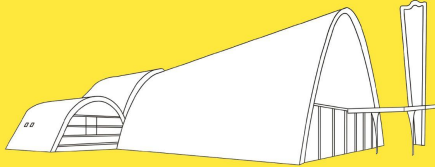


# SOFTWARE ENGINEERING

A Modern Approach



MARCO TULLIO VALENTE

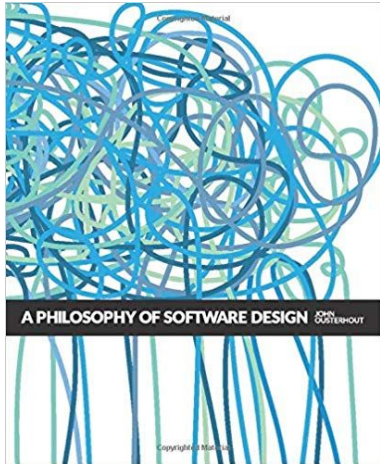
## Chapter 5 - Design Principles

Prof. Marco Tulio Valente

<https://softengbook.org>

CC-BY: This license enables anyone to distribute, remix, adapt, and build upon the material in any medium or format, so long as attribution is given to the author.

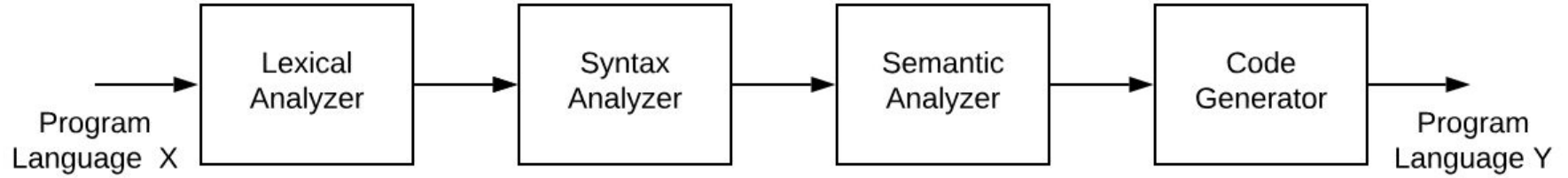
"The most fundamental problem in computer science is problem decomposition: how to take a complex problem and divide it up into pieces that can be solved independently"  
-- John Ousterhout



# Definition

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

# Example: Compiler



# Modules

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

# What are we going to study?

- Design Properties
- Design Principles

# Design Properties

- Conceptual Integrity
- Information Hiding
- Cohesion
- Coupling

# Design Principles

- Single Responsibility
- Interface Segregation
- Prefer Interfaces to Classes
- Open/Closed
- Demeter
- Liskov Substitution



# Design Properties

# Conceptual Integrity

# Conceptual Integrity: coherence of features, design, and implementation decisions



Example



Counter-Example

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

CC-BY: This license enables anyone to distribute, remix, adapt, and build upon the material in any medium or format, so long as attribution is given to the author.

Software Engineering: A Modern Approach

## Chapter 4 - Models

Prof. Marco Tulio Valente

<https://softengbook.org>

*Software Engineering: A Modern Approach*

## Chapter 5 - Design Principles

Prof. Marco Tulio Valente

<https://softengbook.org>, @mtov

CC-BY: This license enables anyone to distribute, remix, adapt, and build upon the material in any medium or format, so long as attribution is given to the author.

# Conceptual Integrity applies to:

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

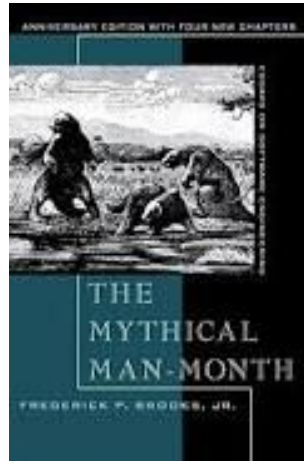
## Examples (referring to user interface)

