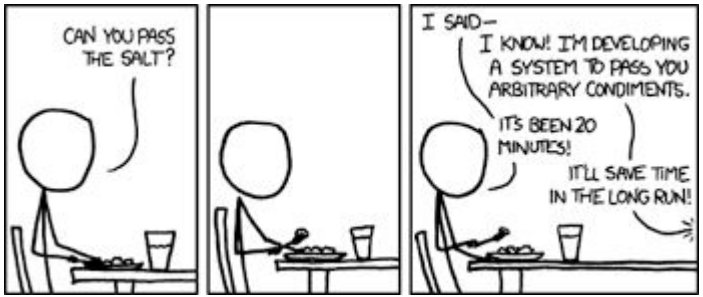
Apply Design Principles

For each extended CRC card (extended by the details on the back), consider design principles applicable to that component and/or its collaborations with other components

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

Bad design is:

- 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

“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

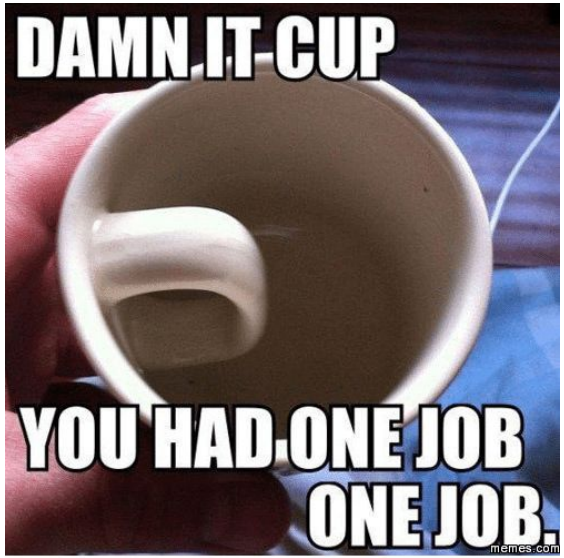
Some of these apply to any language, some assume OO

Single Responsibility Principle (SRP)

Every component should have a single responsibility, and all its methods should contribute to fulfilling that responsibility - move any methods that don't belong to another component

Seemingly contradicts the multiple "responsibilities" of CRC cards, which refer to finer granularity tasks that should all be part of the same focused single purpose.

Better to call it the Single Purpose Principle (SPP)



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

Change introduces new bugs “regression” (regression testing covered later on)

Seemingly contradicts refactoring, but most refactoring operations 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 composition with a new class (delegation)
- Change the behavior of an existing function (or procedure, subroutine, method, etc.) 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

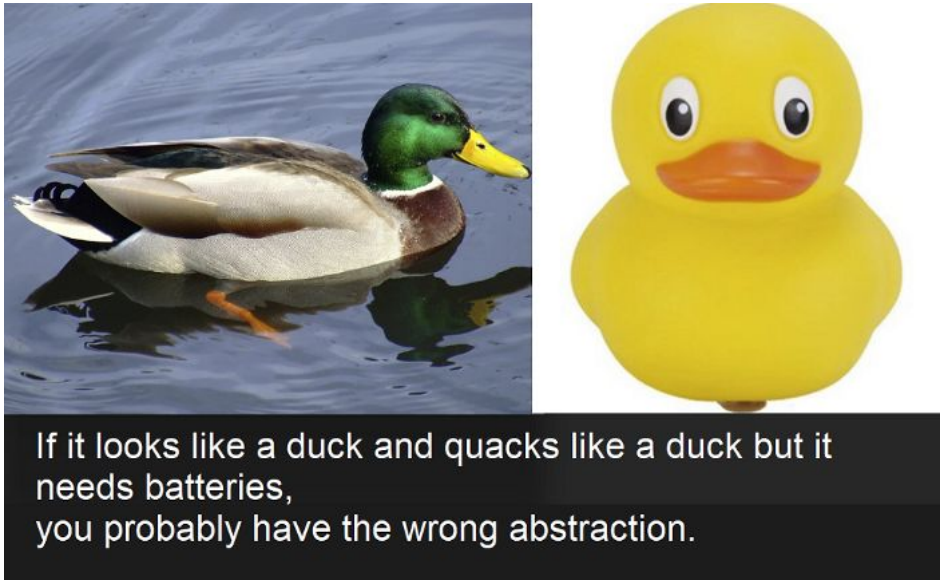
Once upon a time, the conventional way to change the behavior of an existing class was to derive a new subclass, i.e., inheritance Example

Notice CRC cards have composition (collaborators) but no inheritance - you can still implement using inheritance (in an OO programming language), but be careful about violating LSP and ISP principles!

