

ANTI PATTERNS & CODE SMELL

Software Modeling

Author: Eng. Carlos Andrés Sierra, M.Sc.
carlos.andres.sierra.v@gmail.com

Computer Engineer
Lecturer
Universidad Distrital Francisco José de Caldas

2024-I



Outline

- 1 Model-View-Controller Pattern
- 2 Design Principles underlying Design Patterns
- 3 Anti-Patterns & Code Smells



Outline

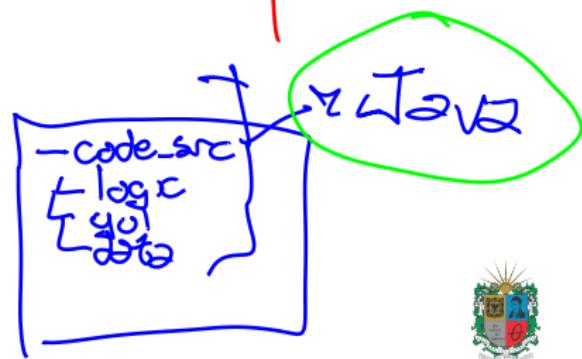
1 Model-View-Controller Pattern

late 80s
90s
2002

2 Design Principles underlying Design Patterns

3 Anti-Patterns & Code Smells

- No dead
- No layers
- More logical Architectures



MVC Pattern

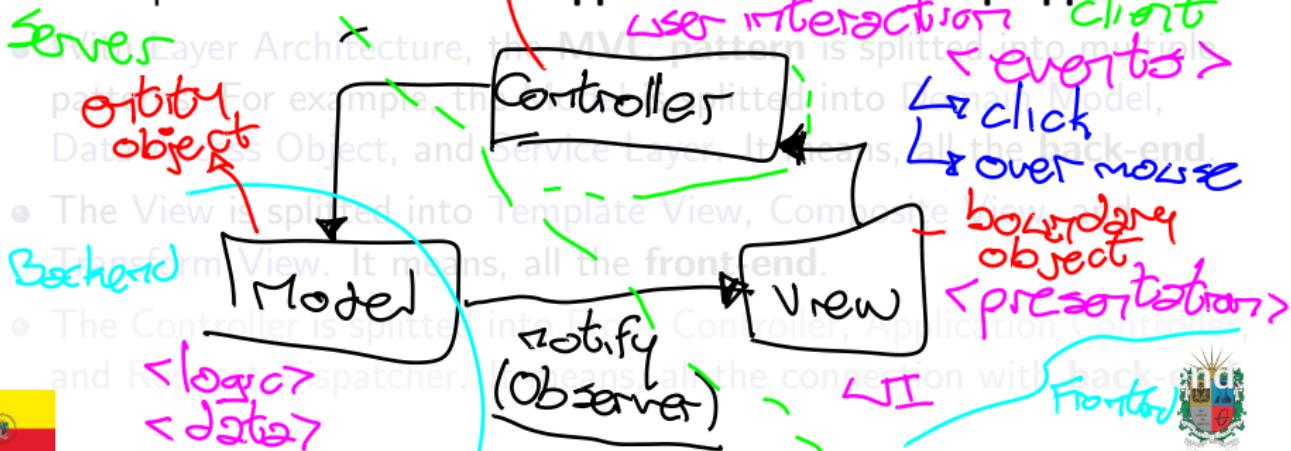
- **Model-View-Controller** is a software design pattern. It is used to separate the concerns of an application. It means, modularize the application with loose coupling.
- It is used to separate the data (Model), the presentation (View), and the user interaction (Controller) of an application. Currently, the MVC pattern is used in web applications and desktop applications.
- With Layer Patterns, the MVC pattern is splitted into multiple patterns. For example, the Model is splitted into Domain Model, Data Access Object, and Service Layer. It means, all the back-end.
- The View is splitted into Template View, Composite View, and Transform View. It means, all the front-end.
- The Controller is splitted into Front Controller, Application Controller, and Request Dispatcher. It means, all the connection with back-end.



MVC Pattern

control objects

- **Model-View-Controller** is a software design pattern. It is used to separate the concerns of an application. It means, **modularize** the application with **loose coupling**.
- It is used to separate the **data** (**Model**), the **presentation** (**View**), and the **user interaction** (**Controller**) of an application. Currently, the MVC pattern is used in **web applications** and **desktop applications**.



