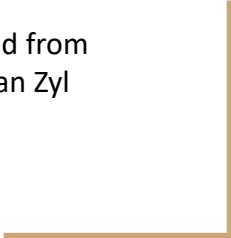




Object- Oriented Design

Tamlin Love

Shamelessly stolen and adapted from
Rylan Perumal and Terence van Zyl



Recap

- Overview of Software Testing
- Coverage and Heuristics for High Coverage
- Practical Aspects of Unit Testing
- Integration and System Testing

Today's Topics

- Object-Oriented Design
- Principles for Good Design
 - SOLID
 - Responsibility-Driven Design

OOP Reminder

- **Object-Oriented Programming (OOP)** is a paradigm based on objects
- **Object-Oriented Design (OOD)** is the process of using objects and object-oriented programming when designing a software solution.

Q: What's the object-oriented way to become wealthy?

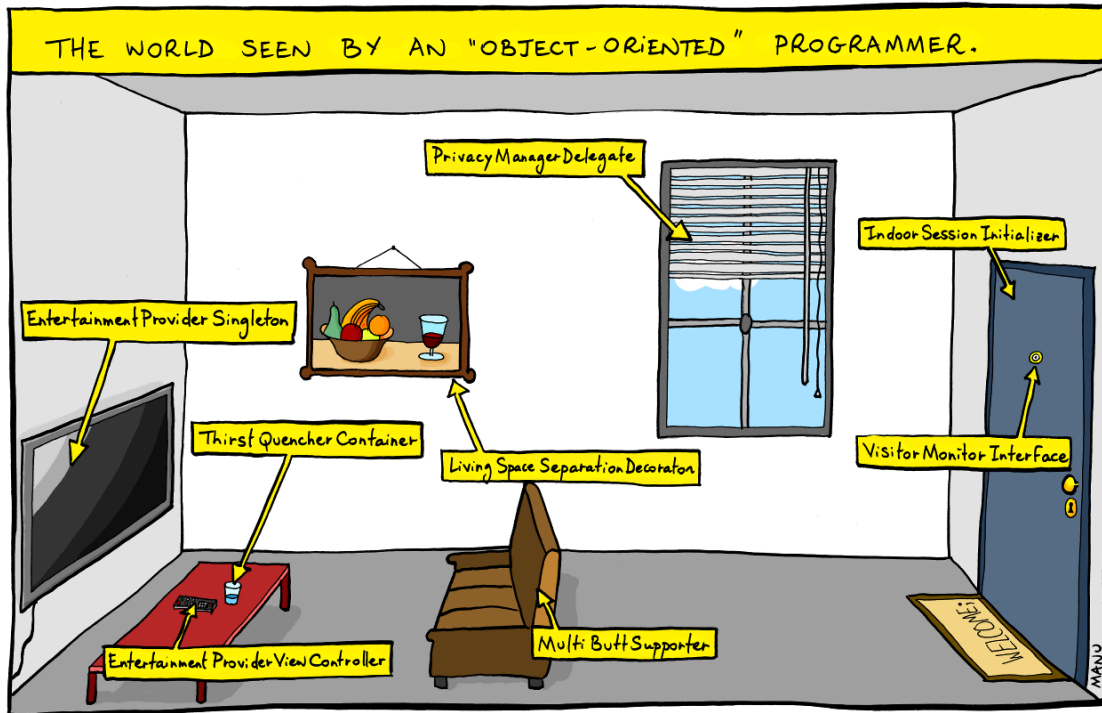
A: Inheritance

OOP Concepts

- **Encapsulation** - a tight coupling between data structures (i.e. attributes) and the methods that act on the data
- **Information Hiding** - the ability to protect some components of the object from external entities
- **Inheritance** - the ability for a class to extend or override functionality of another class.
- **Interface** - a definition of methods, and their signatures to manipulate an object.
- **Polymorphism** - the provision of a single interface to entities of different types

OOD Principles

- How do we use object-oriented design to build better systems?



OOD Principles - SOLID

- **Single responsibility principle**
 - There should never be more than one reason for a class to change.
 - i.e. Every class has a single responsibility
- **Open-closed principle**
 - Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification
- **Liskov substitution principle**
 - Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
- **Interface segregation principle**
 - Many client-specific interfaces are better than one general-purpose interface.
- **Dependency inversion principle**
 - Depend upon abstractions, not concretions.

