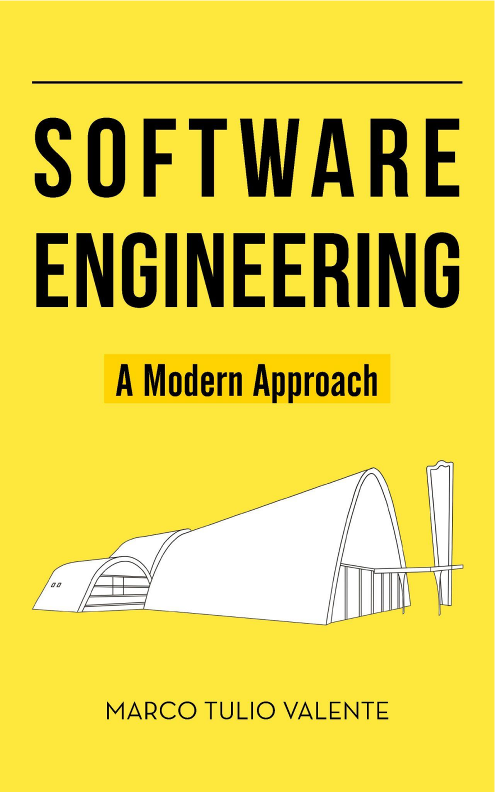
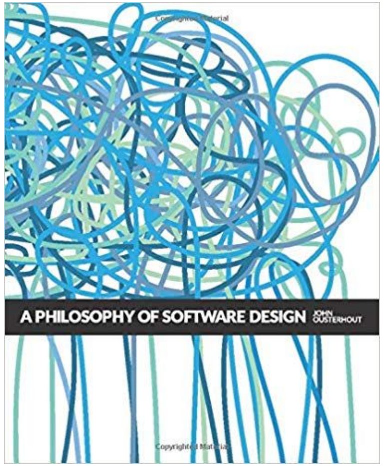


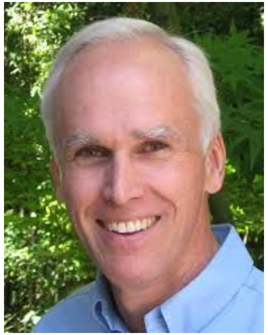
Chapter 5 - Design Principles

Prof. Marco Tulio Valente

https://softengbook.org

"The most fundamental problem in computer science is problem decomposition: how to take a complex problem and divide it up into pieces that can be solved independently"
-- John Ousterhout

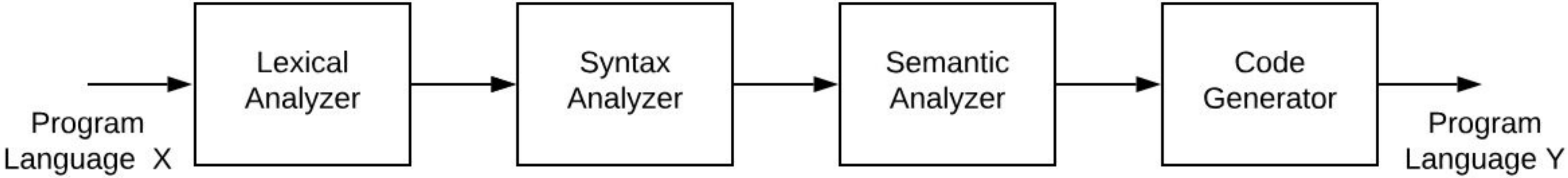




Definition

- Ousterhout's quote is an excellent definition for design
- Software design: breaking a "big problem" into smaller parts
- Implementing the smaller parts implements the "big problem"

Example: Compiler



Modules

- The smaller parts that result from the decomposition of the "big problem"
- Other names: packages, components, folders, layers, etc

What are we going to study?

- Design Properties
- Design Principles

Design Properties

1. Conceptual Integrity
2. Information Hiding
3. Cohesion
4. Coupling

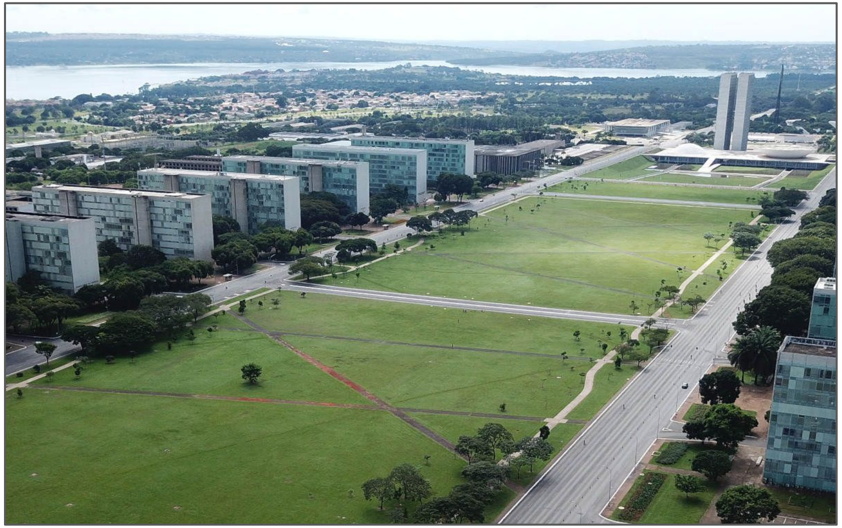
Design Principles

1. Single Responsibility
2. Interface Segregation
3. Prefer Interfaces to Classes
4. Prefer Interface to Composition
5. Open/Closed
6. Demeter
7. Liskov Substitution

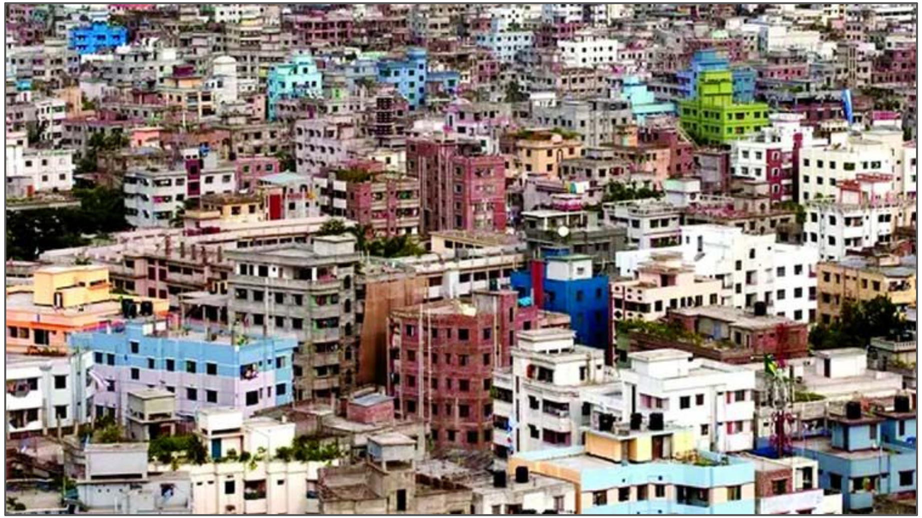
Design Properties

Conceptual Integrity

Conceptual Integrity: the coherence among features, design, and implementation decisions