- "Exit" button should be the same on all pages
- If a system uses tables to present results, all tables should have the same layout
- All 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 (same version)
- If a problem is solved using a data structure X, all similar problems should 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)

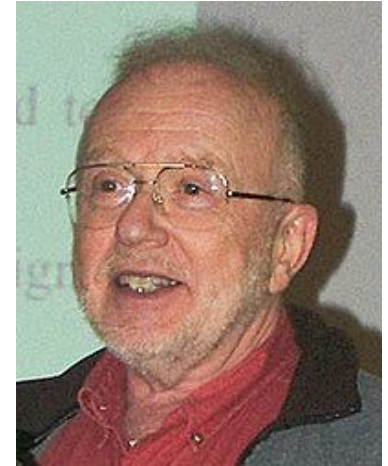
## On the Criteria To Be Used in Decomposing Systems into Modules

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 Gauthier and Pont [1, ¶10.23], which we quote below:<sup>1</sup>



```
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");
    }
}
```

```
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");
    }
}
```

license plate

car model

Constructor (creates the Hashtable)

```
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");
    }
}
```

```
import java.util.Hashtable;

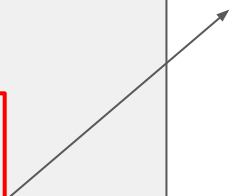
public class ParkingLot {

    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 parking



# Problem

- Classes need a little bit of "privacy"
- For evolving independently of the other classes
- Previous code: client code directly accesses the hashtable
- Comparison with a manual parking control system:
  - Customers have to enter the parking lot booth
  - 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");
    }
}
```

2

```
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");
    }
}
```

```
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 now free to change its internal data structures

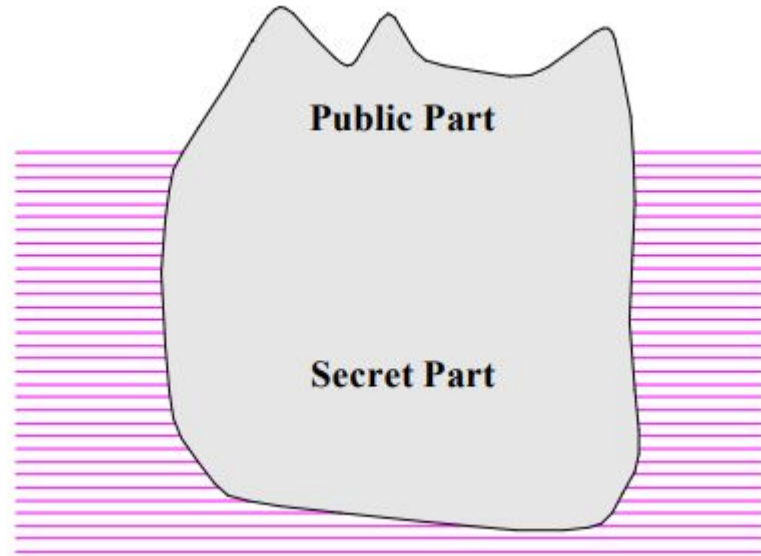
3

# Information Hiding

- Classes should hide their internal implementation details
- Using the `private` modifier
- Especially those details subject to change
- Additionally, the class `interface` should be stable
- Interface: set of public methods and attributes of a class

# 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)

# Cohesion

# Cohesion

- Classes should have a single goal and offer a single service
- This also 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"  
}
```



# 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"  
}
```



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;  
    ...  
}
```



## 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

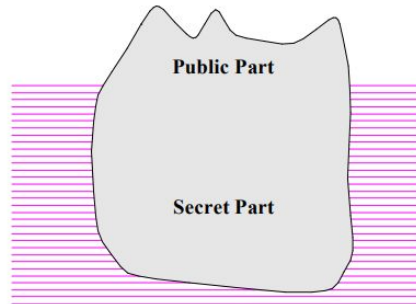
# Coupling

# 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

# 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





```
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");
```

```
    }
```

```
}
```



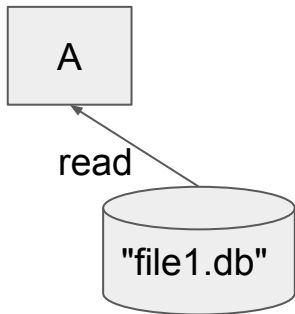
ParkingLot is coupled to Hashtable, but  
this coupling is acceptable

# Poor Coupling

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

How can a class A depend on a class B without it being via B's interface?