OOD Principles - SOLID

- **Single Responsibility Principle (SRP)**
 - There should never be more than one reason for a class to change.
 - If we can divide the functionality of a system into a set of responsibilities, each responsibility should be handled by a single class

SOLID Principle: "Classes should have a single responsibility."

My classes:



OOD Principles - SOLID

- Responsibilities:
 - **Knowing responsibility**
 - Memorizing data or references, such as data values, data collections, or references to other objects, represented as an attribute
 - **Doing responsibility**
 - Performing computations, such as data processing, control of physical devices, etc., represented as a method
 - **Communicating responsibility**
 - Communicating with dependencies to delegate work, represented as message sending (method invocation)

OOD Principles - SOLID

- **Open-Closed Principle**

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification
- i.e. we should be able to extend the behaviour of a software entity without modifying its source code
- How?
 - Inheritance – leave base class alone and write a derived class to extend behaviour
 - Polymorphism – supply any object with the same interface, regardless of type
 - Interfaces are very useful here

OOD Principles - SOLID

- **Liskov Substitution Principle (LSP)**
 - Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
 - Created by Barbara Liskov (1988), formally stated as:
 - Let $\phi(x)$ be a property provable about objects x of type T
 - Then $\phi(y)$ should be true for any object y of type S , where S is a subtype of T
 - In other words, if using a subtype (e.g. inheritance) doesn't break your code, you're doing LSP!

OOD Principles - SOLID

- A typing rule (R) is **covariant** if it preserves an ordering of types
 - e.g. If $A \leq B$, then $R(A) \leq R(B)$
- Example: covariant return type
 - $\text{Dog} \leq \text{Animal}$
 - $\text{DogFood} \leq \text{Food}$
 - $\text{Dog.seekFood()} \leq \text{Animal.seekFood()}$
 - Therefore, the return of *seekFood* is covariant

```
1  public class Animal {  
2  
3      protected Food seekFood() {  
4  
5          return new Food();  
6      }  
7  }  
8  
9  public class Dog extends Animal {  
10  
11      @Override  
12      protected DogFood seekFood() {  
13  
14          return new DogFood();  
15      }  
16  }
```

OOD Principles - SOLID

- A typing rule (R) is **contravariant** if it reverses an ordering of types
 - e.g. If $A \leq B$, then $R(A) \geq R(B)$
- Example: contravariant parameter type
 - $\text{Animal} \leq \text{Object}$ (Java)
 - $\text{CatShelter} \leq \text{AnimalShelter}$
 - input of `CatShelter.putAnimal` \geq input of `AnimalShelter.putAnimal`
 - Therefore, the input of *putAnimal* is contravariant
 - Note: most languages would treat this as method overloading

```
1  class AnimalShelter {  
2  
3      void putAnimal(Animal animal) {  
4          //...  
5      }  
6  }  
7  
8  class CatShelter extends AnimalShelter {  
9      void putAnimal(Object animal) {  
10         // ...  
11     }  
12 }
```

OOD Principles - SOLID

- Other definitions:
 - A typing rule is **bivariant** if it is both covariant and contravariant (i.e. the type ordering remains the same)
 - e.g. If $A \leq B$, then $R(A) \equiv R(B)$
 - A typing rule is **variant** if it is either covariant, contravariant or bivariant
 - A typing rule is **invariant** if it is not variant

OOD Principles - SOLID

- Liskov Substitution Principle imposes the following constraints:
 - A subtype's method parameter types must be contravariant
 - i.e. parameters for subtype's methods must be the same type or more general
 - A subtype's method return types must be covariant
 - i.e. method returns for subtype must be the same type or more specific
 - Methods in the subtype cannot throw new exceptions unless they are subtypes of exceptions thrown by the supertype
- These rules help ensure that a type can always be substituted by a subtype