MVC Pattern

- **Model-View-Controller** is a **software design pattern**. It is used to separate the concerns of an application. It means, **modularize** the application with **loose coupling**.
- It is used to separate the **data (Model)**, the **presentation (View)**, and the **user interaction (Controller)** of an application. Currently, the MVC pattern is used in **web applications** and **desktop applications**.
- With Layer Architecture, the **MVC pattern** is splitted into multiple patterns. For example, the **Model** is splitted into **Domain Model**, **Data Access Object**, and **Service Layer**. It means, all the **back-end**.
- The **View** is splitted into **Template View**, **Composite View**, and **Transform View**. It means, all the **front-end**.
- The Controller is splitted into **Front Controller**, **Application Controller**, and **Request Dispatcher**. It means, all the connection with **back-end**.



MVC Pattern

- **Model-View-Controller** is a **software design pattern**. It is used to separate the concerns of an application. It means, **modularize** the application with **loose coupling**.
- It is used to separate the **data (Model)**, the **presentation (View)**, and the **user interaction (Controller)** of an application. Currently, the MVC pattern is used in **web applications** and **desktop applications**.
- With **Layer Architecture**, the **MVC pattern** is splitted into multiple patterns. For example, the **Model** is splitted into **Domain Model**, **Data Access Object**, and **Service Layer**. It means, all the **back-end**.
- The **View** is splitted into **Template View**, **Composite View**, and **Transform View**. It means, all the **front-end**.
- The **Controller** is splitted into **Frontend Controller**, **Application Controller**, and **Request Dispatcher**. It means, all the connection with back-end.



MVC Pattern



User is not in format

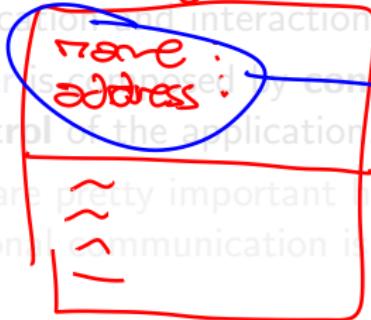
- **Model-View-Controller** is a software design pattern. It is used to separate the concerns of an application. It means, modularize the application with loose coupling.
- It is used to separate the **data (Model)**, the **presentation (View)**, and the **user interaction (Controller)** of an application. Currently, the MVC pattern is used in **web applications** and **desktop applications**.
- With Layer Architecture, the **MVC pattern** is splitted into multiple patterns. For example, the **Model** is splitted into Domain Model, Data Access Object, and Service Layer. It means, all the **back-end**.
- The **View** is splitted into Template View, Composite View, and Transform View. It means, all the **front-end**.
- The **Controller** is splitted into Front Controller, Application Controller, and Request Dispatcher. It means, all the connection with **back-end**.