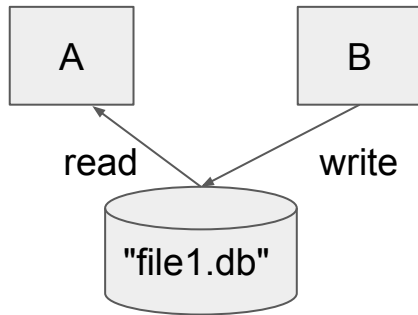
```
class A {  
    private void f() {  
        int total; ...  
        File file = File.open("file1.db");  
        total = file.readInt();  
        ...  
    }  
}
```



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

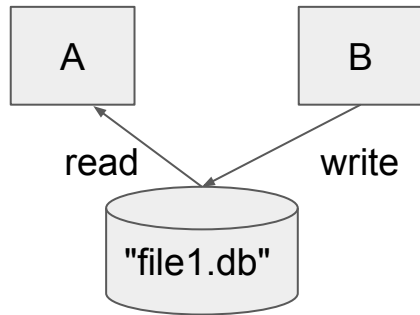


```
class B {  
    private void g() {  
        int total;  
        // computes total value  
        File file = File.open("file1.db");  
        file.writeInt(total);  
        ...  
        file.close();  
    }  
}
```



# 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



Also called evolutionary coupling  
(or logical coupling)

How to solve this problem?

How to turn poor coupling into good 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 (or eliminate) primarily poor coupling

# Summary

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

# Exercises

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

Hypothetically, propose a design for this implementation with the worst possible cohesion but, at the same time, the best possible coupling.

2. Consider the following code that performs operations on bank accounts. (a) Which design property 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:

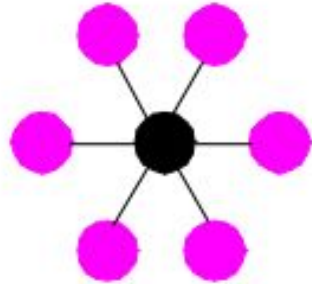
- are implemented in different directories
- A has a reference in its code to B

Whenever a developer has, as part of a maintenance task, to modify both A and B he concludes by moving B to the same directory as A.

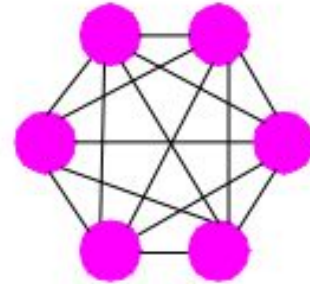
(a) By acting in this way, the developer is improving which design property (when measured across directories)?

(b) And which design property is affected negatively?