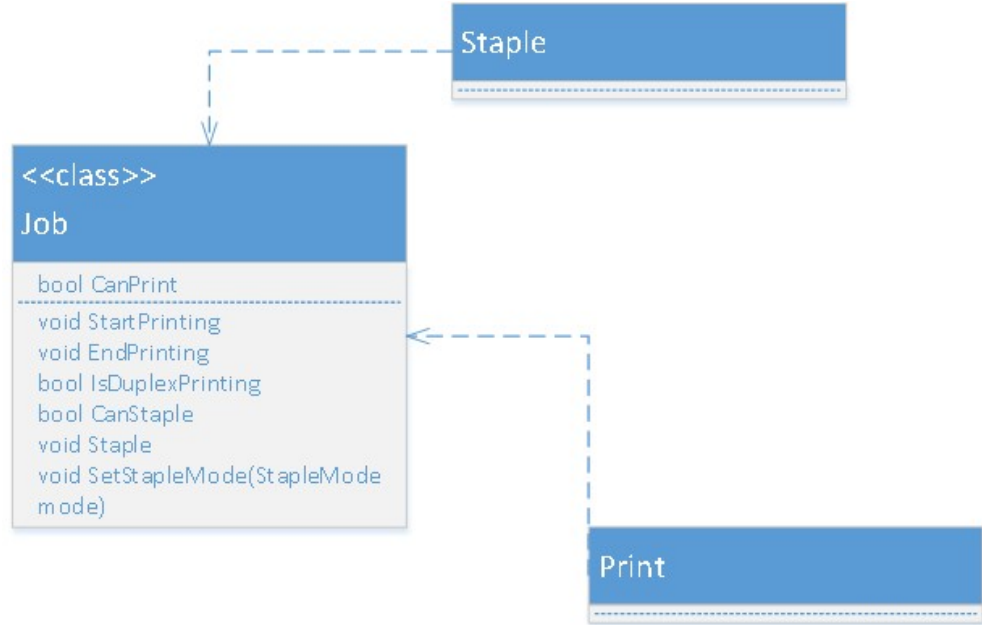
OOD Principles - SOLID

- **Interface segregation principle**

- Many client-specific interfaces are better than one general-purpose interface.
- No object should be forced to depend on methods it does not use
- Therefore, split large interfaces into smaller, role-specific interfaces (called **role interfaces**) so that client objects only know about methods that are of interest to them

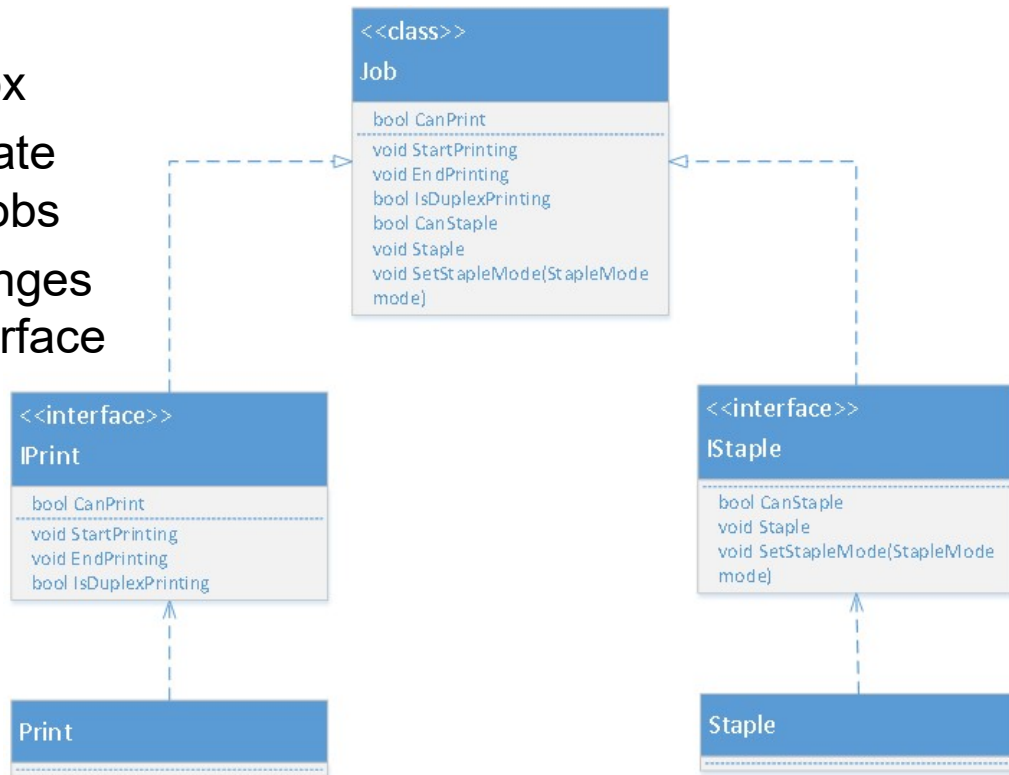
OOD Principles - SOLID

- Real example: printing jobs at Xerox
 - Multiple different types of printers use the same software, all relying on a Job class
 - Updates to any part of the Job class require every printer to update, which could take a long time if the Job class is bloated