MVC Implementation

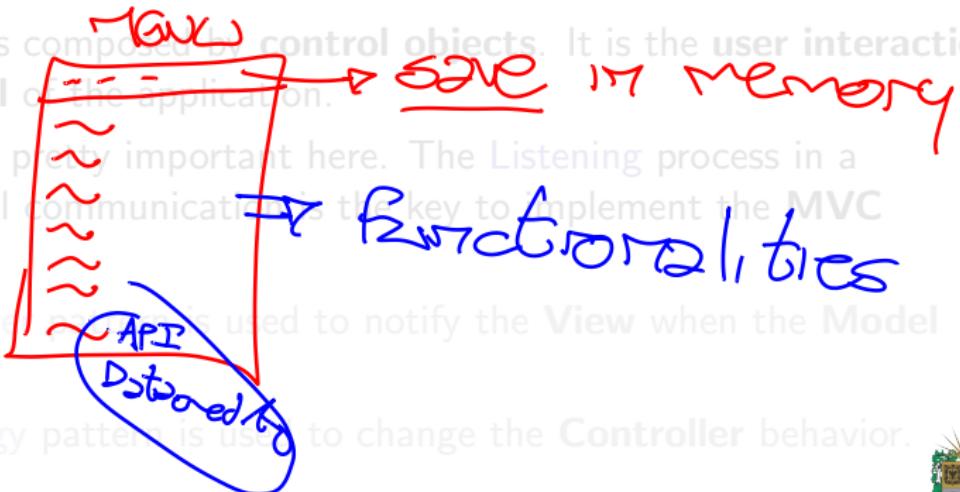
objects

- **Model** is composed by **entity models**. It is the **data** and **logic** of the application.
- View is composed by **boundary objects**. It is the presentation of the application's interaction with external elements as users.
- Controller is composed by **control objects**. It is the interaction and **control** of the application.
- Sockets are pretty important here. The Listening process in a bidirectional communication is the key to implement the **MVC** pattern.
- The Observer pattern is used to notify the **View** when the **Model** changes.
- The Strategy pattern is used to change the **Controller** behavior.



MVC Implementation

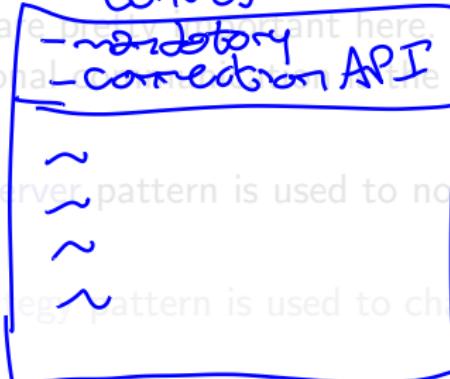
- Model is composed by **entity models**. It is the **data** and **logic** of the application.
- View is composed by **boundary objects**. It is the **presentation** of the application and interaction with external elements as **users**.
- Controller is composed by **control objects**. It is the **user interaction** and **control** of the application.
- Sockets are pretty important here. The Listening process in a bidirectional communication is the key to implement the **MVC** pattern.
- The Observer pattern is used to notify the View when the Model changes.
- The Strategy pattern is used to change the Controller behavior.



MVC Implementation

FastAPI → Pydantic

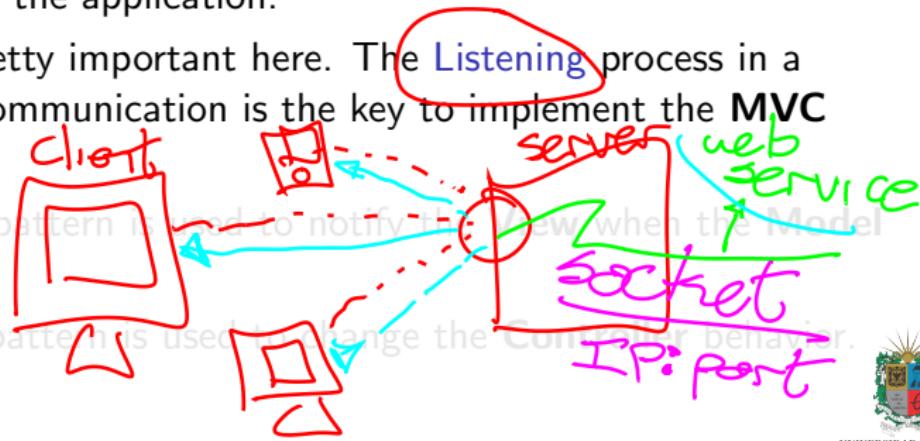
- Model is composed by **entity models**. It is the **data** and **logic** of the application.
- View is composed by **boundary objects**. It is the **presentation** of the application and interaction with external elements as **users**.
- Controller is composed by control objects. It is the **user interaction** and **control** of the application.

- Sockets are mandatory here. The Listening process in a bidirectional pattern.
 - The Observer pattern is used to notify the View when the Model changes.
 - The Strategy pattern is used to change the Controller behavior.
- 
- Controller
- mandatory
- connection API
- ~
- ~
- ~
- ~
- Fwrc (input-Frot)
- return output-bz)
- ↳ invoke backend



MVC Implementation

- **Model** is composed by **entity models**. It is the **data** and **logic** of the application.
 - **View** is composed by **boundary objects**. It is the **presentation** of the application and interaction with external elements as **users**.
 - **Controller** is composed by **control objects**. It is the **user interaction** and **control** of the application.
 - **Sockets** are pretty important here. The **Listening** process in a bidirectional communication is the key to implement the **MVC pattern**.



