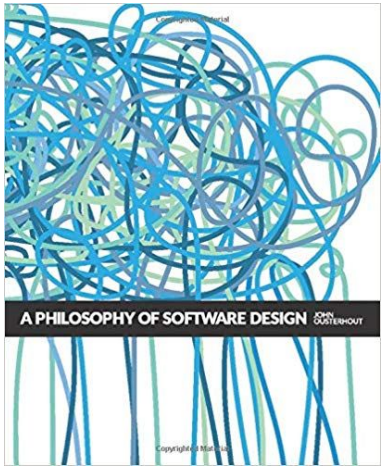
Software Design **Properties**

CSCI 4440U | Topics in Computer Science:
AI-Powered Software Engineering

Cristiano Politowski

Adapted from: Marco Tulio Valente, Software Engineering: A Modern Approach, 2024.

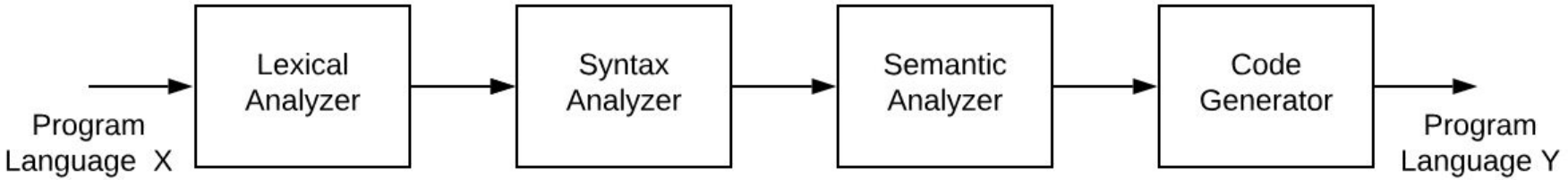
"The most fundamental problem in computer science is problem decomposition: how to take a complex problem and divide it up into pieces that can be solved independently."
— John Ousterhout



Definition

- Ousterhout's quote is an excellent definition of design
- Software design: breaking a "big problem" into smaller parts
- Implementing the smaller parts leads to implementing the "big problem"

Example: Compiler



Modules

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

What we will study

- Design Properties
- Design Principles

Design Properties

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

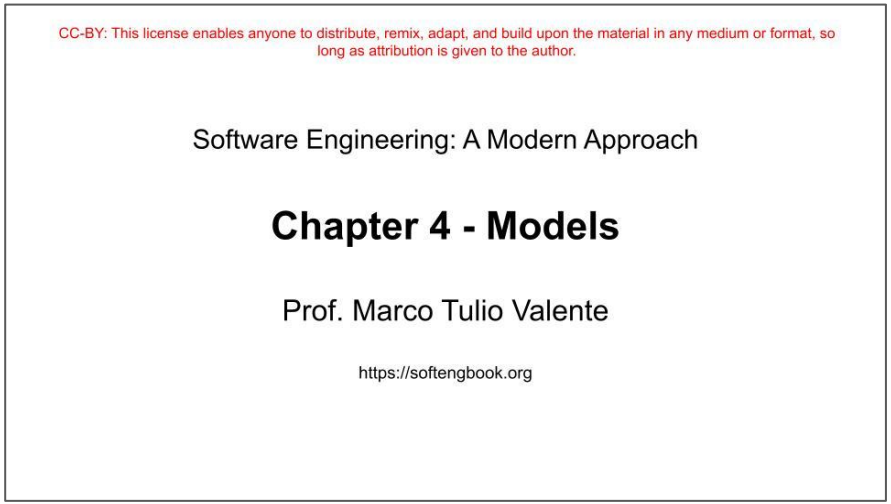
Design Principles

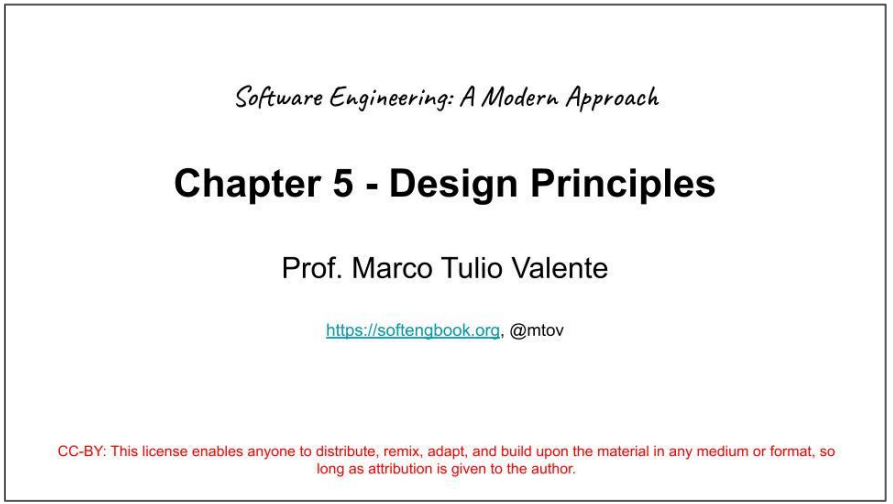
1. Single Responsibility
2. Interface Segregation
3. Prefer Interfaces to Classes (aka Dependency Inversion)
4. Prefer Composition to Inheritance
5. Open/Closed
6. Demeter
7. Liskov Substitution