OOD Principles - SOLID

- Real example: printing jobs at Xerox
 - To solve this, they created separate interfaces for different types of Jobs
 - Speeds up redeployment as changes to a type of job only affect its interface



OOD Principles - SOLID

- **Dependency inversion principle**
 - Depend upon abstractions, not concretions.
- Two basic principles:
 - High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g. interfaces).
 - Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

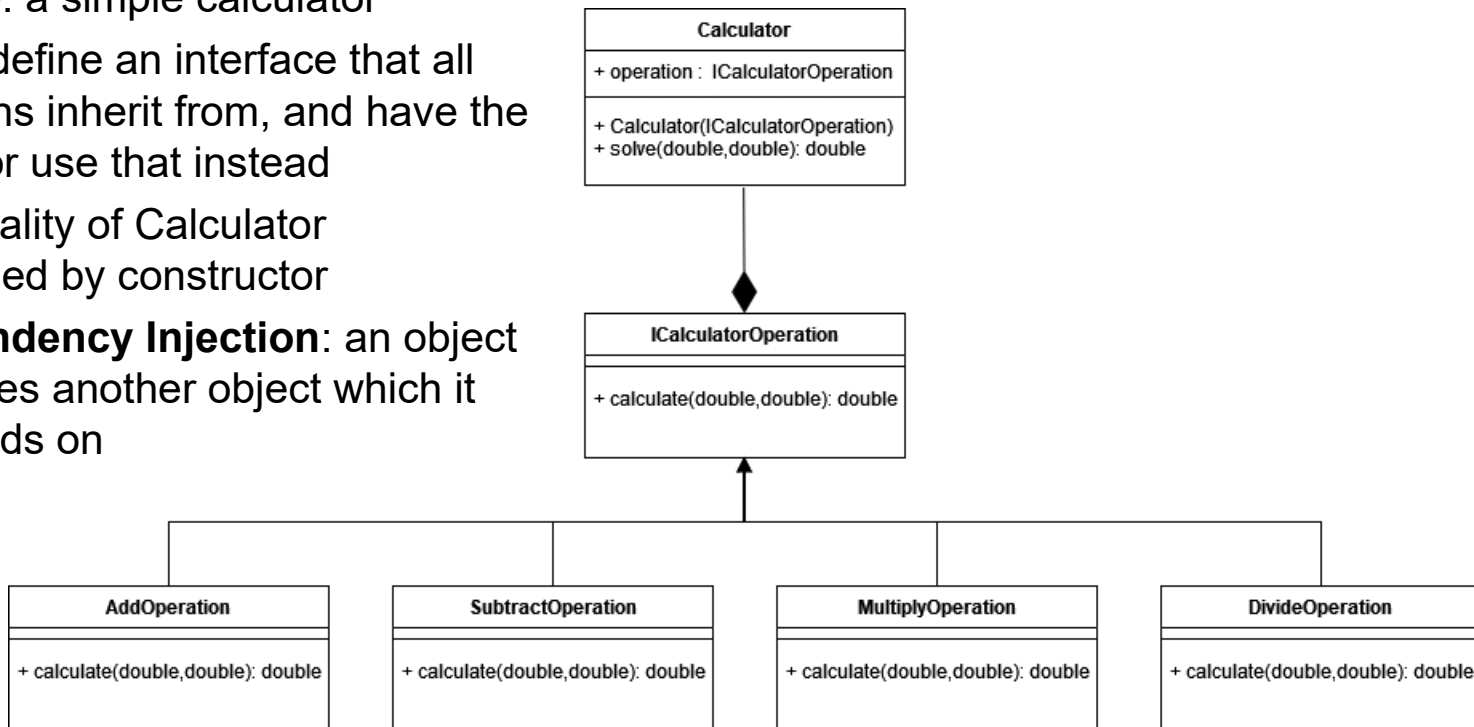
OOD Principles - SOLID

- Example: a simple calculator
- We could implement a single class Calculator with defined methods (e.g. add and subtract)
- But what if we want to extend Calculator to include multiply and divide in a later version?
 - Can't modify Calculator, as this violates Open-Closed Principle