MVC Implementation

- **Model** is composed by **entity models**. It is the **data** and **logic** of the application.
- **View** is composed by **boundary objects**. It is the **presentation** of the application and interaction with external elements as **users**.
- **Controller** is composed by **control objects**. It is the **user interaction** and **control** of the application.
- **Sockets** are pretty important here. The **Listening** process in a bidirectional communication is the key to implement the **MVC pattern**.
- The **Observer** pattern is used to notify the **View** when the **Model** changes.
- The **Strategy** pattern is used to change the **Controller** behavior.



MVC Implementation

- Model is composed by **entity models**. It is the **data** and **logic** of the application.
- View is composed by **boundary objects**. It is the **presentation** of the application and interaction with external elements as **users**.
- Controller is composed by **control objects**. It is the **user interaction** and **control** of the application.
- Sockets are pretty important here. The **Listening** process in a bidirectional communication is the key to implement the **MVC pattern**.
- The **Observer** pattern is used to notify the **View** when the **Model** changes.
- The **Strategy** pattern is used to change the **Controller** behavior.



Outline

- 1 Model-View-Controller Pattern

~~SOLID~~

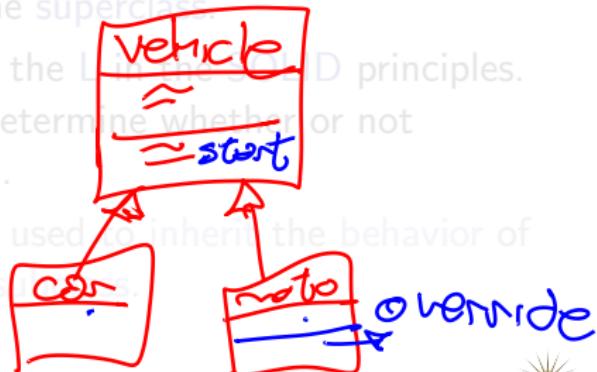
- 2 Design Principles underlying Design Patterns

- 3 Anti-Patterns & Code Smells



Liskov Substitution Principle

Functionality ≠ expected output



Liskov Substitution Principle

- **Liskov Substitution Principle** is a **design principle** that states that objects of a superclass should be replaceable with objects of its subclasses without affecting the functionality of the program.
- It means, the subclass should be able to extend the superclass without changing the behavior of the superclass.
- The Liskov Substitution Principle is the L in the SOLID principles. This principle uses substitution to determine whether or not inheritance has been **properly used**.
- The Liskov Substitution Principle is used to inherit the behavior of the superclass and extend it in the subclass.



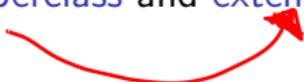
Liskov Substitution Principle

- **Liskov Substitution Principle** is a **design principle** that states that objects of a superclass should be replaceable with objects of its subclasses without affecting the functionality of the program.
- It means, the **subclass** should be able to **extend** the **superclass** without changing the **behavior** of the **superclass**.
- The **Liskov Substitution Principle** is the **L** in the **SOLID** principles. This principle uses **substitution** to determine whether or not **inheritance has been properly used.**
- The Liskov Substitution Principle is used to inherit the behavior of the superclass and extend it in the subclass.