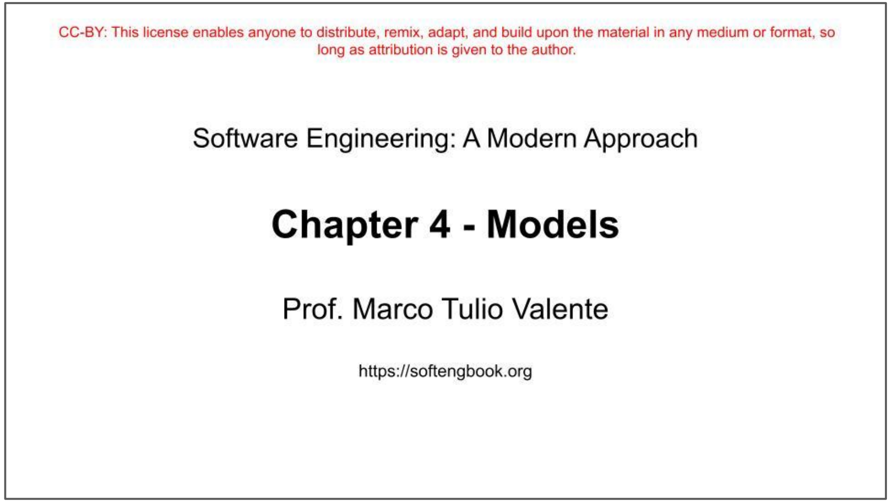


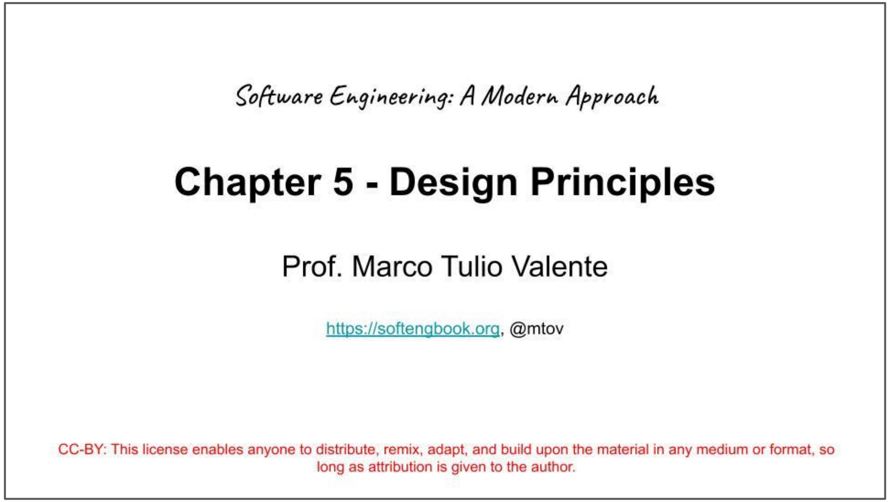
Example



Counter-Example

Why is there a lack of conceptual integrity in these slides?





Conceptual Integrity applies to:

- User interface
- Design decisions
- Implementation decisions
- Technological decisions
- etc

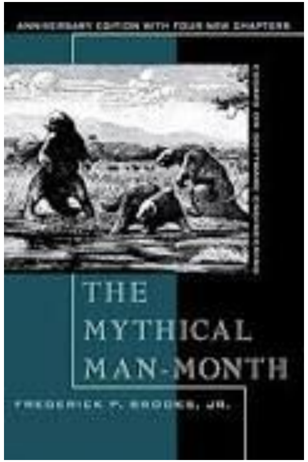
Examples (referring to user interface)

- The "Exit" button should be located in the same position on all pages
- If a system uses tables to present results, all tables should have the same layout
- All numerical results should be shown with 2 decimal places

Examples (at design/code level)

- All variables should follow the same naming pattern
  - Counter-example: `total_note` vs `averageNote`
- All modules should use the same framework version
- If a problem is solved using a data structure X, all similar problems must use X

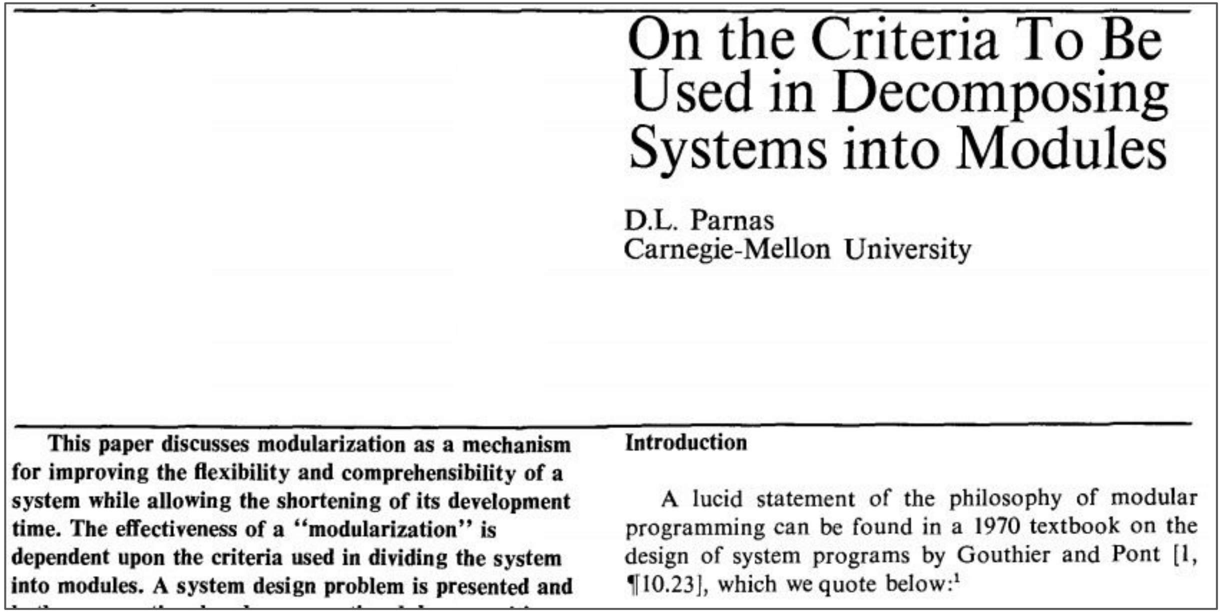
"Conceptual integrity is the most important consideration in system design" -- Fred Brooks



Reason: Conceptual integrity facilitates the use and understanding of a system

Information Hiding

Origin of this property (David Parnas, 1972)



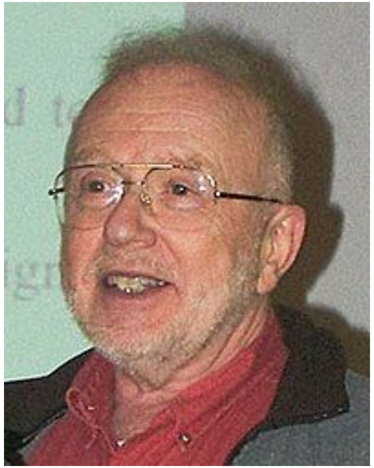
On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

This paper discusses modularization as a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time. The effectiveness of a "modularization" is dependent upon the criteria used in dividing the system into modules. A system design problem is presented and

Introduction

A lucid statement of the philosophy of modular programming can be found in a 1970 textbook on the design of system programs by Gouthier and Pont [1, ¶10.23], which we quote below:[1]



```
 import java.util.Hashtable;

public class ParkingLot {

  public Hashtable<String, String> vehicles;

  public ParkingLot() {
    vehicles = new Hashtable<String, String>();
  }

  public static void main(String[] args) {
    ParkingLot p = new ParkingLot();
    p.vehicles.put("TCP-7030", "Accord");
    p.vehicles.put("BNF-4501", "Corolla");
    p.vehicles.put("JKL-3481", "Golf");
  }
}
```

```java
import java.util.Hashtable;

public class ParkingLot {

  public Hashtable<String, String> vehicles;

  public ParkingLot() {
    vehicles = new Hashtable<String, String>();
  }

  public static void main(String[] args) {
    ParkingLot p = new ParkingLot();
    p.vehicles.put("TCP-7030", "Accord");
    p.vehicles.put("BNF-4501", "Corolla");
    p.vehicles.put("JKL-3481", "Golf");
  }
}
```