4. In general terms, which of the following designs is better? Justify (nodes = classes; edges = dependencies)

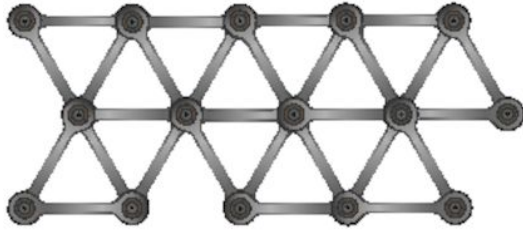


**(A)**



**(B)**

5. In general terms, which of the following designs is better? Justify (nodes = classes; edges = dependencies)



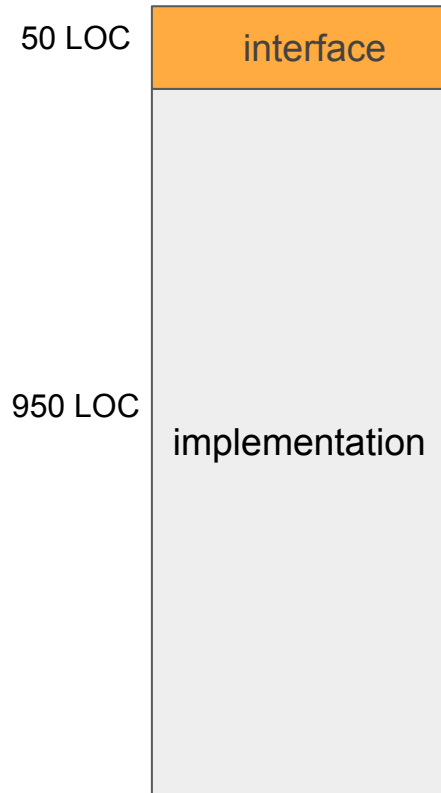
(a)



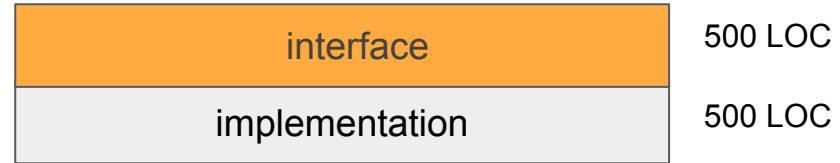
(b)



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



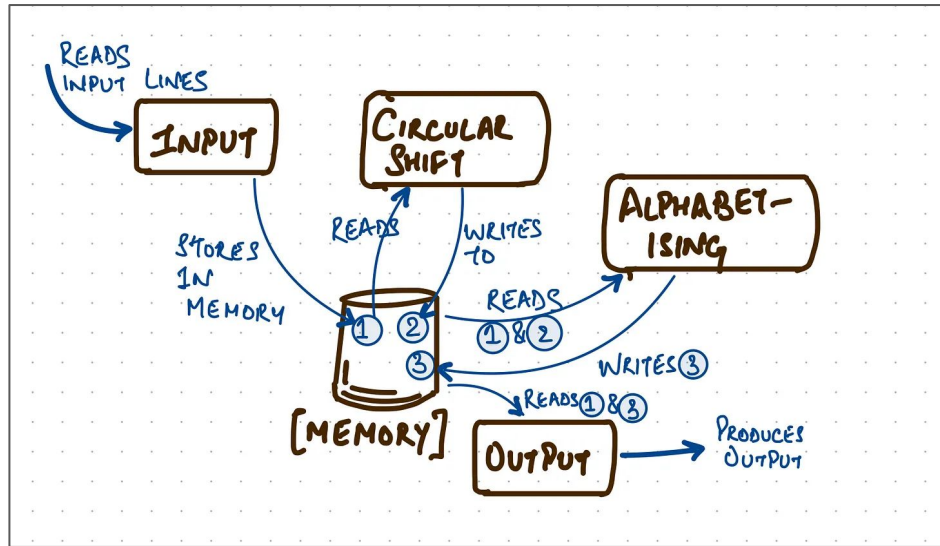
(A)



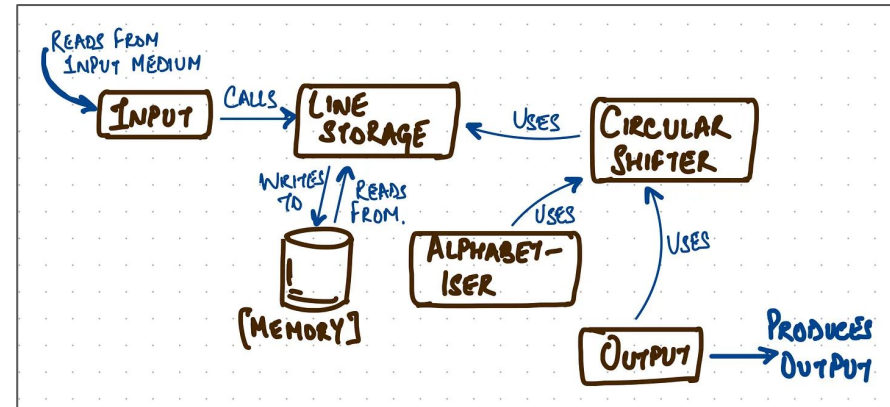
(B)

Inspired by concepts proposed in the book *A Philosophy of Software Design*. 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 meet?



