# Good Object-Oriented Design Practices and the Observer Design Pattern

Dr. Rodrigo Spínola

## Last time

**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.

We also formalized good method design characteristics using the SOFA: methods are <u>s</u>hort, do <u>o</u>ne thing, have <u>f</u>ew arguments, single level of <u>a</u>bstraction.

## Agenda

- Cohesion and Coupling
- Design Patterns
  - Observer Design Pattern

## Cohesion and Coupling

## Understanding Cohesion and Coupling

- Let's take as an example a **Smartphone**
- Any product can be broken down into two constituents:
  - **Components:** Individual elements that perform a single task. In our Smartphones, individual components are RAM, ROM, Light Sensor, Lens, Image Processor, CPU, GPU, etc.
  - **Modules:** Groups of functionally similar components. Smartphone components are grouped as Memory (RAM, ROM), Camera (Light Sensor, Lens, Image Processor), System-on-Chip (CPU, GPU), etc.

**Cohesion and Coupling define how these modules of a product are formed and how dependent they are on each other.**

## Cohesion

**Cohesion** is a measure of the **degree** to which the **elements of a module are functionally related**. In other words, Cohesion defines how closely the components of a given module are related to each other with respect to their functionality.

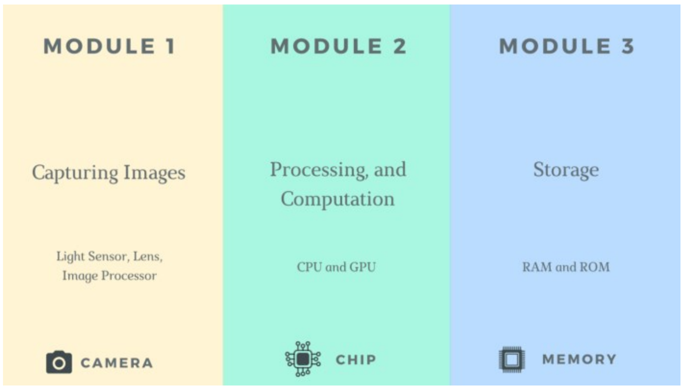
**High Cohesion** is when the components of a module are directed towards performing a **single task**.

**Low Cohesion** is the exact opposite of High Cohesion. Here, each module ends up performing more than one task since its components are not directed towards a single task.

Should we avoid high or low cohesion?

## High Cohesion

- For example, components of the Camera module, Light Sensor, Lens, Image Processor are all directed towards performing a single task of capturing images.



## Coupling

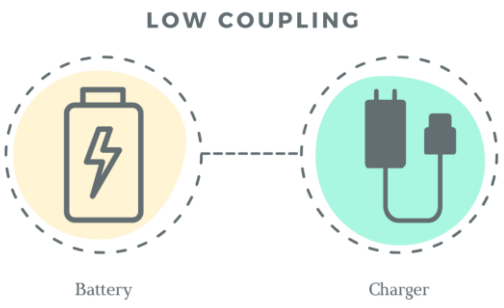
**Coupling** is the measure of the degree of **interdependence between modules**. In simple terms, coupling defines how heavily two different modules are dependent on each other.

**Low/Loose Coupling** is when two modules are **very lightly dependent on each other** to perform their duties.

**High/Tight Coupling** is when two modules are **heavily dependent on each other** to perform their duties.

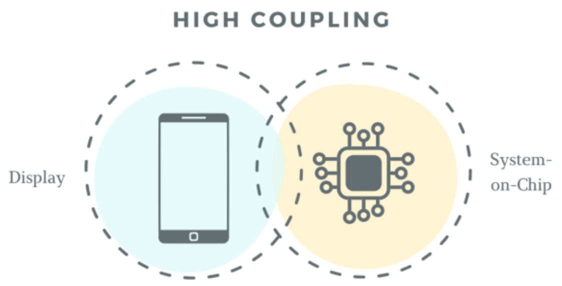
## Low/Loose Coupling Example

- The Battery and Battery-charger are loosely coupled. If the Battery-charger gets corrupted, another Battery-charger can be used to charge the Battery, and if the Battery gets corrupted, then the same Battery-charger can still be used to charge other phones batteries.