Problem: Developers have to manipulate an internal data structure to register a vehicle for parking

Problem

- Classes need some degree of "privacy"
- To allow them to evolve independently of other classes
- Previous code: client code directly accessed the hash table

Comparison with a manual parking control system

- Customers have to enter the parking lot booth
- And write down their car data in the logbook

Implementation with information hiding

1

```
import java.util.Hashtable;

public class ParkingLot {

  private Hashtable<String,String> vehicles;

  public ParkingLot() {
    vehicles = new Hashtable<String, String>();
  }

  public void park(String license, String vehicle) {
    vehicles.put(license, vehicle);
  }

  public static void main(String[] args) {
    ParkingLot p = new ParkingLot();
    p.park("TCP-7030", "Accord");
    p.park("BNF-4501", "Corolla");
    p.park("JKL-3481", "Golf");
  }
}
```

```java
 import java.util.Hashtable;

public class ParkingLot {

  private Hashtable<String,String> vehicles;

  public ParkingLot() {
    vehicles = new Hashtable<String, String>();
  }

  public void park(String license, String vehicle) {
    vehicles.put(license, vehicle);
  }

  public static void main(String[] args) {
    ParkingLot p = new ParkingLot();
    p.park("TCP-7030", "Accord");
    p.park("BNF-4501", "Corolla");
    p.park("JKL-3481", "Golf");
  }
}
```

2

```java
import java.util.Hashtable;

public class ParkingLot {

  private Hashtable<String,String> vehicles;

  public ParkingLot() {
    vehicles = new Hashtable<String, String>();
  }

  public void park(String license, String vehicle) {
    vehicles.put(license, vehicle);
  }

  public static void main(String[] args) {
    ParkingLot p = new ParkingLot();
    p.park("TCP-7030", "Accord");
    p.park("BNF-4501", "Corolla");
    p.park("JKL-3481", "Golf");
  }
}
```

3

ParkingLot is now free to change its internal data structures

Information Hiding

- Classes should hide their internal implementation details
  - By using the private modifier
  - Particularly those details that are subject to change
- Additionally, the class interface should remain stable
- Interface: the set of public methods and attributes of a class

Meanings of the word interface

1. Interface: set of public methods of a class
2. Interface: language construct (reserved keyword)
3. User interface (UI), graphical user interface (GUI), mobile interface, etc ⇒ outside the scope of this course

Interface in Java

```java
interface Animal {
  void makeSound();
}

class Dog implements Animal {
  public void makeSound() {
System.out.println("Woof!");
  }
}

class Cat implements Animal {
  public void makeSound() {
System.out.println("Meow!");
  }
}
```

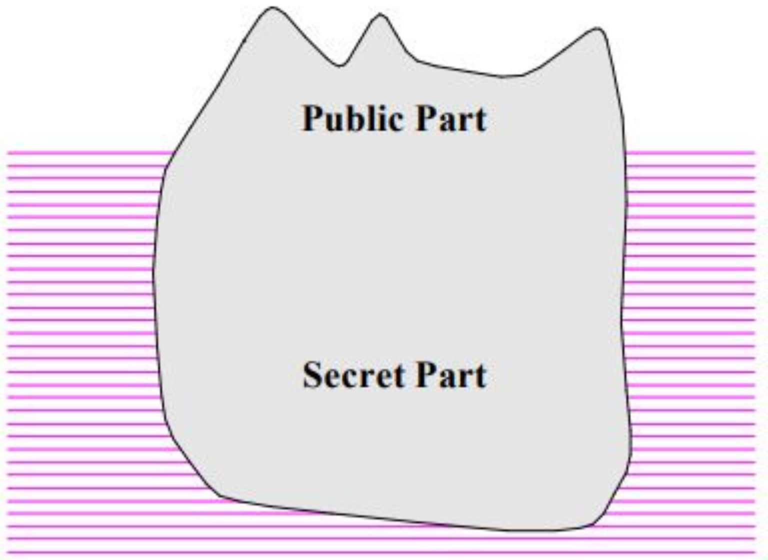
Dog Cat etc

```java
class MyClass {
  public void f(Animal animal) {
    ...
    animal.makeSound();
    ...
  }
}
```

Even if a class doesn't implement an interface (reserved keyword), it has an interface (its public methods)

Good modules are like icebergs

(small public and visible part; large submerged and private part)



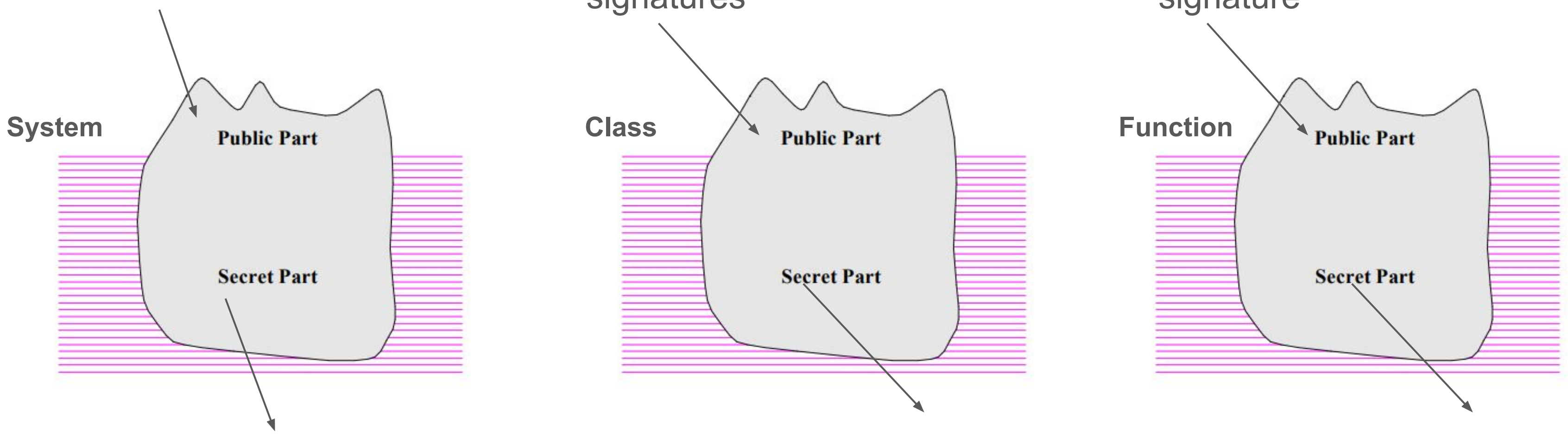
Source: Bertrand Meyer, Object-oriented software construction, 1997 (page 51)

Generalizing to systems, classes, and functions

API (REST, gRPC, GraphQL

Public method signatures

Function or method signature



Complete system

Implementation of public methods; private methods

Function or method implementation

Another name: encapsulation

- Some authors prefer the term **encapsulation**, but with the same meaning as information hiding.