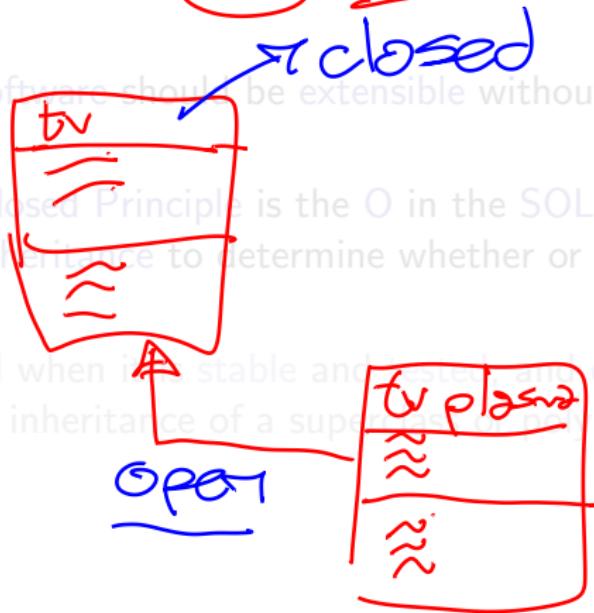
Liskov Substitution Principle

- **Liskov Substitution Principle** is a **design principle** that states that objects of a **superclass** should be replaceable with objects of its subclasses without affecting the functionality of the program.
- It means, the **subclass** should be able to **extend** the superclass without changing the **behavior** of the **superclass**.
- The **Liskov Substitution Principle** is the **L** in the **SOLID** principles. This principle uses **substitution** to determine whether or not inheritance has been **properly used**.
- The **Liskov Substitution Principle** is used to **inherit** the **behavior** of the **superclass** and **extend** it in the **subclass**.



Open — Closed Principle

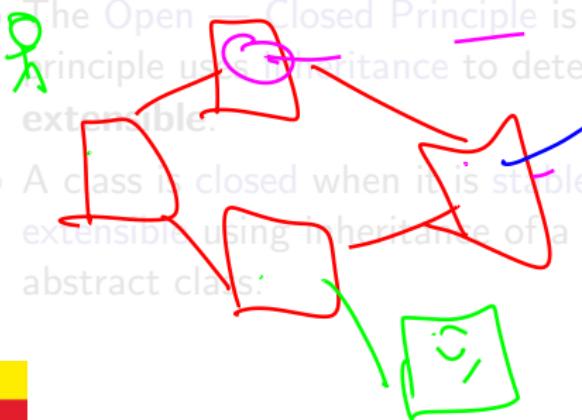
- **Open — Closed Principle** is a design principle that states that software entities should be open for extension but closed for modification.
- It means, the software should be extensible without changing the source code.
- The Open — Closed Principle is the O in the SOLID principles. This principle uses inheritance to determine whether or not the software is extensible.
- A class is closed when it is stable and open when it is extensible using inheritance of a super-class or polymorphism from an abstract class.



Open — Closed Principle

- **Open — Closed Principle** is a **design principle** that states that software entities should be open for extension but closed for modification.
- It means, the software should be extensible without changing the source code.

- The Open — Closed Principle is the O in the SOLID principles. This principle uses inheritance to determine whether or not the software is extensible.
- A class is closed when it is stable and tested, and open when it is extensible using inheritance of a superclass or polymorphism from an abstract class.



Open-Closed



Open — Closed Principle

- **Open — Closed Principle** is a **design principle** that states that software entities should be open for extension but closed for modification.
 - It means, the **software** should be **extensible** without changing the source code.
 - The **Open** — Closed Principle is the **O** in the **SOLID** principles. This principle uses **inheritance** to determine whether or not the **software** is **extensible**.
 - A class is closed when it is stable and tested, and open when it is extensible using inheritance of a superclass or polymorphism from an abstract class.
- ↳ maintainable
flexible*



Open — Closed Principle



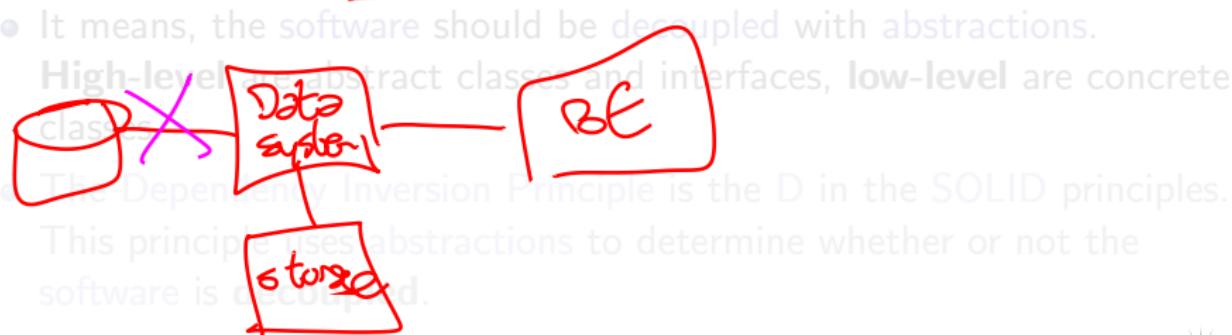
- **Open — Closed Principle** is a design principle that states that software entities should be open for extension but closed for modification.
- It means, the **software** should be **extensible** without changing the source code.
- The Open — Closed Principle is the **O** in the **SOLID** principles. This principle uses **inheritance** to determine whether or not the **software** is **extensible**.
- A class is closed when it is stable and tested, and open when it is extensible using inheritance of a superclass or polymorphism from an abstract class.