Modularization I



Modularization II

# 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 (with sorted “circular shifts”)

## **Input:**

Pattern-Oriented Software Architecture  
Software Architecture  
Introducing Design Patterns

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

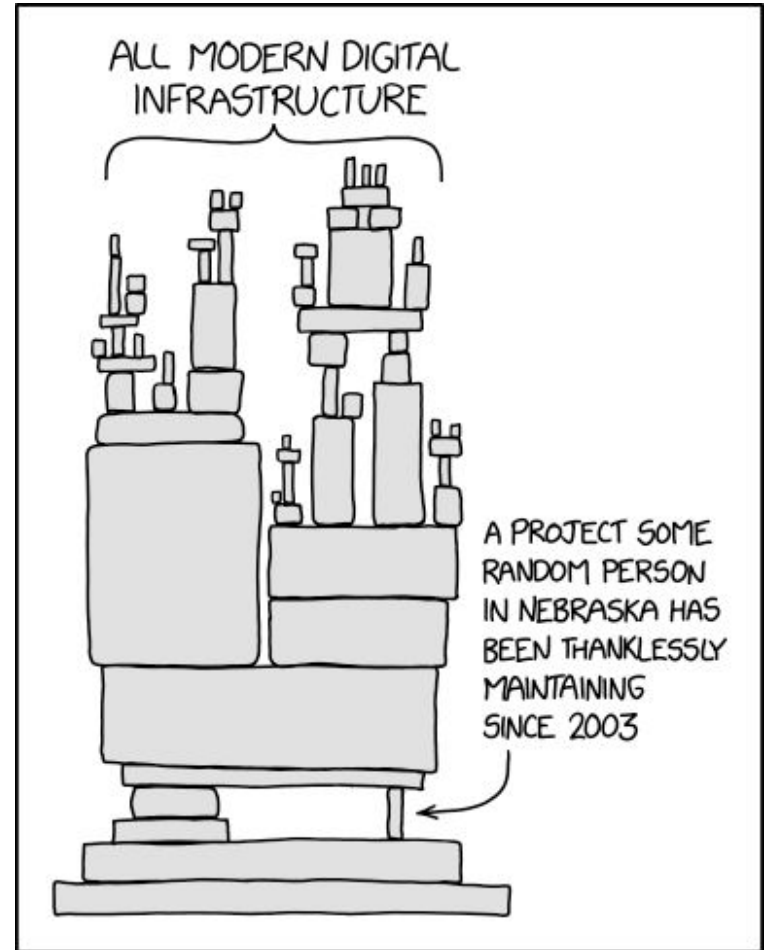
Architecture Software  
Architecture Pattern-Oriented Software  
Design Patterns Introducing  
Introducing Design Patterns  
Patterns Introducing Design  
Pattern-Oriented Software Architecture  
Software Architecture  
Software Architecture Pattern-Oriented

8. Suppose two methods `f` and `g`. There is a **temporal coupling** 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) Is the temporal coupling that exists in the following code acceptable or poor? If it's poor, suggest a refactoring to remove this type of coupling.

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

9. The following figure illustrates the concerns that can appear when not following to which design property?



# Design Principles

---

## Design Principle

---

Single Responsibility

Interface Segregation

Dependency Inversion

Favor Composition over Inheritance

Demeter

Open/Closed

Liskov Substitution

---

## Design Property

Cohesion

Cohesion

Coupling

Coupling


Information Hiding

Extensibility

Extensibility



Guidelines



Benefits (what we will gain by following the principle)



Single  
responsibility



Open-Closed  
Principle



Liskov  
substitution



Interface  
segregation



Dependency  
inversion





# **(1) Single Responsibility Principle (SRP)**

# Single Responsibility Principles

- Every class should have a single responsibility
- There should be only one reason to modify a class