Design Properties

Conceptual Integrity

Why do these slides lack conceptual integrity?





Conceptual Integrity applies to:

- User interface
- Design decisions
- Implementation decisions
- Technological decisions
- and so on

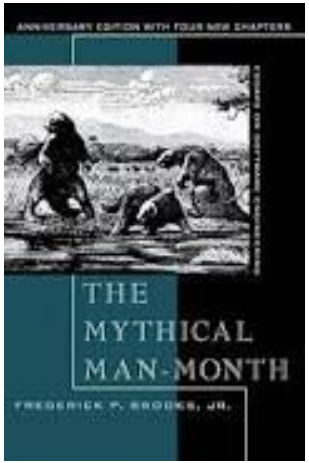
Examples (referring to user interface)

- 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 numeric results should be shown with two 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 one problem is solved using data structure X, all similar problems should also use X

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



Reason: Conceptual integrity makes a system easier to use and understand

Information Hiding

Origin of this property: David Parnas (1972)

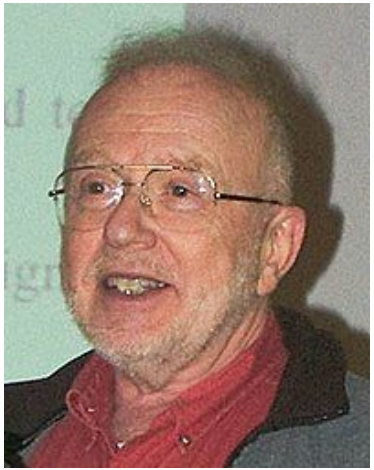
On the Criteria To Be Used in Decomposing Systems into Modules

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");
  }
}
```

```
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 must manipulate an internal data structure to register a vehicle for parking

Problem

- Classes need some degree of privacy.
- This enables them to evolve independently of other classes
- In the previous code, client code directly accessed the hash table

Comparison with a manual parking control system

- Customers must enter the parking lot booth and write their car data in the logbook

Implementation with information hiding

```
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");
  }
}
```

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");
  }
}
```

3

ParkingLot is now free to change its internal data structures

Information Hiding

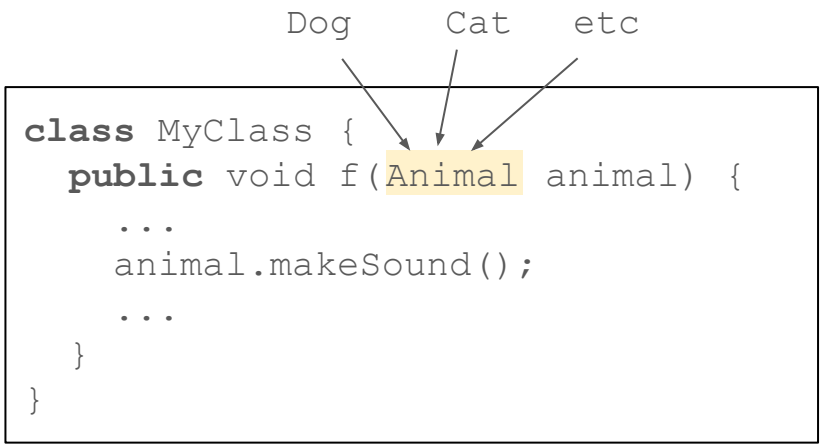
- Classes should hide their internal implementation details
  - Use the **private** modifier
  - Especially for elements likely to change over time
- The class interface should remain stable
- Interface = the set of public methods and attributes of a class

Meanings of the word interface

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

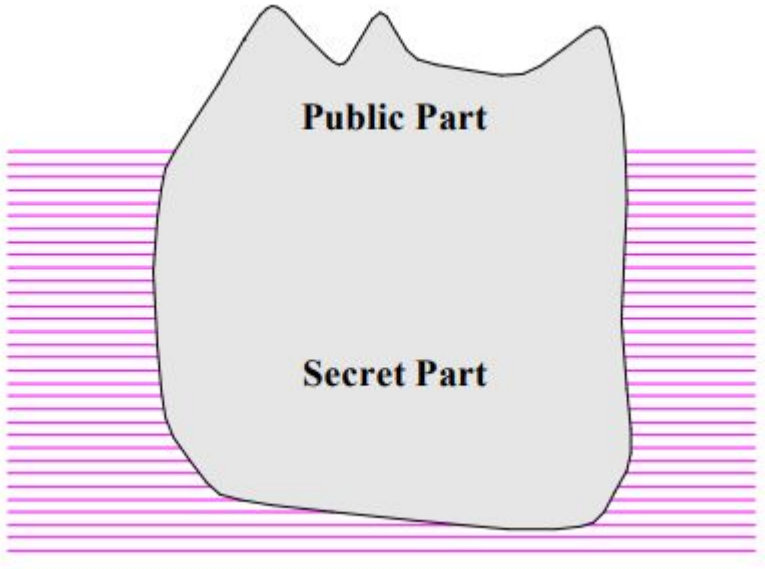
Interface in Java