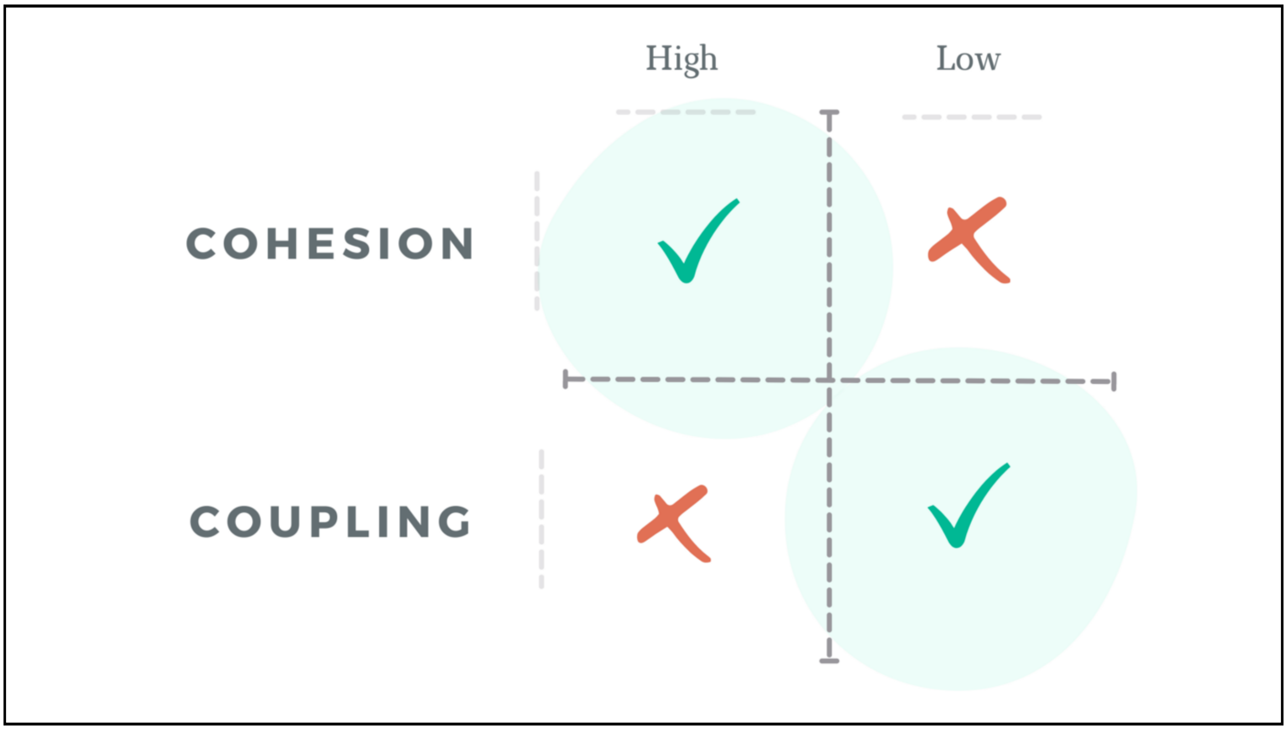


## High/Tight Coupling Example

- The Display module and System-on-Chip module are more tightly coupled. If the System-on-Chip module gets corrupted, the Display module will not be able to perform its duties and if the Display gets corrupted, the System-on-chip (GPU) will not be able to perform its duties properly.



Should we avoid high or low coupling?