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

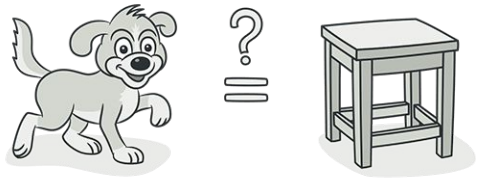


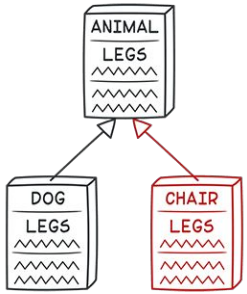
LSP

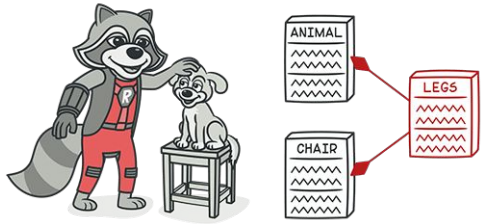
Do inherited methods make sense for instances of the derived class? Do overridden methods take the same parameter types and return the same types as the base class? Do they have side-effects on the same data? Example

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







Interface Segregation Principle (ISP)

CRC cards can be thought of as modeling interfaces instead of how those interfaces are implemented in code

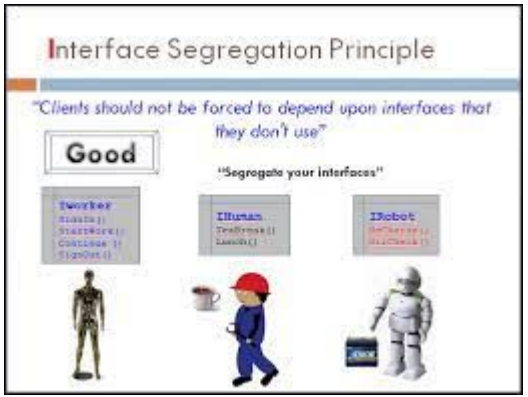
Large interfaces should be split into smaller ones

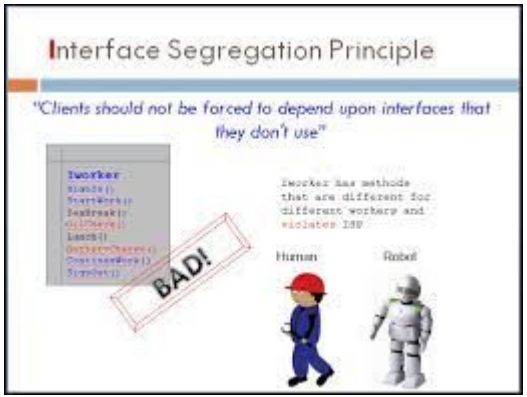
Components that implement an interface should only have to provide implementations of the methods relevant to them

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]



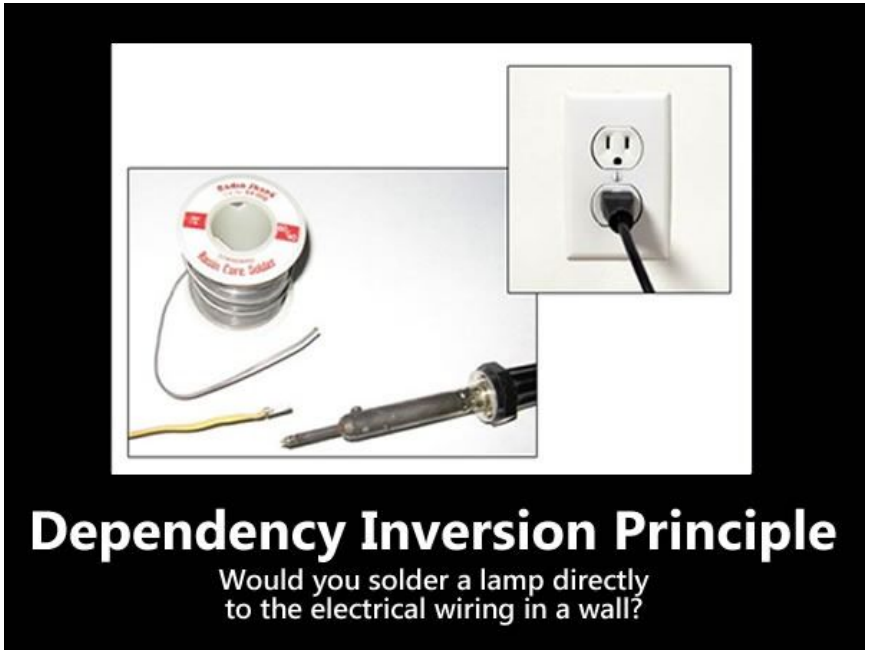


Dependency Inversion Principle (DIP)