```
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!");
  }
}
```



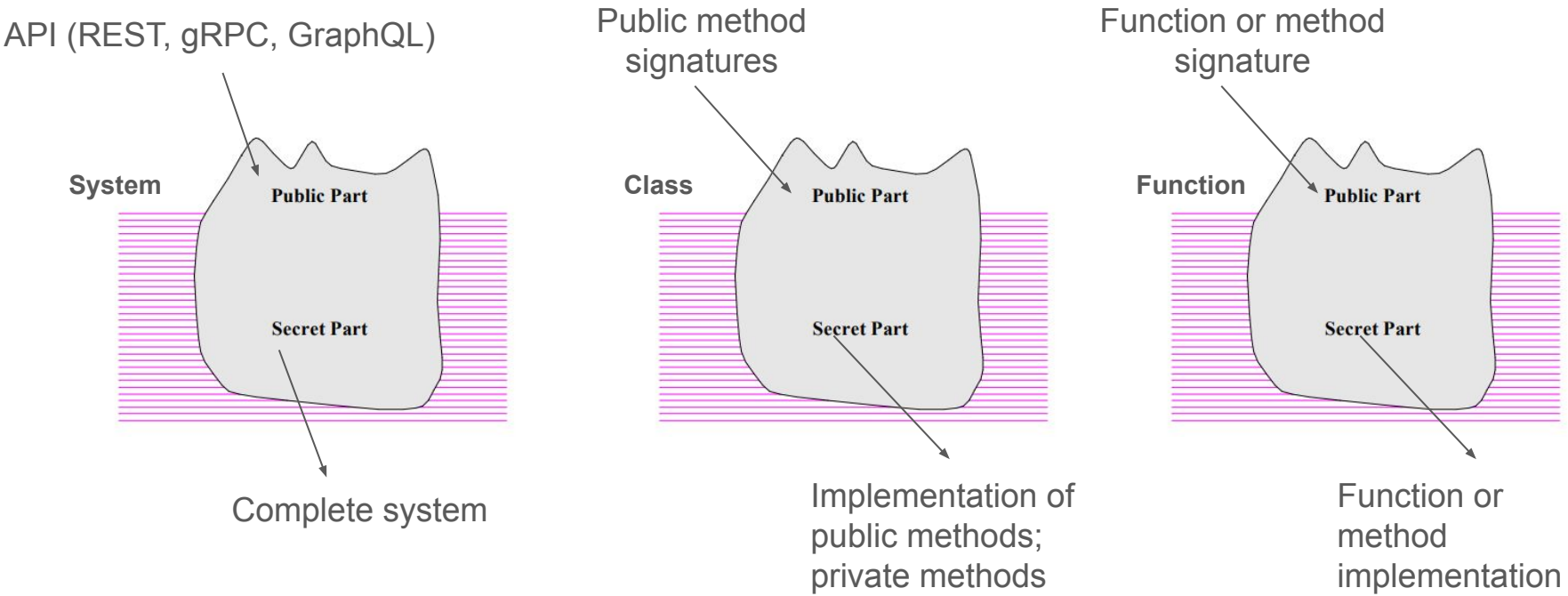
Good modules are like icebergs:

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



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

Generalizing to systems, classes, and functions



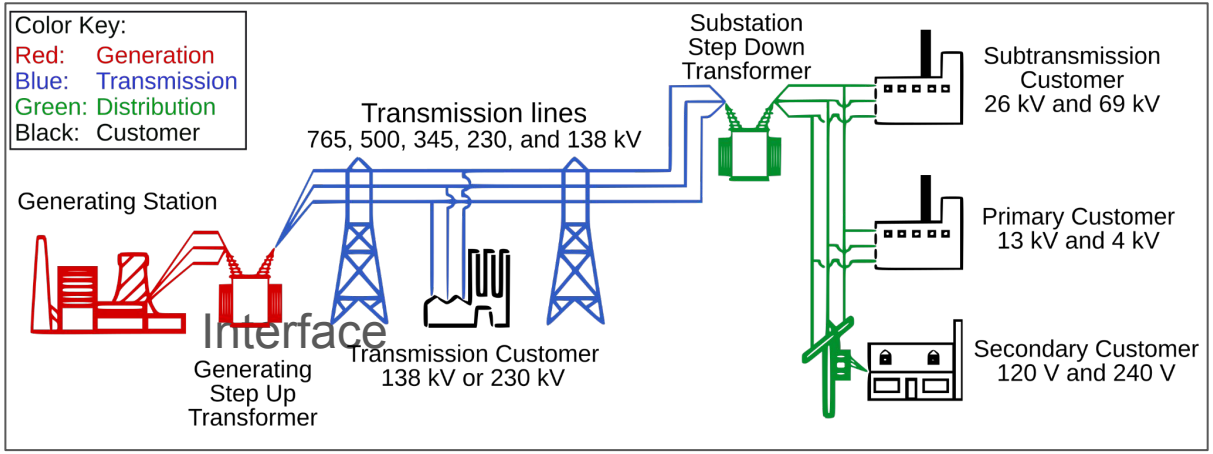
Another name: encapsulation

- Some authors use the term “encapsulation” with the same meaning as 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 also applies to functions, methods, and packages

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 split into two functions: sin and cos

Counter-example 2

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

A separate Manager class should be extracted to hold manager data

Example

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

All these methods operate on Stack elements

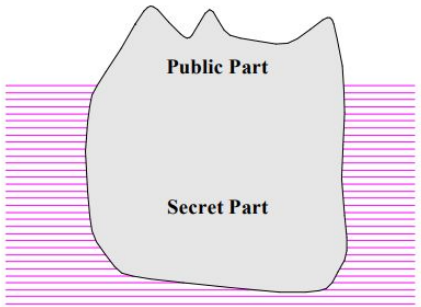
Coupling

Coupling

- No class is an island — classes depend on each other
- They call methods of other classes or extend them
- The key concern is the quality of the coupling
- Types of coupling:
  - Acceptable coupling ("good")
  - Poor coupling ("bad")

Acceptable Coupling

- Class A uses class B and:
  - B provides a useful service to A
  - B has a stable interface
  - A interacts with B only via 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");
  }
}
```