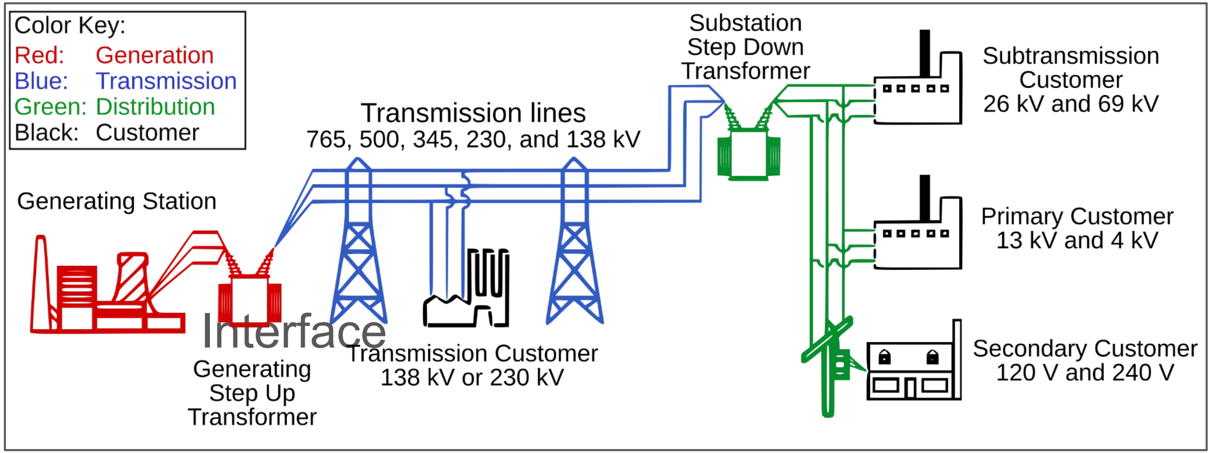
Encapsulation

See *information hiding*.

Glossário do livro Object-oriented Software Construction.
Bertrand Meyer (p. 1195)

Information Hiding in 1 slide

Implementation



Interface

110 volts

https://en.wikipedia.org/wiki/Electrical_grid

https://en.wikipedia.org/wiki/AC_power_plugs_and_sockets

Cohesion

Cohesion

- Classes should have a single goal and offer a single service
- This recommendation applies to functions, methods, packages, etc.

Counter-example 1

```
float sin_or_cos(double x, int op) {
  if (op == 1)
    "calculates and returns the sine of x"
  else
    "calculates and returns the cosine of x"
}
```

This should be broken down into two functions: sin and cos

Counter-example 2

```
class ParkingLot {
  ...
  private String managerName;
  private String managerPhone;
  private String managerSSN;
  private String managerAddress;
  ...
}
```

We should extract a Manager class, with the data about managers

Example

```
class Stack<T> {
  boolean empty() { ... }
  T pop() { ... }
  push (T) { ... }
  int size() { ... }
}
```

✅

All these methods manipulate Stack elements

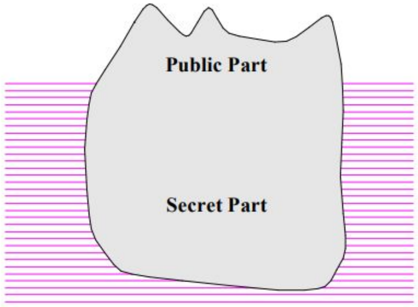
Coupling

Coupling

- No class is an island… Classes depend on each other
- They call methods of other classes, extend other classes,...
- The main issue is the quality of this coupling
- Types of coupling:
  - Acceptable coupling ("good")
  - Poor coupling ("bad")

Acceptable Coupling

- Class A uses a class B and:
  - B provides a very useful service for A
  - B has a stable interface
  - A only calls methods from B's interface



```java
import java.util.Hashtable;

public class ParkingLot {

  private Hashtable<String,String> vehicles;

  public ParkingLot() {
    vehicles = new Hashtable<String, String>();
  }

  public void park(String license, String vehicle) {
    vehicles.put(license, vehicle);
  }

  public static void main(String[] args) {
    ParkingLot p = new ParkingLot();
    p.park("TCP-7030", "Accord");
    p.park("BNF-4501", "Corolla");
    p.park("JKL-3481", "Golf");
  }
}
```

✅

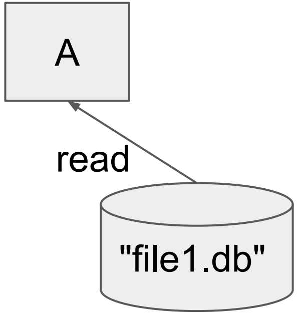
ParkingLot is coupled to Hashtable, but this coupling is acceptable

Poor Coupling

- Class A uses a class B:
  - But B's interface is unstable
  - Or the usage does not occur via B's interface

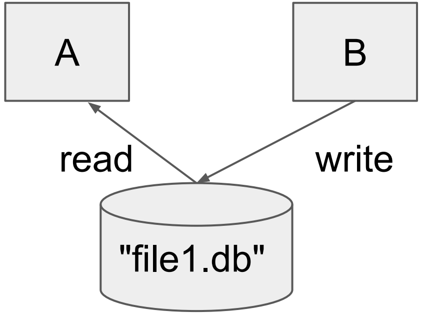
How can class A be coupled to a class B without it being via B's interface?

```
class A {
  private void f() {
    int total; ...
    File file = File.open("file1.db");
    total = file.readInt();
    ...
  }
}
```



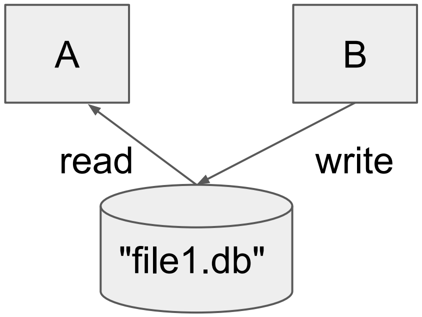
```java
class A {
  private void f() {
    int total; ...
    File file = File.open("file1.db");
    total = file.readInt();
    ...
  }
}
```

```java
class B {
  private void g() {
    int total;
    // computes total value
    File file = File.open("file1.db");
    file.writeInt(total);
    ...
    file.close();
  }
}
```



Poor Coupling

- Changes in B can easily impact A
- Example: B can change the format of the file or remove the data used by A



This is also called evolutionary coupling (or logical coupling)

How to solve this problem?

How to turn poor coupling into acceptable coupling?

Refactoring poor into acceptable coupling

```
class B {

  int total;

  public int getTotal() {
    return total;
  }

  private void g() {
    // computes total value
    File file = File.open("file1");
    file.writeInt(total);
    ...
  }
}
```

```
class A {

  private void f(B b) {
    int total;
    total = b.getTotal();
    ...
  }
}
```

✅

Common recommendation in software design:

Maximize cohesion, minimize coupling

But be careful: minimize primarily poor coupling

Summary

- Static (or structural) coupling:
  - In A's code, there is an explicit reference to B
  - Can be either acceptable or poor coupling
- Evolutionary (or logical) coupling:
  - In A's code, there is no reference to B
  - But, changes in B can impact A
  - Poor coupling (always)

Exercises

1. Suppose you are responsible for implementing a system that will have 100 KLOC.

Just as an exercise, propose a design for your implementation with the worst possible cohesion while maintaining the best possible coupling.