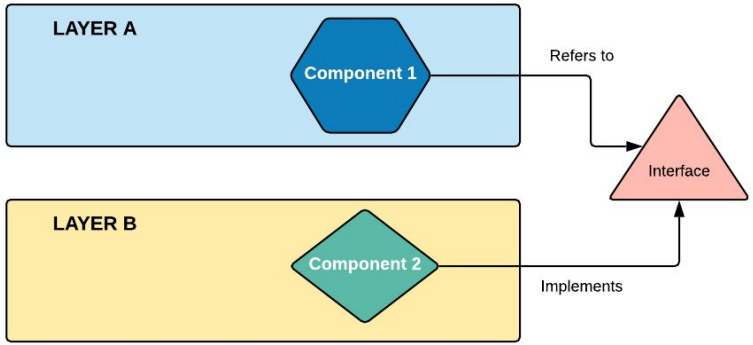
High-level modules should not depend on low-level modules, which makes it hard to reuse the high-level modules

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.

Don’t Repeat Yourself principle (DRY)

Another D - SOLIDD?

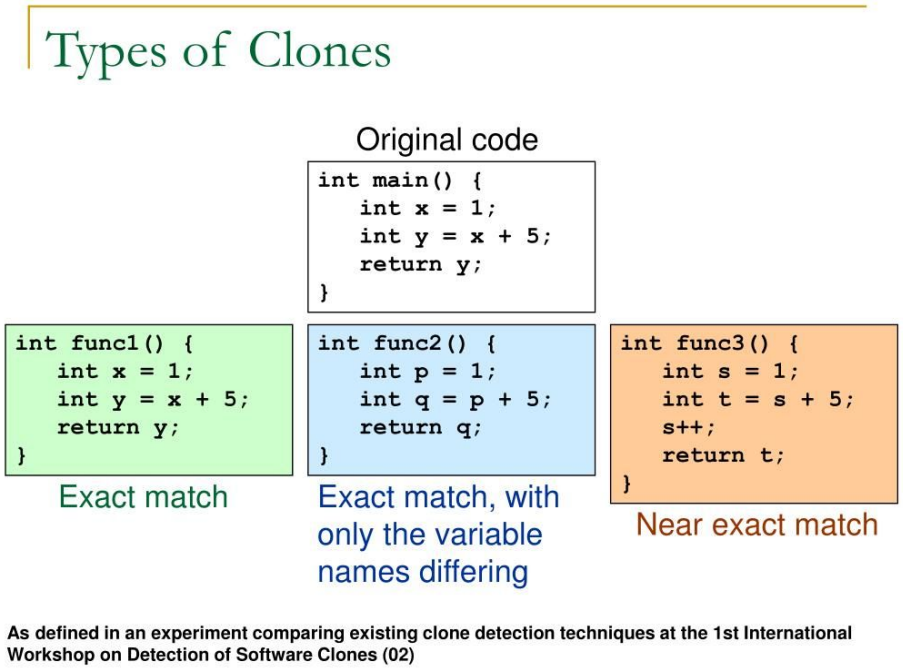
Avoid duplicate code and redundant work, instead abstract out common code and put in a single location

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: During CRC animation, are there two or more ways to fulfill the same use case or flow?



“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.

Team Project

Preliminary Project Proposal due October 13 (next Wednesday)

Your service has to implement some API, but it does not need to be a REST API, and could involve interacting with other services, RPC, mediation, callbacks, events, etc. - but no GUI

No GUI does not necessarily mean no UI, there might be an administrative console using CLI (command line interface), just make sure your service cannot be mistaken for an “app”

Individual Mini-Project part 2 and part 3 Deadlines Extended

Three parts:

1. Implementing a simple game graded, rubric posted here
2. Testing the game past due, grading in progress
3. Saving game state nominally due yesterday, but you can submit up to Saturday with no penalty