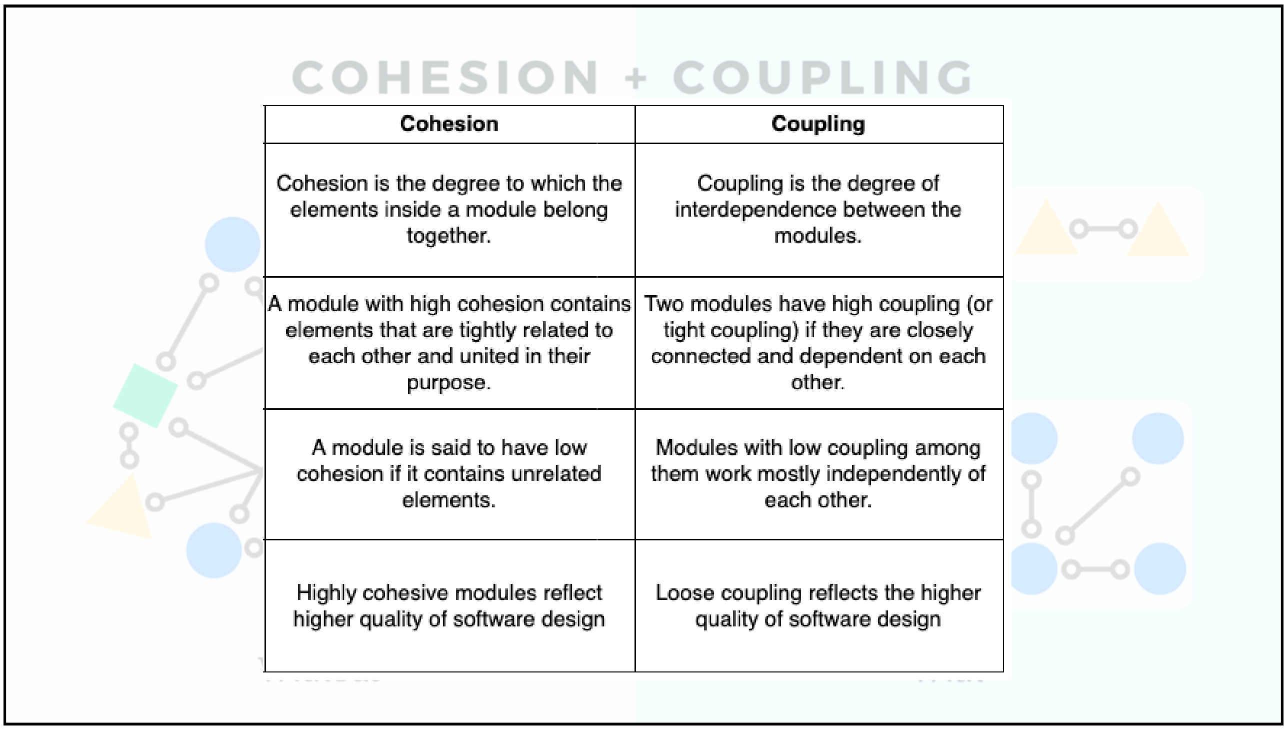


|  | High | Low |
|---|---|---|
| COHESION | ✓ | ✗ |
| COUPLING | ✗ | ✓ |

What are the advantages of having high cohesion and low coupling in software projects?

## High Cohesion and Low Coupling is the principle

- There is tremendous value in striving for High Cohesion and Low Coupling:
  - **Easier troubleshooting:** when a product malfunctions, it is now easier to locate the module that is causing the problem
  - **Easier fixing:** a malfunctioning module can easily be replaced without having to touch other modules
  - **Easier testing:** since we know the exact module that has been replaced and what other modules it is coupled with, we no longer have to test the entire product.



COHESION + COUPLING

| Cohesion | Coupling |
|---|---|
| Cohesion is the degree to which the elements inside a module belong together. | Coupling is the degree of interdependence between the modules. |
| A module with high cohesion contains elements that are tightly related to each other and united in their purpose. | Two modules have high coupling (or tight coupling) if they are closely connected and dependent on each other. |
| A module is said to have low cohesion if it contains unrelated elements. | Modules with low coupling among them work mostly independently of each other. |
| Highly cohesive modules reflect higher quality of software design | Loose coupling reflects the higher quality of software design |

## Design Patterns

## Design Patterns

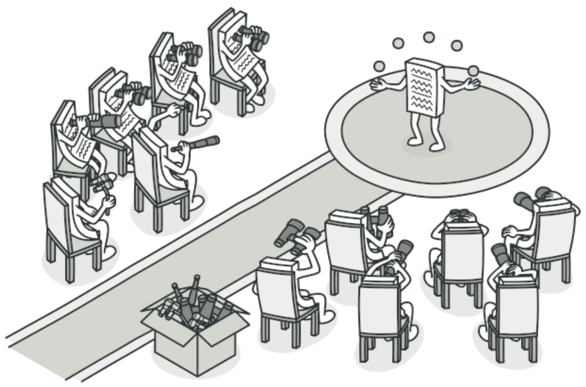
**Design patterns** are **typical solutions** to **common problems** in software design. Each pattern is **like a blueprint** that you can **customize** to solve a particular design problem in your code.

- You can't just find a pattern and copy it into your program. The pattern is not a specific piece of code, but a general concept for solving a particular problem. You can follow the pattern details and implement a solution that suits the realities of your own program

## What does the pattern consist of?

- Most patterns are described very formally so people can reproduce them in many contexts. Here are the sections that are usually present in a pattern description:
  - **Intent** of the pattern briefly describes both the problem and the solution
  - **Motivation** further explains the problem and the solution the pattern makes possible
  - **Structure of classes** shows each part of the pattern and how they are related
  - **Code example** in one of the popular programming languages makes it easier to grasp the idea behind the pattern

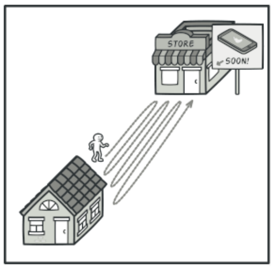
Design patterns are a **toolkit of tried and tested solutions** to common problems in software design. Even if you never encounter these problems, knowing patterns is still useful because it teaches you how to solve all sorts of problems using principles of object-oriented design.