2. Consider the following code that performs operations on bank accounts. (a) What design principle is violated by this code? (b) How would you improve the design of this code?

```
var balance = [150, 10, 90]; // global

function deposit(account, value) {
  balance[account] += value;
}

function getBalance(account) {
  return balance[account];
}
```

3. Assume two classes A and B that:

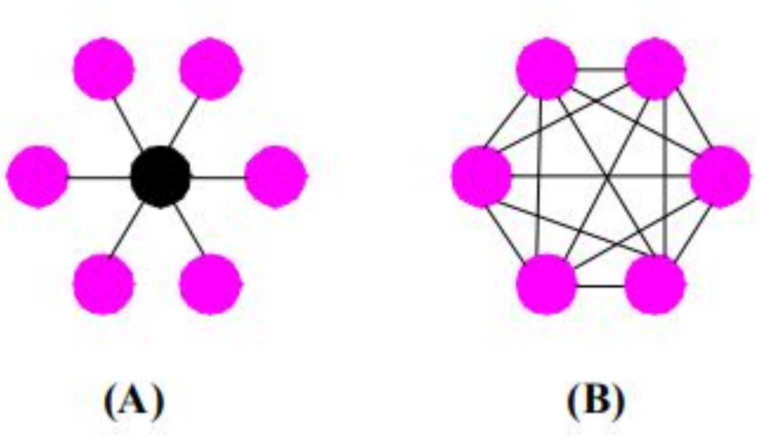
- are implemented in different directories
- class A has a reference in its code to class B

During maintenance, when a developer modifies both A and B, they always decide to move B to the same directory as A.

(a) When measured at the directory level, which design property is improved by this behavior?

(b) Which design property is compromised by this behavior?

4. Compare these two designs, where the nodes represent classes and the edges represent dependencies. Which design is generally better?



Source: Bertrand Meyer, Object-oriented software construction, 1997 (page 47)

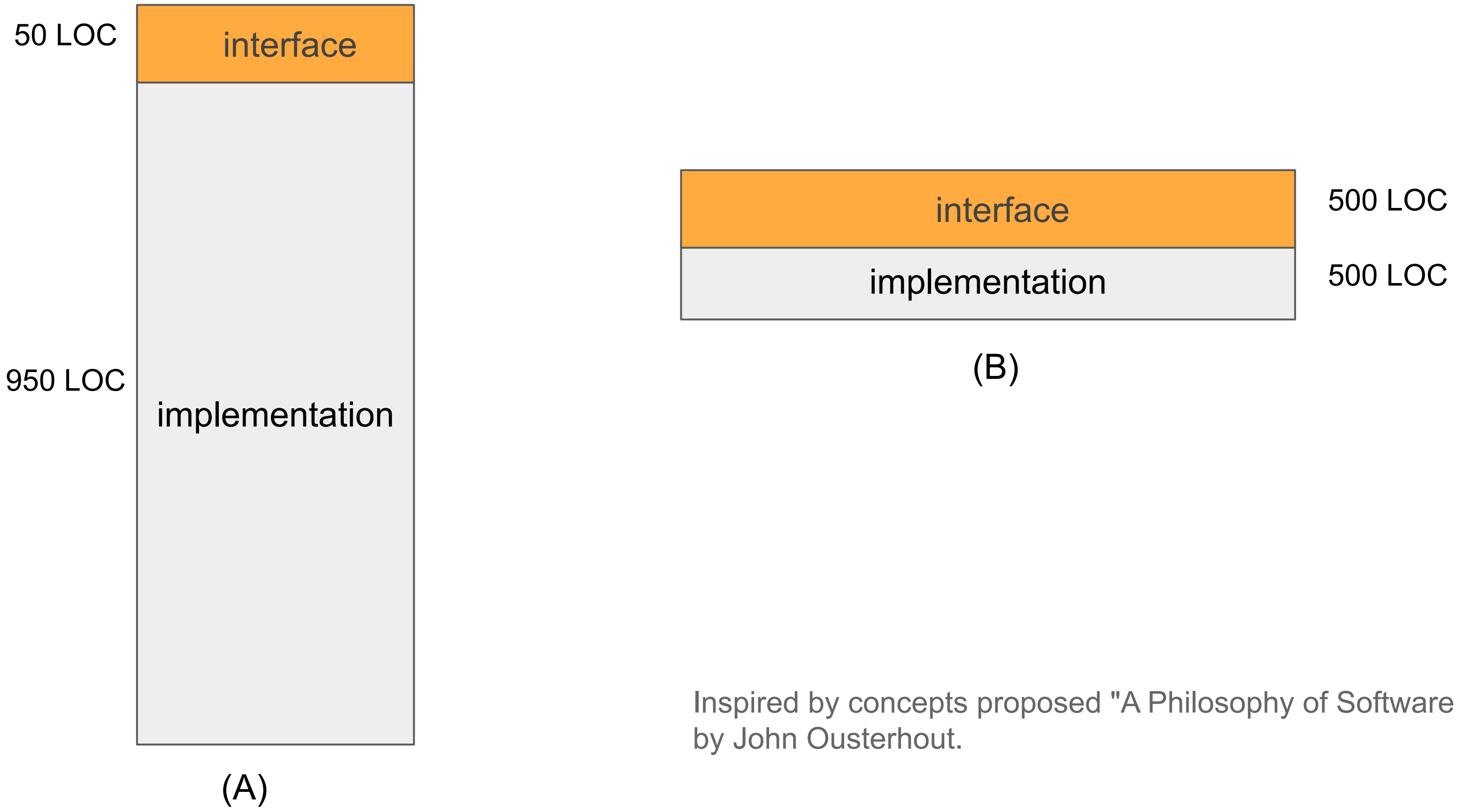
5. Compare these two designs, where the nodes represent classes and the edges represent dependencies. Which design is generally better?

(a)

(b)

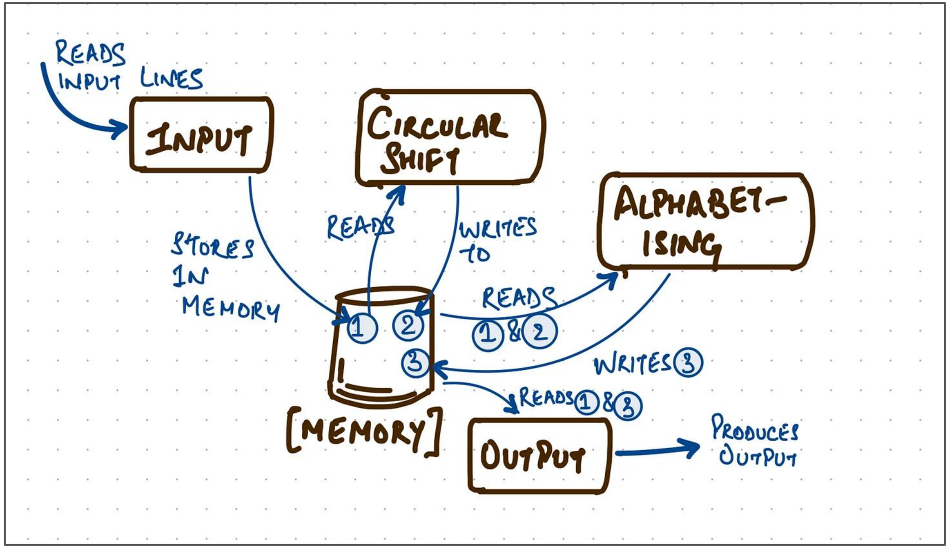
Source: The Pragmatic Programmer, 20th Anniversary Edition, Chapter 5

6. Which of the following modules is better? Justify.

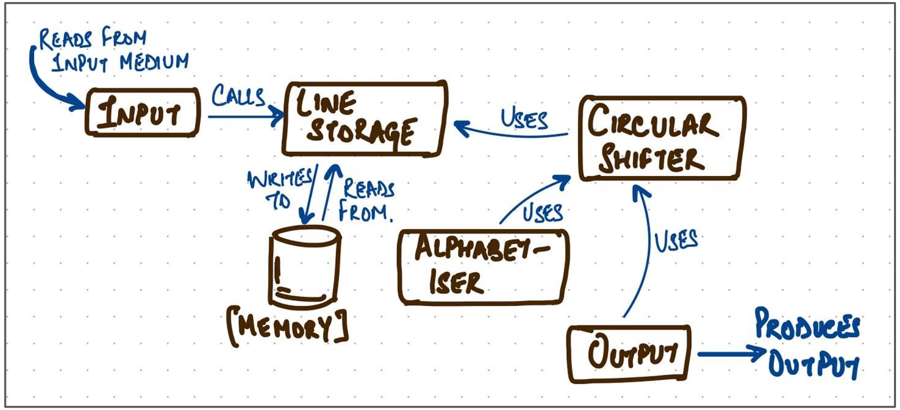


Inspired by concepts proposed "A Philosophy of Software Design".
by John Ousterhout.