Calculator
+ add(double,double): double + subtract(double,double): double

OOD Principles - SOLID

- Example: a simple calculator
- Instead define an interface that all operations inherit from, and have the calculator use that instead
- Functionality of Calculator determined by constructor
 - **Dependency Injection:** an object receives another object which it depends on

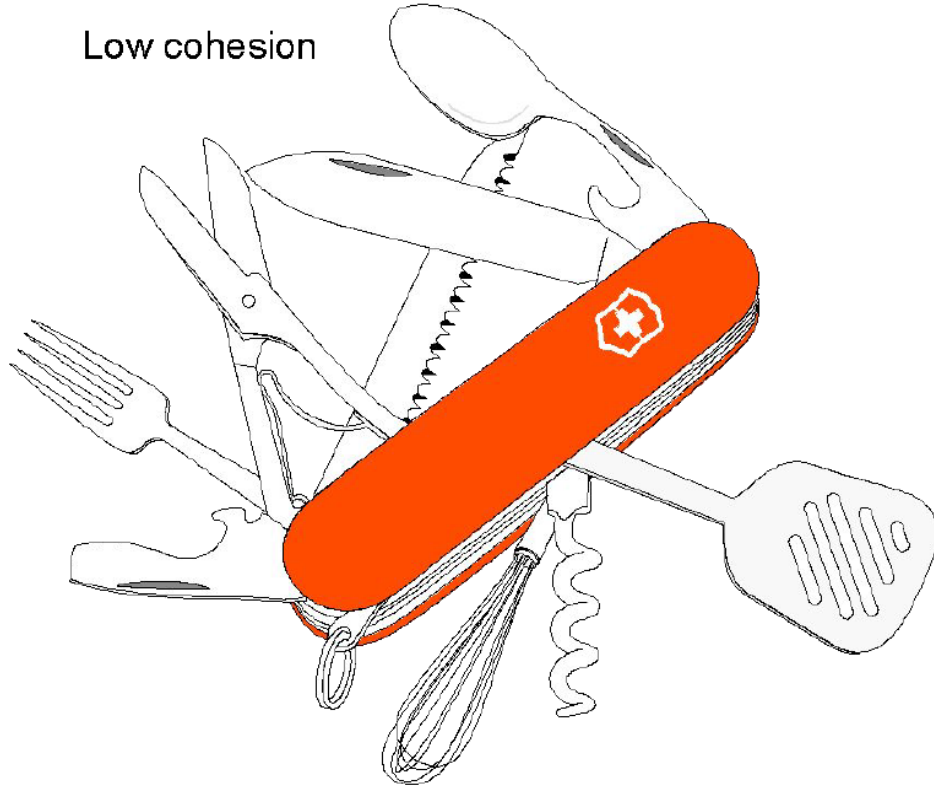


Other OOD Principles

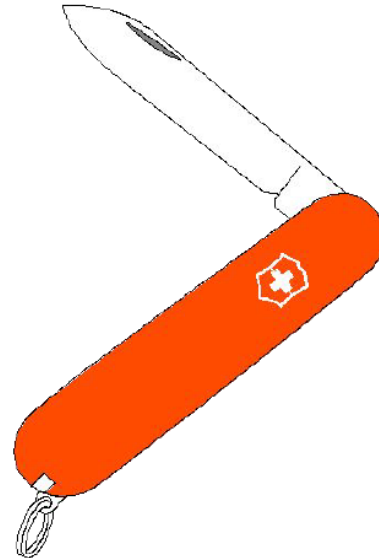
- **Expert-Doer Principle** – the component that knows should do the task
 - How to recognize a violation: if a method call passes too many parameters
- **High Cohesion Principle** – all the responsibilities of a component are related
 - i.e. do not take on too many computation responsibilities
 - How to recognize a violation: If a class has many loosely or not related attributes and methods
- **Low/Loose Coupling Principle** – a component is minimally connected to other components
 - i.e. do not take on too many communication responsibilities
 - How to recognize a violation: change in one component has large effect on multiple other components