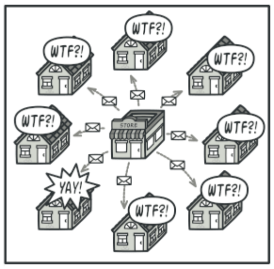


## The Observer Design Pattern

## Problem

- Imagine that you have two types of objects: a **Customer** and a **Store**. The customer is very interested in a particular brand of product (say, it's a new model of the iPhone) which should become available in the store very soon
- The customer could visit the store every day and check product availability. But while the product is still en route, most of these trips would be pointless
- On the other hand, the store could send tons of emails to all customers each time a new product becomes available. This would save some customers from endless trips to the store. At the same time, it'd upset other customers who aren't interested in new products.





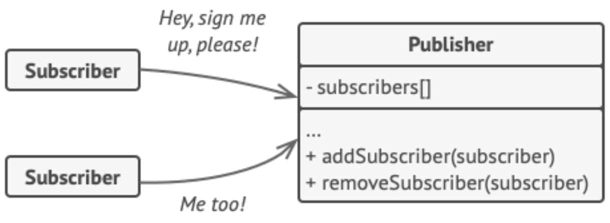
Bad solution

## Solution

- If you subscribe to a newspaper or magazine, you no longer need to go to the store to check if the next issue is available. Instead, the publisher sends new issues directly to your mailbox right after publication or even in advance

## Solution

- The object that has some interesting state is often called **subject**, but since it's also going to notify other objects about the changes to its state, we'll call it **publisher**. All other objects that want to track changes to the publisher's state are called **subscribers**.

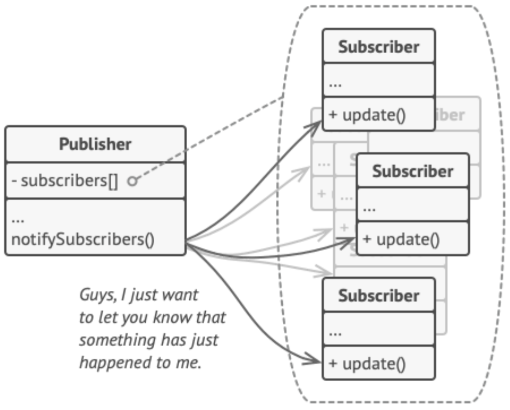


*Now, whenever an important event happens to the publisher, it goes over its subscribers and calls the specific notification method on their objects.*

## But...

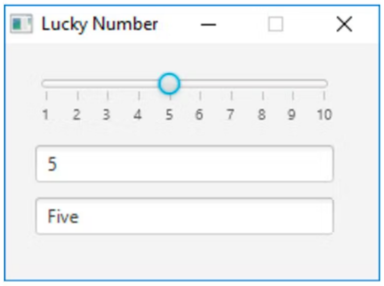
- Real apps might have dozens of different subscriber classes that are interested in tracking events of the same publisher class. You wouldn't want to **couple** the publisher to all of those classes. Besides, you might not even know about some of them beforehand if your publisher class is supposed to be used by other people.

- That's why it's crucial that all subscribers implement the same **interface** and that the publisher communicates with them only via that interface.



## Motivation

- Consider this application which selects (and shows) a number in 3 different ways

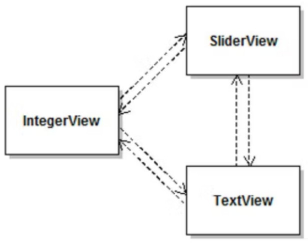


TDresearchteam
Technical Debt Research Team

VCU Computer Science
College of Engineering

## Motivation

- Naive way to implement this application has complete pairwise dependencies
  - High coupling: each pane explicitly depends on many other panels
  - Complexity: complex program logic is required to keep the different panels consistent
  - Low extensibility: to add or remove a panel, it is necessary to modify all other panels