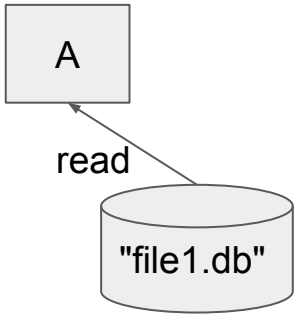
ParkingLot is coupled to Hashtable, but this coupling is acceptable

Poor Coupling

- Class A uses class B, but
  - B's interface is unstable
  - Or A does not use B through its interface

How can class A be coupled to class B without using B's interface?

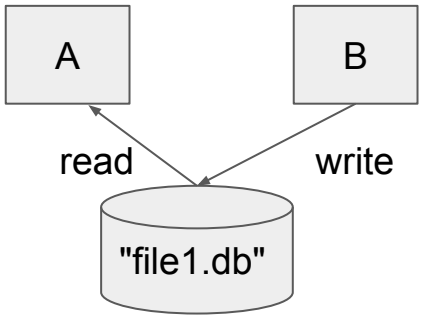
```
class A {
  private void f() {
    int total; ...
    File file = File.open("file1.db");
    total = file.readInt();
    ...
  }
}
```



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

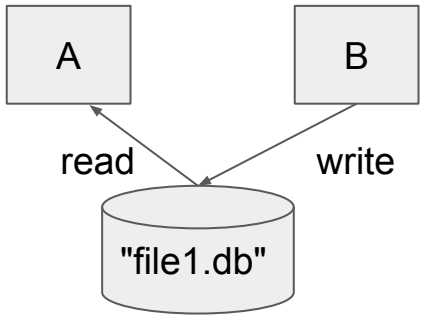


```
class B {
  private void g() {
    int total;
    // computes total value
    File file = File.open("file1.db");
    file.writeInt(total);
    ...
    file.close();
  }
}
```

Poor Coupling

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



This is also called evolutionary (or logical) coupling

How can we solve this problem? How can we refactor poor coupling into acceptable coupling?

Refactoring poor coupling 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: focus on minimizing poor coupling

Summary

- Static (structural) coupling:
  - A’s code contains an explicit reference to B
  - Can be acceptable or poor
- Evolutionary (logical) coupling:
  - A’s code contains no reference to B
  - But changes in B can still impact A
  - Always considered poor coupling

Exercises