Other OOD Principles

Low cohesion



High cohesion

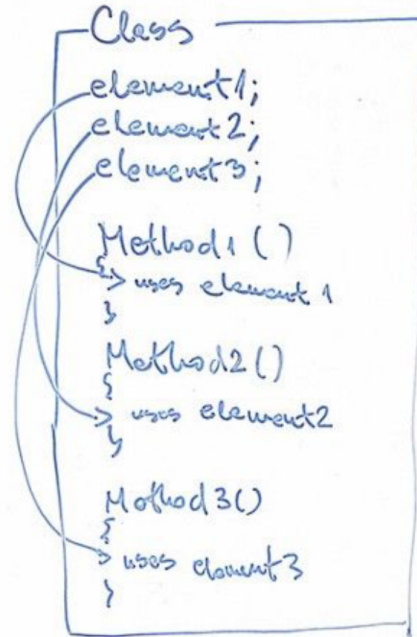


Other OOD Principles

HIGH COHESION

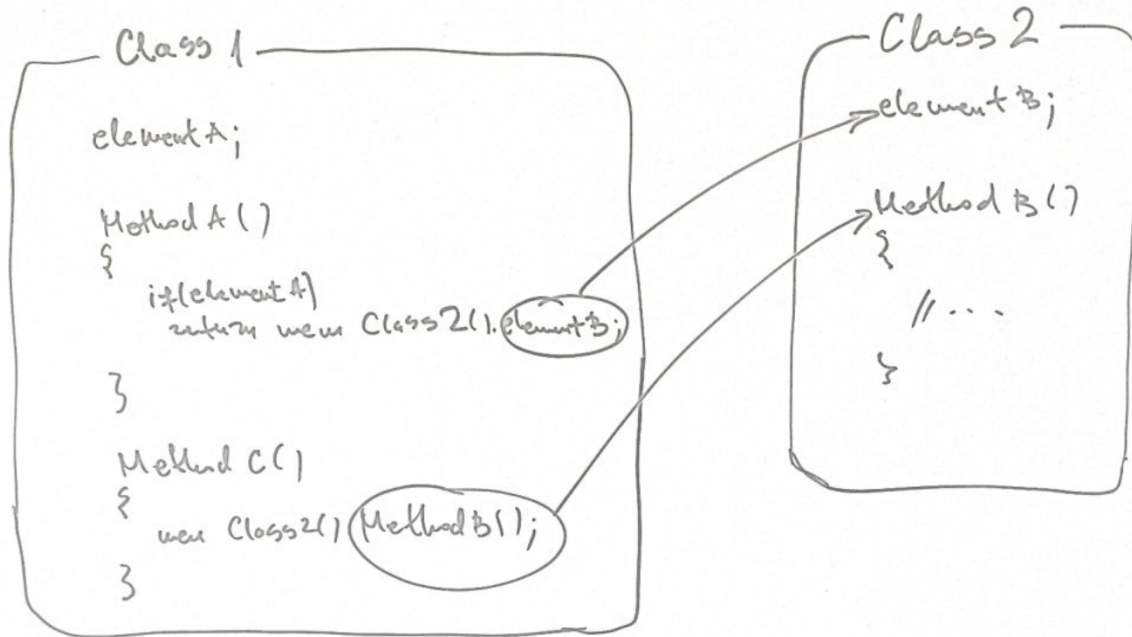


LOW COHESION

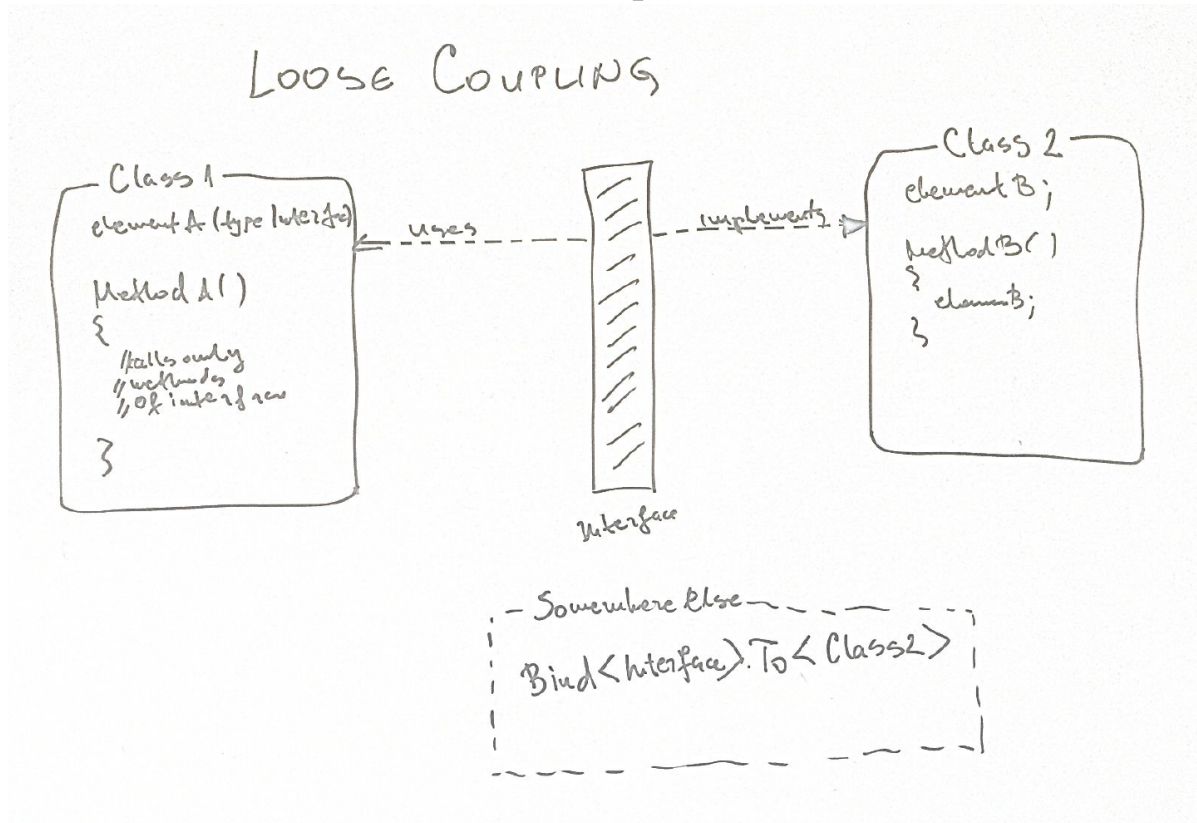


Other OOD Principles

TIGHT COUPLING

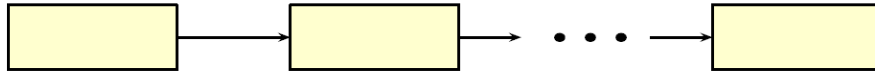


Other OOD Principles

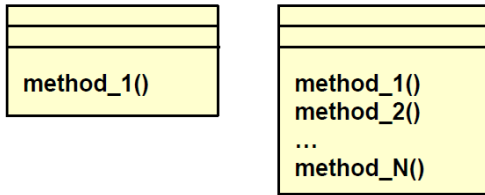


General Signs of Good OOD

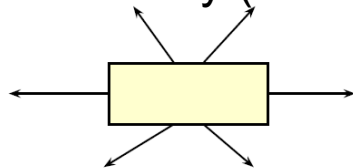
- Short communication chains between objects



- Balanced workload across objects



- Low degree of connectivity (associations) between objects



A Note on Design Principles

- SOLID and other good design principles, as well as OOD in general, are tools
- They are not magic, one-size-fits-all solutions
- Apply them where they make sense and ignore them where they don't