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



Responsibility #1: compute dropout rate



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

Responsibility #2: print dropout rate



```
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;
    }
}
```



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

Single responsibility: user interface

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



```
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

# Advantages

- Business class (`Course`) can be used by more than one user interface class (`Console`, `WebApp`, `MobileApp` ...)
- Division of labor:
  - Interface concerns: frontend dev
  - Business concerns: backend dev



## **(2) Interface Segregation Principle (ISP)**

# Interface Segregation Principle

- Interfaces should be small, cohesive, and specific for each type of client
- Particular case of SRP, but focused on interfaces

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



# Implementation following ISP

```
interface Account {  
    double getBalance();  
}  
  
interface SavingsAccount extends Account {  
    double getInterest();  
}  
  
interface SalaryAccount extends Account {  
    int getSalary();  
}
```



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

```
interface SavingsAccount extends Account {  
    double getInterest();  
}
```

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



Common to all account

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

```
interface SavingsAccount extends Account {  
    double getInterest();  
}
```

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



Specific to SavingsAccounts

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

interface SavingsAccount extends Account {
    double getInterest();
}

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



Specific to SalaryAccounts



## **(3) Dependency Inversion Principle (DIP)**

# Dependency Inversion

- We usually call this principle **Prefer Interfaces to Classes**
- Because it better conveys its idea

# Example without using DIP

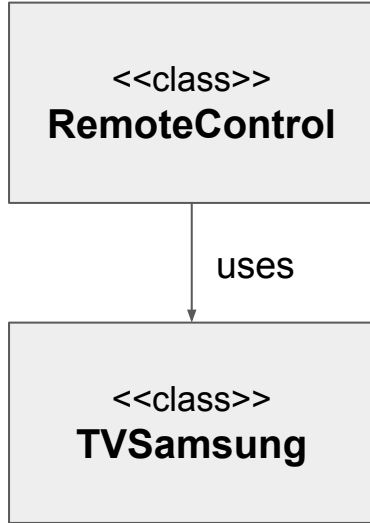
```
class RemoteControl {  
    TVSamsung tv;  
    ...  
}  
  
class TVSamsung {  
    ...  
}
```

What is the problem with this design related to coupling and extensibility?

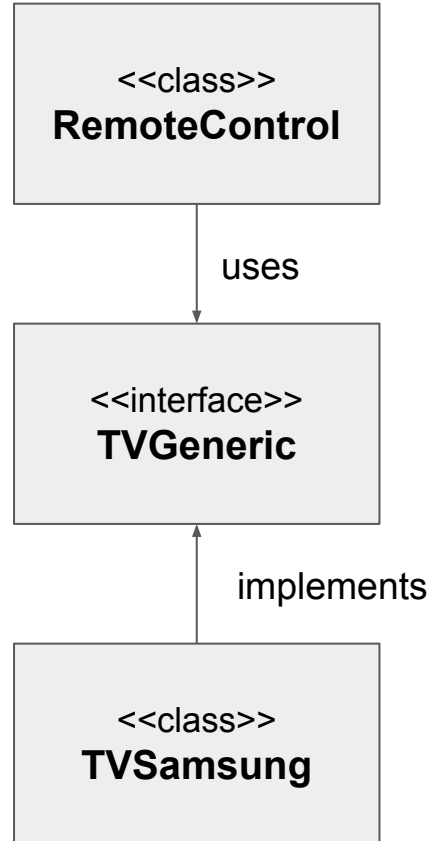
# Example using DIP