Dependency Inversion Principle

- A common problem is: To what degree is my system, or subsystems, dependent on a particular resource? - Coupling

- Dependency Inversion Principle is a design principle that states the high-level modules should not depend on low-level modules. Both should depend on abstractions.



Dependency Inversion Principle

- A common problem is: *To what degree is my system, or subsystems, dependent on a particular resource?* - **Coupling**
- **Dependency Inversion Principle** is a **design principle** that states that high-level modules should not depend on low-level modules. Both should depend on abstractions.
- It means, the software should be decoupled with abstractions. **High-level** are abstract classes and interfaces, **low-level** are concrete classes.
- The Dependency Inversion Principle is the D in the SOLID principles. This principle uses abstractions to determine whether or not the software is **decoupled**.



Dependency Inversion Principle

- A common problem is: *To what degree is my system, or subsystems, dependent on a particular resource?* - **Coupling**
- **Dependency Inversion Principle** is a **design principle** that states that high-level modules should not depend on low-level modules. Both should depend on abstractions.
- It means, the **software** should be decoupled with abstractions.
High-level are abstract classes and interfaces. **low-level** are concrete classes.
- The Dependency Inversion Principle is the D in the SOLID principles. This principle uses abstractions to determine whether or not the software is **decoupled**.



Dependency Inversion Principle

- A common problem is: *To what degree is my system, or subsystems, dependent on a particular resource?* - **Coupling**
- **Dependency Inversion Principle** is a **design principle** that states that high-level modules should not depend on low-level modules. Both should depend on abstractions.
- It means, the **software** should be **decoupled** with **abstractions**.
High-level are abstract classes and interfaces, **low-level** are concrete classes.
- The **Dependency Inversion Principle** is the **D** in the **SOLID** principles. This principle uses abstractions to determine whether or not the software is **decoupled**.

*+ generalizations
— details*



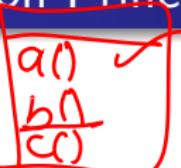
Composing Objects Principle

- **Composing Objects Principle** is a design principle that states that software entities should be composed of objects.
- It means, the software should be composed of objects to modularize the software.
- The Composing Objects Principle is used to reduce coupling and increase cohesion in the software.
- This principle states that classes should achieve code reuse through composition or aggregation rather than inheritance.
- Design patterns like Composite and Decorator are used to implement the Composing Objects Principle.
- The disadvantage of this principle is that it can increase the number of objects in the software. It reduces options of share code.

+ memory

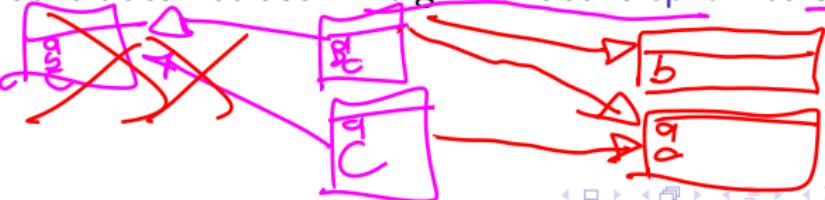


Interface Segregation Principle



buses
Forced

- **Interface Segregation Principle** is a design principle that states that a client should not be forced to implement an interface that it does not use.
- It means, the software should be composed of interfaces to modularize the software.
- The Interface Segregation Principle is the I in the SOLID principles. This principle uses interfaces to determine whether or not the software is modularized.
- This principle states that a class should not be forced to implement interfaces that it does not use. A big interface is split into smaller interfaces.