 - But make sure you are considering why you would or would not use a particular tool

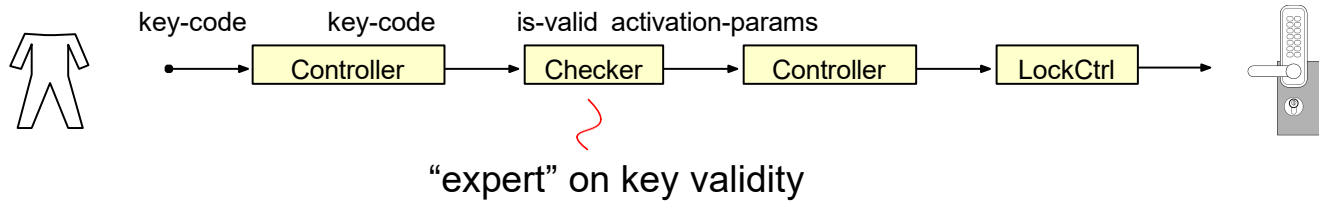
Responsibility-Driven Design

- Step 1: Identify system responsibilities
 - Requirements analysis
 - You'll invariably miss some at first, but you'll catch them in later iterations
- Step 2: For each responsibility, identify the possible assignments
- Step 3: Consider the trade-offs between each possible assignment by applying good design principles
 - Spoiler alert: Design Patterns
 - Select what you consider optimal, based on requirement priorities
- Step 4: Document the process by which you arrived at your chosen assignment to ensure traceability
 - This includes external documentation (e.g. a wiki) and internal documentation (e.g. code comments)