7. Next, we show two modularizations of a program that reads lines from the input, creates all the “circular shifts” of those lines, and prints the shifts in alphabetical order (details in the next slide). (a) Which modularization is better? (b) Which design property does it address?



Modularization I



Modularization II

Source: https://www.riverandsoftware.com/p/criteria-to-be-used-in-modularisation-paper

Comments on the previous exercise

- This system, called KWIC (Keywords in Context), was used as an example in Parnas' software modularization paper (1972)
- Example of input and output (showing sorted “circular shifts”)

Input:

Pattern-Oriented Software Architecture
Software Architecture
Introducing Design Patterns

Output (assuming Pattern-Oriented treated as one word):

Architecture Software
Architecture Pattern-Oriented Software
Design Patterns Introducing
Introducing Design Patterns
Patterns Introducing Design
Pattern-Oriented Software Architecture
Software Architecture
Software Architecture Pattern-Oriented

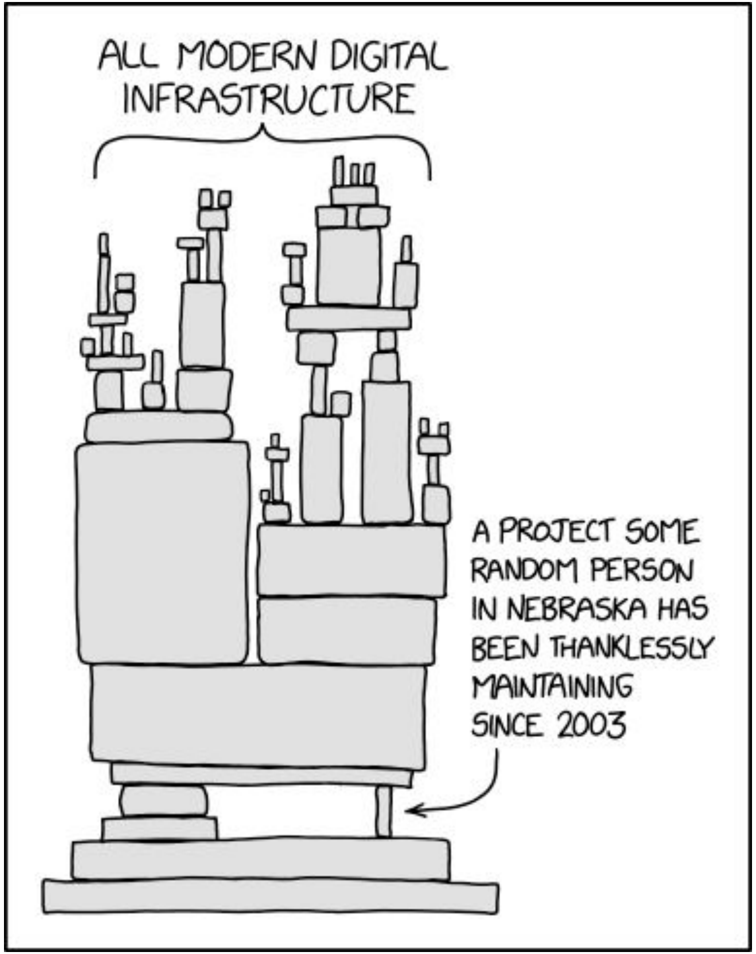
8. Suppose two methods `f` and `g`. A **temporal coupling** exists between them when to call `g` we have to call `f` first.

(a) Give an acceptable and common example of temporal coupling (that is, give concrete names of methods `f` and `g`).

(b) Analyze the temporal coupling in the following code. Is it acceptable or problematic? If problematic, propose a refactoring.

```
var circle = new Circle();
circle.setRadius(5);
circle.getArea();
```

9. The following cartoon relates to a violation of which design property?



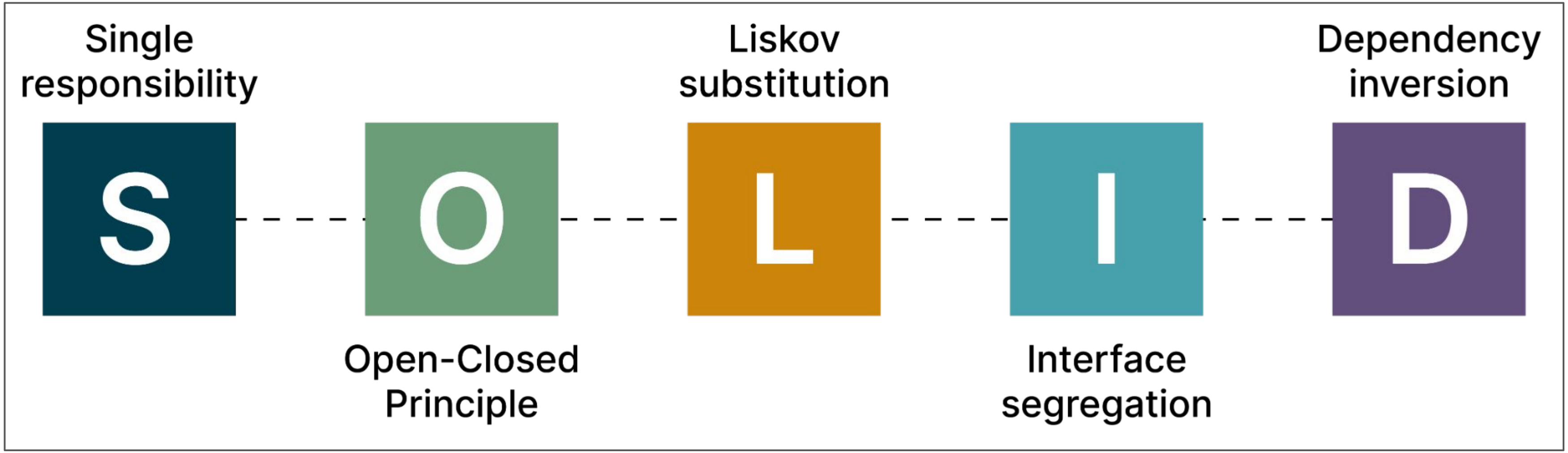
https://xkcd.com/2347

Design Principles

Design Principle	Design Property
Single Responsibility	Cohesion
Interface Segregation	Cohesion
Dependency Inversion	Coupling
Favor Composition over Inheritance	Coupling
Demeter	Information Hiding
Open/Closed	Extensibility
Liskov Substitution	Extensibility

Guidelines

Benefits (what we can gain by following these principles)



Source: ThoughtWorks blog (link)

(1) Single Responsibility Principle (SRP)

Single Responsibility Principles

- Every class should have a single responsibility
- A class should have only one reason to change