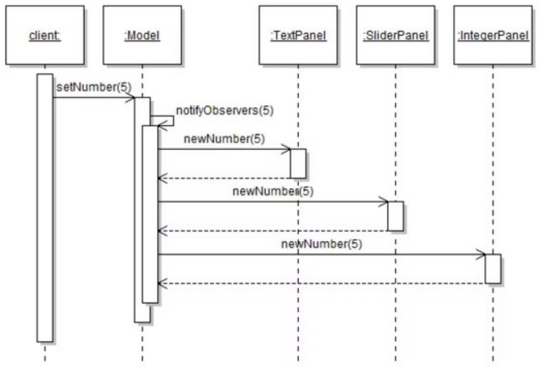


TDresearchteam
Technical Debt Research Team

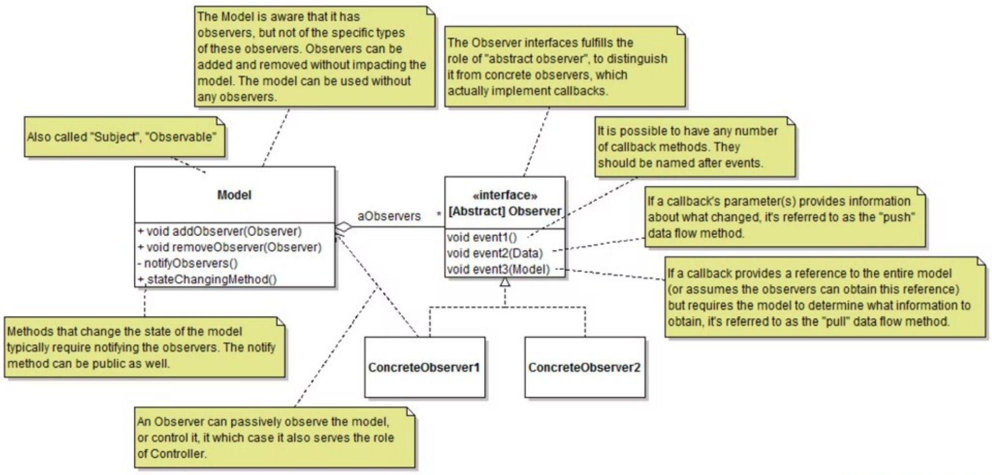
VCU Computer Science
College of Engineering

## Observer

- How do the observers learn that there is new information in the model that they need to know about?



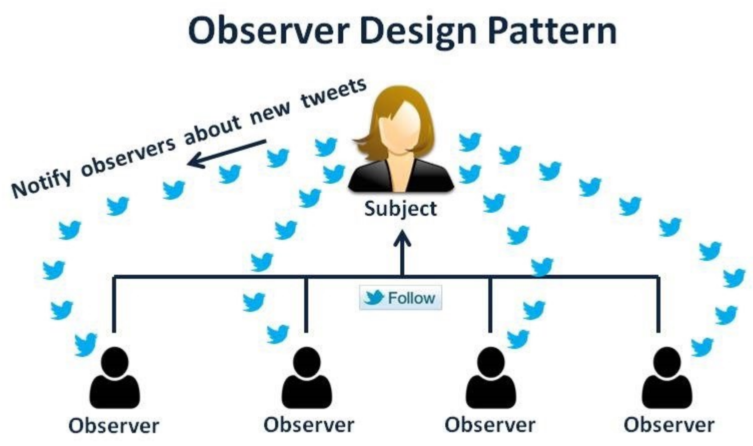
## Observer


The Model is aware that it has observers, but not of the specific types of these observers. Observers can be added and removed without impacting the model. The model can be used without any observers.
The Observer interfaces fulfills the role of "abstract observer", to distinguish it from concrete observers, which actually implement callbacks.
Also called "Subject", "Observable"
It is possible to have any number of callback methods. They should be named after events.
Model
+ void addObserver(Observer)
+ void removeObserver(Observer)
- notifyObservers()
+ stateChangingMethod()
aObservers
*
«interface»
[Abstract] Observer
void event1()
void event2(Data)
void event3(Model)
If a callback's parameter(s) provides information about what changed, it's referred to as the "push" data flow method.
If a callback provides a reference to the entire model (or assumes the observers can obtain this reference) but requires the model to determine what information to obtain, it's referred to as the "pull" data flow method.
Methods that change the state of the model typically require notifying the observers. The notify method can be public as well.
ConcreteObserver1
ConcreteObserver2
An Observer can passively observe the model, or control it, it which case it also serves the role of Controller.