How Data Travels

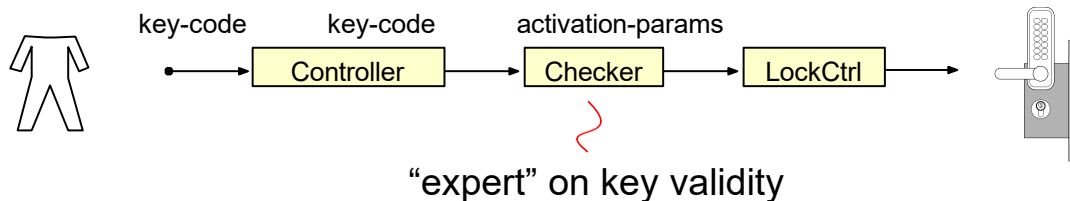
Option A:

“expert” (Key Checker) passes the information (key validity) to another object (Controller) which uses it to perform some work (activate the lock device)



Option B:

“expert” (Key Checker) directly uses the information (key validity) to perform some work (activate the lock device)

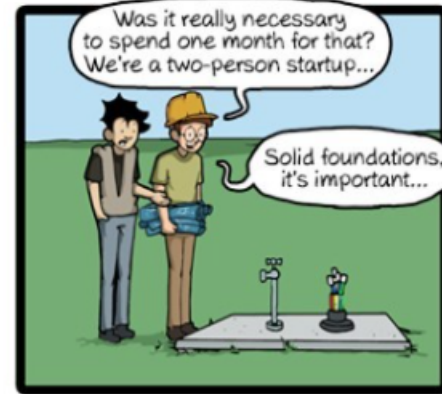
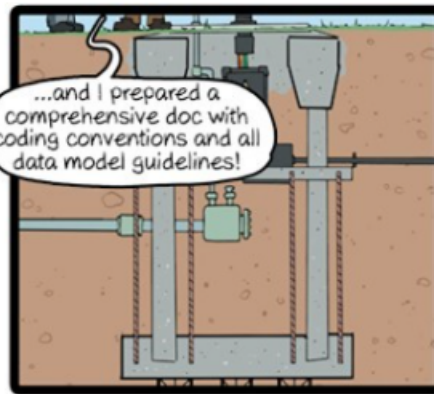
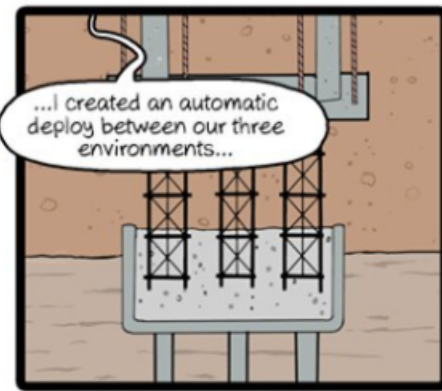
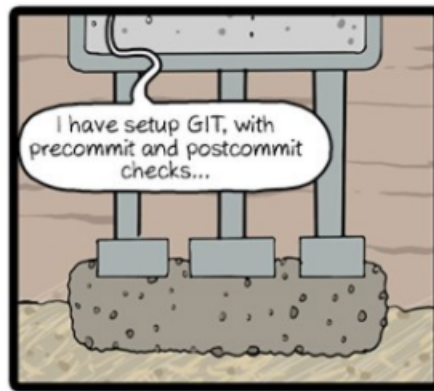


Advantage:
Shorter communication chain

Drawback:
Extra responsibility for Checker

Conclusion

- SOLID and other such principles provide good practices to ensure software systems can evolve smoothly over time
 - However, they are not gospel. Use what works and discard what does not
 - Ignoring a design practice should be a conscious choice, not because you don't understand it
- A good plan in the beginning can save you a lot of work later on refactoring code
 - However, getting a minimum viable product out as soon as possible is also very important
- Everything in software design is a trade-off!



CommitStrip.com