Responsibility #1: compute dropout rate

```
class Course {
  void calculateDropoutRate() {
    rate = "compute dropout rate";
    System.out.println(rate);
  }

}
```

Responsibility #2: print result

Single responsibility: user interface
⇒ frontend dev

```
class Console {
  void printDropoutRate(Course course) {
    double rate = course.calculateDropoutRate();
    System.out.println(rate);
  }
}

class Course {
  double calculateDropoutRate() {
    double rate = "compute the dropout rate"
    return rate;
  }
}
```

✅

Single responsibility:
business logic
⇒ backend dev
⇒ easier to test

(2) Interface Segregation Principle (ISP)

Interface Segregation Principle

- Basically, this principle applies SRP to interfaces
- Interfaces should be:
  - Small
  - Cohesive
  - Specific for each type of client

```java
interface Account {
  double getBalance();
  double getInterest(); // only applicable to SavingsAccounts
  int getSalary(); // only applicable to SalaryAccounts
}
```

❌

✅

```java
interface Account {
  double getBalance();
}

interface SavingsAccount extends Account {
  double getInterest();
}

interface SalaryAccount extends Account {
  int getSalary();
}
```

Common to all accounts

Specific to SavingsAccount

Specific to SalaryAccount

(3) Dependency Inversion Principle (DIP)

Dependency Inversion

- We usually refer to this principle as "Prefer Interfaces to Classes"
- Because it better express its core concept

Example without using DIP

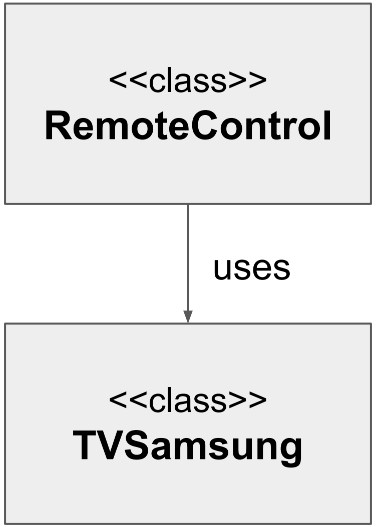
```
class RemoteControl {
   TVSamsung tv;
   ...
}


class TVSamsung {
   ...
}
```

What is the issue with this design regarding coupling and extensibility?

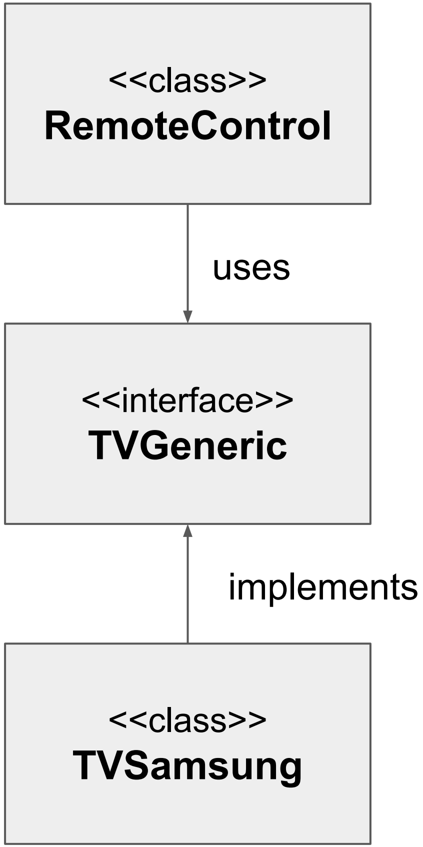
Example using DIP

```
class RemoteControl {
  TVGeneric tv;
   ...
}

interface TVGeneric {
   ...
}

class TVSamsung implements TVGeneric {
   ...
}
```

Without DIP

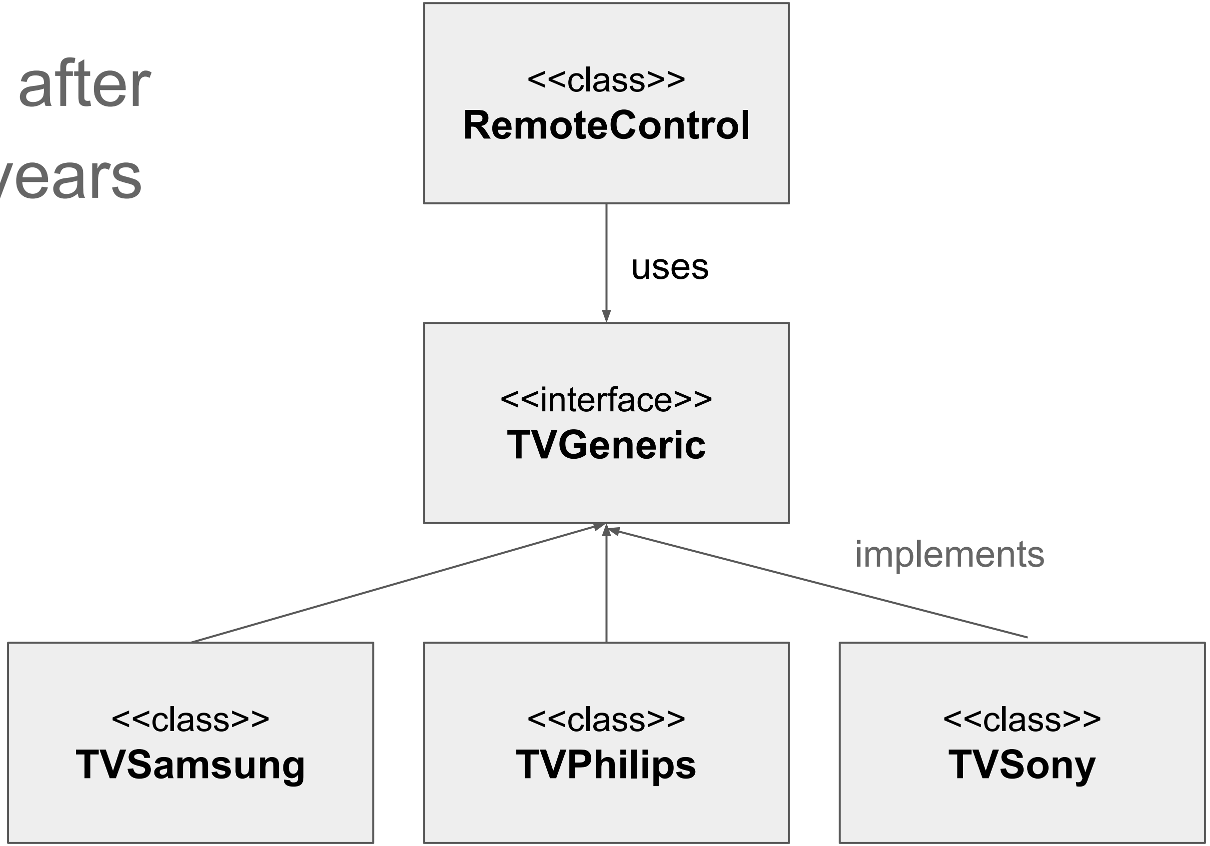


With DIP



Key Benefits: RemoteControl remains generic and reusable; We can work with different TV implementations without modifying RemoteControl.