## Observer

- The main purpose of this pattern is for example when the state of an object changes, so that all the dependencies of this object are notified and updated
- Observer pattern is used in Facebook when, for example, you get a notification about an event or when you receive a notification, if someone else commented on a post or, for example, YouTube when someone subscribes to a channel

## Another well-known example

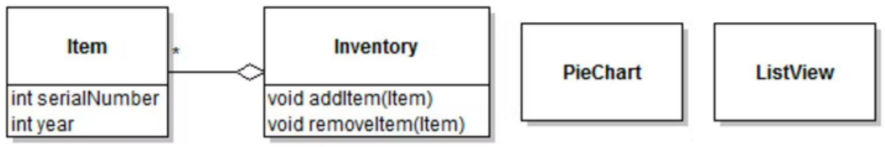


## Observer Case Study

- We are interested in an inventory system capable of keeping track of electronic equipment. An **item** of equipment records a serial number and production year. An **inventory** object aggregates a bunch of **items**. Clients can add or remove items from the inventory at any time. Various entities are interested in changes to the state of the inventory.
- For example, it should be possible to show items in the inventory in a **ListView**. It should also be possible to view a **PieChart** representing the proportion of items in the inventory for each production year (e.g., 2004=25%; 2005=30%, etc.). Views should be updated whenever items are added or removed from the inventory.

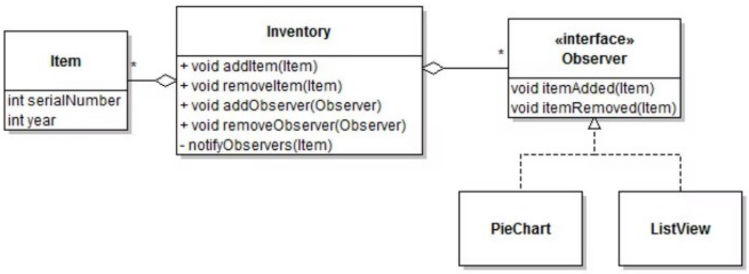
## Observer Case Study

- Class we would need



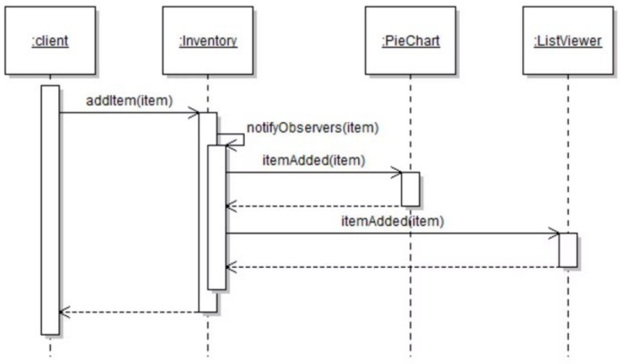
## Observer Case Study

- Adding in Observer



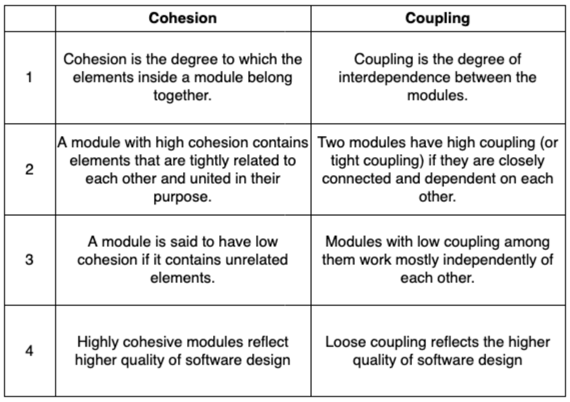
## Observer Case Study

- Adding in Observer



## Summary

- Cohesion and Coupling
- Observer Design Pattern



| | Cohesion | Coupling |
|---|---|---|
| 1 | Cohesion is the degree to which the elements inside a module belong together. | Coupling is the degree of interdependence between the modules. |
| 2 | A module with high cohesion contains elements that are tightly related to each other and united in their purpose. | Two modules have high coupling (or tight coupling) if they are closely connected and dependent on each other. |
| 3 | A module is said to have low cohesion if it contains unrelated elements. | Modules with low coupling among them work mostly independently of each other. |
| 4 | Highly cohesive modules reflect higher quality of software design | Loose coupling reflects the higher quality of software design |

Class is over, questions?

# Good Object-Oriented Design Practices and the Observer Design Pattern

Dr. Rodrigo Spínola

Lecture 25 - Good OO Design Practices

TDresearchteam
Technical Debt Research Team

VCU Computer Science
College of Engineering