Principle of Least Knowledge

- **Principle of Least Knowledge** is a design principle that states that a software entity should not have knowledge of unnecessary details.
- The Principle of Least Knowledge is used to modularize the software with objects. *Facade*
- The Law of Demeter is a specific case of the Principle of Least Knowledge. It states that a software entity should only have knowledge of its immediate friends. *object*
- Classes should only have knowledge of their attributes and methods. They should not have knowledge of the attributes and methods of other classes.



Outline

1 Model-View-Controller Pattern

2 Design Principles underlying Design Patterns

3 Anti-Patterns & Code Smells



Bad Coding

~~COV~~DRY

↳ Bad Coding - 96 cartones ~ Tot \Rightarrow 7 Tb/week



} Class - methods
functions / modules

- **Bad Coding** is a software design problem that states that the code is not well written.
- If the software has bad coding, it is not maintainable and extensible.
- Spaghetti Code is a bad coding that is difficult to understand and maintain.
- **Bad practices** as copy-paste code, hardcoded values, and magic numbers are bad coding.

Converts
functions

Use
args
return

\rightarrow Algorithm

↳ Duplicate Code

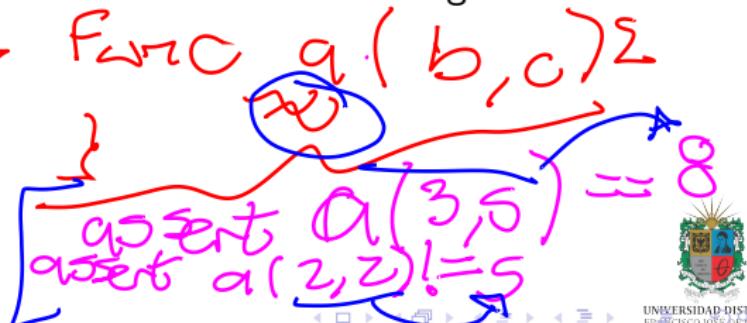
\rightarrow IP = "127.0.0.1"



Code Quality



- **Code Quality** is a process to validate that the code is well written.
- Metrics as code coverage, cyclomatic complexity, and code smells are used to measure the code quality.
- Code Review is a process to validate that the code is well written by another developer.
- Unit Testing is a process to validate that a small fragment of code is working as expected



Anti-Patterns



- **AntiPatterns** are bad practices in software design.
- An AntiPattern is a pattern that is commonly used but is ineffective and counterproductive.
- AntiPatterns are used to identify and fix bad practices in software design.
- Techniques to avoid AntiPatterns are refactoring, code review, and unit testing.

Change code
to change behavior

`assert time(2ss, 10s) < 0.5 (seg)`

`assert ~raiseError`

`assert time(2ss, 10s) < 0.5 (seg)`



Identify and Fix Code Smells

Shotgun Surgery

- Identify Code Smells is a process to find the bad coding in the software.
- Fix Code Smells is a process to correct the bad coding in the software.
- To identify and fix code smells, the software should be refactored.
- Refactoring is a process to improve the software without changing the behavior.

Unit test

- Techniques like code review and unit testing are used to identify and fix code smells.
- Linters and static analysis tools are used to identify and fix code smells.

Long methods

class

GodClass → Black Hole

DAO

Factory
get & set

abstraction
composite



Outline

• ~~Primitive Obsession~~

• Long Parameter List → ≤ 5

↳ Encapsulate in an Object

Model-View-Controller Pattern

**kwargs ↳ key-value

Facade

• Feature envy

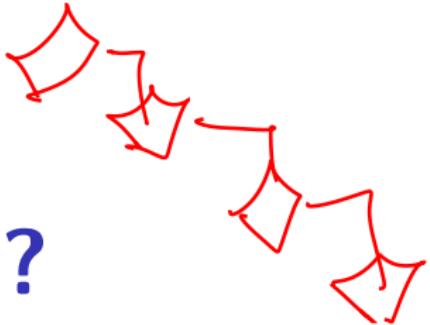


3 Anti-Patterns & Code Smells

- Improper abc Introspect
- Message Chain
- Atmref()



- Switch statmmt **Thanks!**



elif

Questions?



Repo: <https://github.com/EngAndres/ud-public/tree/main/courses/software-modeling>