Design after some years



(4) Prefer Composition to Inheritance

Historical Context

- During the 1980s, when OOP became popular, developers began to overuse inheritance
- They saw inheritance as a "silver bullet" for enabling large-scale reuse

Inheritance: “is-a” relationship

- In UML, corresponds to an association
- `GasolineEngine` <u>is a</u> `Engine`

```
class GasolineEngine extends Engine {
   ... // inherits attributes and methods from Engine
}
```

Composition: “has” relationship

- In UML, corresponds to an association
- Dashboard <u>has</u> a RPMGauge

```
class Dashboard {
  RPMGauge rpm; // has an attribute
  ...
}
```

Prefer Composition to Inheritance ⇒ don't force the use of inheritance

(5) Demeter Principle

Demeter

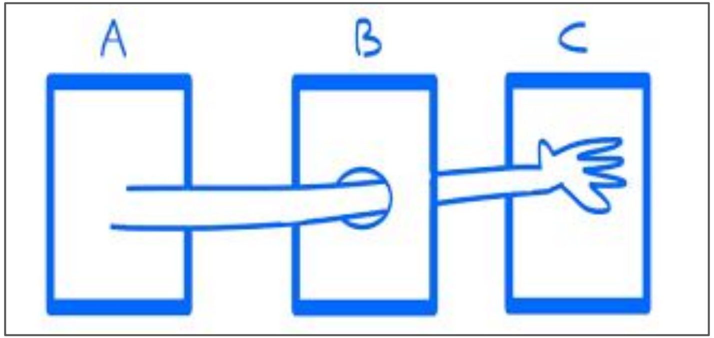
- Demeter: research group from a US university
- Avoid long chains of method calls
- Example:

```
obj.getA().getB().getC().getD().doSomething();
```

pass-through objects

Reason

- Long call chains break encapsulation
- It forces us to traverse through A, B, C... to get what we need



https://medium.com/@evan.hopkins.us/the-law-of-demeter-and-its-application-to-react-ab1e054f13c5

```
class DemeterPrinciple {

  T1 attr;

  void f1() {
    ...
  }

  void m1(T2 p) {   // method following Demeter
    f1();           // case 1: own class
    p.f2();         // case 2: parameter
    new T3().f3();  // case 3: created by the method
    attr.f4();      // case 4: class attribute
  }

  void m2(T4 p) {  // method violating Demeter
    p.getX().getY().getZ().doSomething();
  }

}
```

Warning

- Demeter and other principles are recommendations
- We should not be dogmatic and assume that method call chains are always prohibited
- Specific cases may exist and have valid justification

Acceptable example of method chaining

```
const numbers = [1, 2, 3, 4, 5, 6];
const result = numbers
  .map(num => num * 2)
  .filter(num => num % 3 === 0)
  .reduce((acc, num) => acc + num, 0);
console.log(result); // Output: 18
```

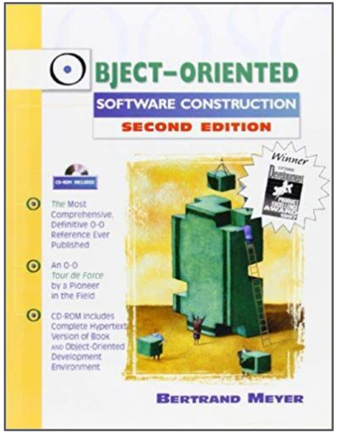
✅

Methods of the language or its API

(6) Open/Close Principle (OCP)

Open/Closed Principle

- Proposed by Bertrand Meyer
- A class should be closed for modification, but open for extension



1988

Explaining further

- Suppose you are going to implement a class
- Clients will want to use your class -- that's expected!
- But, they will also want to customize and extend it
- You should design for and enable such extensions
- Objective: prevent clients from having to edit your class to customize it

How to make a class open to extensions, while keeping its code closed to modifications?

- Parameters
- Inheritance
- Higher-order functions (takes other functions as arguments or result)
- Design patterns (chapter 7)
- etc

Example

Sorts the list

```
List<String> names;
names = Arrays.asList("john", "megan", "alexander", "zoe");

Collections.sort(names);

System.out.println(names);
// result: ["alexander","john","megan","zoe"]
```

But now one user (developer) wants to sort the the list elements by their length (# of chars)

Can "sort" accommodate this extension while keeping its code closed to modification?

list to be sorted

Function used by sort to compare s1 and s2:

- result < 0 ⇒ order is s1, s2
- result = 0 ⇒ order doesn't matter
- result > 0 ⇒ order is s2, s1

```
Collections.sort(names, (s1, s2) -> s1.length() - s2.length());

System.out.println(names);
// result: ["zoe", "john", "megan", "alexander"]
```

✅

There is no free lunch: to call "sort", we now need to implement and pass a small function as a parameter that defines the sorting criteria.

In summary: when implementing a class, think about extension points!

(7) Liskov Substitution Principle (LSP)

Liskov Substitution Principle

- The name is a reference to Prof. Barbara Liskov
- LSP defines best practices for implementing inheritance
- It provides guidelines for redefining methods in subclasses

First, let's understand what we mean by substitution

```
void f(A a) {
  ...
  a.g();
  ...
}
```

```
void f(A a) {
  ...
  a.g();
  ...
}
```

```
f(new B1());  // f can receive objects from subclass B1
...
f(new B2());  // and from any subclass of A, such as B2
...
f(new B3());  // and B3
```

```
void f(A a) {
  ...
  a.g();
  ...
}
```

Type A can be replaced by B1, B2, B3,...

As long as they are subclasses of A

Liskov Substitution Principle

- Substitutions from A to B can occur as long as B provides at least the same services as A
- For the code that uses A, the substitution should be imperceptible

Example that follows LSP

```
class RemoteControl {

    // range of 10 meters

}

class PremiumRemoteControl extends RemoteControl {

    // range of 20 meters

}
```

✅

Example that does not follow LSP

```
class RemoteControl {

    // range of 10 meters

}

class BasicRemoteControl extends RemoteControl {

    // range of 5 meters

}
```

Exercises

1. Which design principle is violated by a call like the one shown below? What design change would you make in the `Library` class (the type of `lib`) to remove this violation?

```
lib.getCollection()
    .getKnowledgeArea("SE")
    .getBooks()
    .find("SoftEngBook")
    .getNumCopies();
```

2. Suppose the following class:

```
class Table {
  ...
  void print() {
    // prints the table header
    // prints each line of the table
    // prints the table footer
  }
  ...
}
```

This class violates the Open/Closed Principle because it lacks flexibility in configuring the header and footer messages. How would you refactor this class to follow this principle?

3. Consider a `Calculator` class with a method that checks if a number between 0 and 10,000 is prime. A subclass called `FastCalculator` implements a more efficient algorithm, but it only works with numbers between 1,000 and 9,000.

`class Calculator {`
`    boolean isPrime(n) {`
`        // 0 <= n < 10000`
`    }`
`}`
	`class FastCalculator extends Calculator {`
	`    boolean isPrime(n) {`
	`        // 1000 <= n < 9000`
	`    }`
	`}`

Which SOLID design principle is violated in this implementation? Explain your reasoning.

4. Consider you finished an outreach course offered by your university and want to receive your certificate. To do that, you had to:

- Send a mail to the coordinator, who asked you to send a mail to the department secretary.
- Then, you sent a mail to the secretary, who asked you to send a mail to the Center of Extension (CENEX).
- Then, you sent a mail to CENEX, who asked you to send a mail to the Pro-Rectorate of Extension (PROEX).
- Then, you sent a mail to PROEX, who returned your certificate.

(a) Which design principle is violated in this process? (b) Besides the multiple email exchanges, what other problem exists in this solution?

5. In Software Engineering, we sometimes implement unnecessarily complex solutions. This problem is called overengineering or premature optimization.

Provide an example where applying one of the design principles we studied is a premature optimization. You may reference examples from previous slides.

End