```
class RemoteControl {  
    TVGeneric tv;  
    ...  
}  
  
interface TVGeneric {  
    ...  
}  
  
class TVSamsung implements TVGeneric {  
    ...  
}
```

## Without DIP



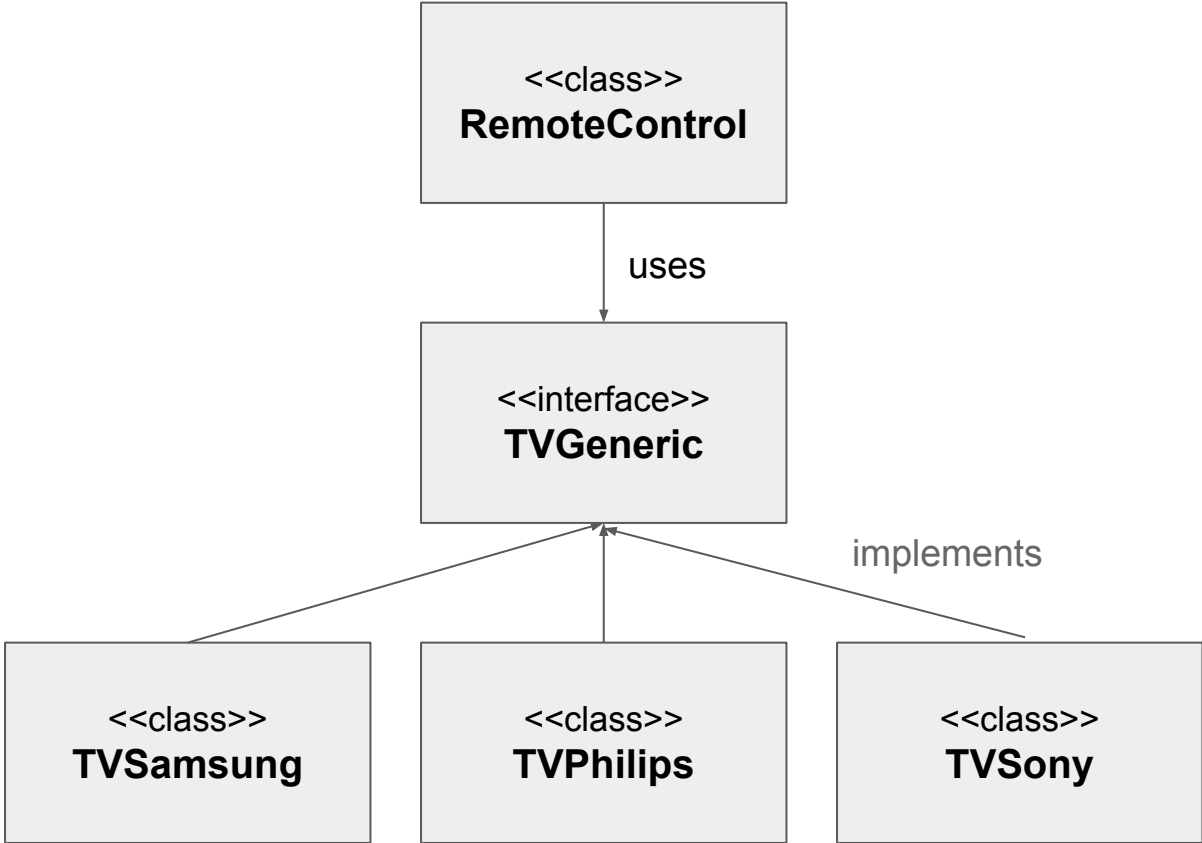
## With DIP



Advantage: generic remote control, for a generic TV

We can switch TVs, without having to change the code of the RemoteControl class

# Design after some years



## **(4) Prefer Composition to Inheritance**

# Historical Context

- In the 80s, when OO became popular, developers started to abuse on the usage of inheritance
- They thought that inheritance would be a silver bullet, promote large-scale reuse, etc.



# Inheritance: “is-a” relationship

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

# Composition: “has” relationship

- In UML, corresponds to an association

```
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:

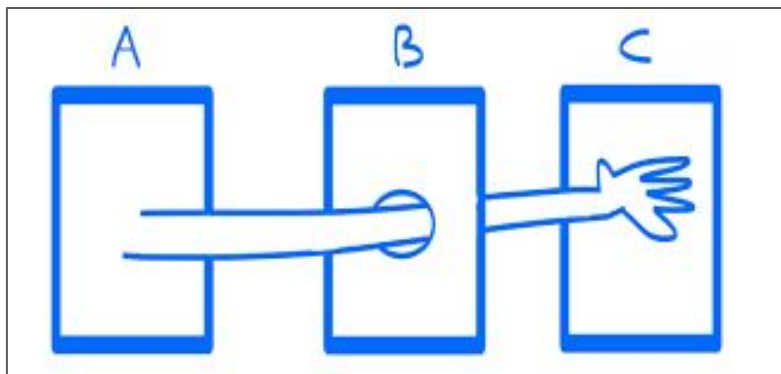
```
obj.getA().getB().getC().getD().doSomething();
```



pass-through objects

# Reason

- Long call chains break encapsulation
- We don't want to go 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 radical and assume that method call chains are always prohibited
- Particula cases may exist and have a good justification



# Acceptable example of method chaining

```
const numbers = [1, 2, 3, 4, 5];  
const result = numbers  
  .map(num => num * 2)  
  .filter(num => num % 3 === 0)  
  .reduce((acc, num) => acc + num, 0);  
console.log(result); // Output: 12 (2*3 + 4*3)
```



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 modifications, but **open** for extensions



# Explaining further

- Suppose you are going to implement a class
- Clients will want to use your class (obvious!)
- But they will also want to customize, configure, and extend it!
- You should anticipate and make such extensions possible
- Goal: avoid clients having to edit your class to customize it

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

- Parameters
- Design patterns
- Inheritance
- Higher-order functions
- 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"]
```



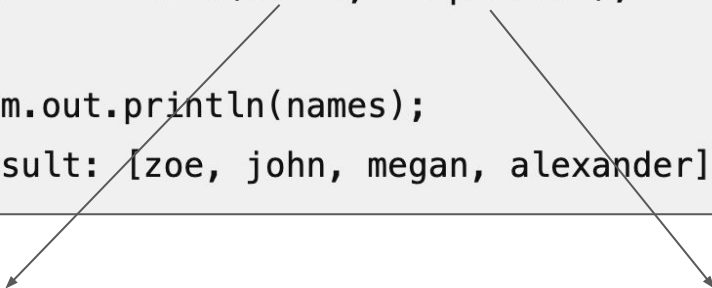
But now one user (developer) wants to sort the  
the list elements by their length (# of chars)

Is the sort method `open` (prepared) to this extension?

But keeping its code `closed`, i.e., without having to change it



```
Comparator<String> comparator = new Comparator<String>() {  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
};  
Collections.sort(names, comparator);  
  
System.out.println(names);  
// result: [zoe, john, megan, alexander]
```



list of  
strings

Object with a method to compare two strings.  
There's no free lunch: client has to implement this method

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 using inheritance
- Specifically, for redefining methods in subclasses



First: let's understand the term substitution

```
void f(A a) {
```

```
    ...
```

```
    a.g();
```

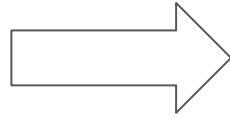
```
    ...
```

```
}
```

```
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 the same services as A
- For code that uses A, the substitution is 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 `bib`) to remove this violation?

```
bib.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 does not follow the Open/Closed Principle, since in our system it is important to configure the header and footer messages. How would you change the implementation of this class to follow this principle?

3. Suppose a `Calculator` with a method that checks if a number between 0 and 10,000 is prime. Also suppose that a more efficient algorithm was implemented in a subclass `FastCalculator`. However, 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 design principle is violated in this implementation? Justify.

4. Suppose you finished an outreach course offered by UFMG 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 Center of Extension (CENEX)
- Then, you sent a mail to CENEX, who asked you to send a mail to Pro-Rectorate for Extension (PROEX)
- Then, you sent a mail to PROEX, who returned your certificate.

These steps illustrate a violation of which design principle? Besides sending several mails, what's another problem with this solution?



5. In Software Engineering, we sometimes use more complex solutions when this is not necessary. This problem is called **overengineering** or **premature optimization**.

Thus, describe a context in which the use of one of the design principles that we studied is as a premature optimization.

If you want, you can reuse examples from the previous slides.

**End**