AntiPatterns & Code Smells

Software Modeling

Author: Eng. Carlos Andrés Sierra, M.Sc.
cavirguezs@udistrital.edu.co

Computer Engineer
Lecturer
Universidad Distrital Francisco José de Caldas

2024-III

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Outline

1. Model-View-Controller Pattern

2. Design Principles underlying Design Patterns

3. Anti-Patterns & Code Smells

Outline

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

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 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 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 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 Front Controller, Application Controller, and Request Dispatcher. It means, all the connection with **back-end**.

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.

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.

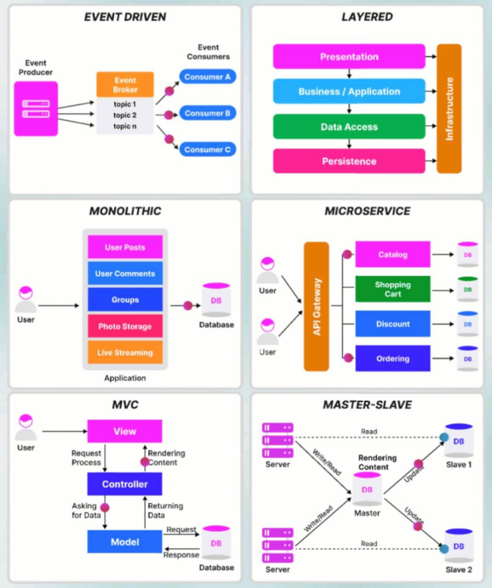
MVC Implementation

- 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.

Web Development Patterns



Outline

1. Model-View-Controller Pattern

2. Design Principles underlying Design Patterns

3. Anti-Patterns & Code Smells

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.

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 tested, and open when it is extensible using inheritance of a superclass 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 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 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 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**.

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**.

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 redice 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 disavantage of this principle is that it can increase the number of objects in the software. It reduces options of share code.

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 redice 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 disavantage of this principle is that it can increase the number of objects in the software. It reduces options of share code.

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 redice 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 disavantage of this principle is that it can increase the number of objects in the software. It reduces options of share code.

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 redice 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 disavantage of this principle is that it can increase the number of objects in the software. It reduces options of share code.

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 redice 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 disavantage of this principle is that it can increase the number of objects in the software. It reduces options of share code.

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 redice 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 disavantage of this principle is that it can increase the number of objects in the software. It reduces options of share code.

Interface Segregation Principle

- **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.

Interface Segregation Principle

- **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.

Interface Segregation Principle

- **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.

Interface Segregation Principle

- **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.
- **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.
- 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

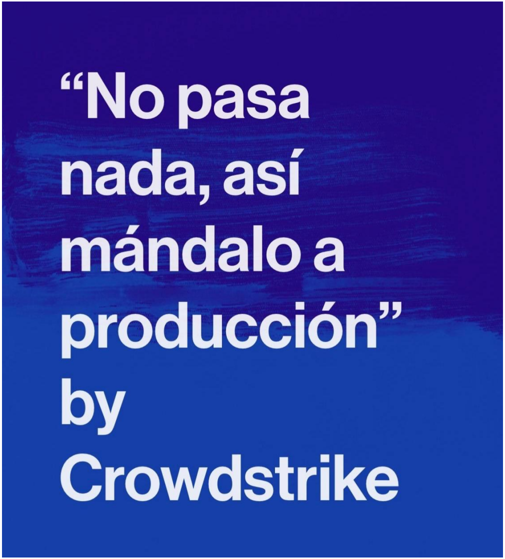
Bad Coding

- **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.

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

Stupid Deployments!



“No pasa nada, así mándalo a producción” by Crowdstrike

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.

Identify and Fix Code Smells

- 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. A good book is *Refactoring: Improving the Design of Existing Code*, by Martin Flower.
- 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.

Examples of Code Smells I

- **Comments** are used to explain the code. It could be a code smell because the code maybe is not self-explanatory. Should have a equilibrium of comments.
- **Long Methods** and **Long Classes** (`Good Classes` or `Black-Hole Classes`) are used to group the code. It could be a code smell because the method or the class maybe is doing too much. Remember: Single Responsability and Separation of Concerns.
- **Magic Numbers** are used to `hardcode` values. It could be a code smell because the value maybe is not modularized. Use constants instead.
- **Duplicated Code** is used to reuse the code, maybe in blocks of code that are similar. It could be a code smell because the code maybe is not modularized. DRY (*don't repeat yourself*) principle.

Examples of Code Smells I

- **Comments** are used to explain the code. It could be a code smell because the code maybe is not self-explanatory. Should have a equilibrium of comments.
- **Long Methods** and **Long Classes** (`Good Classes` or `Black-Hole Classes`) are used to group the code. It could be a code smell because the method or the class maybe is doing too much. Remember: Single Responsability and Separation of Concerns.
- **Magic Numbers** are used to `hardcode` values. It could be a code smell because the value maybe is not modularized. Use constants instead.
- **Duplicated Code** is used to reuse the code, maybe in blocks of code that are similar. It could be a code smell because the code maybe is not modularized. DRY (*don't repeat yourself*) principle.

Examples of Code Smells I

- **Comments** are used to explain the code. It could be a code smell because the code maybe is not self-explanatory. Should have a equilibrium of comments.
- **Long Methods** and **Long Classes** (`Good Classes` or `Black-Hole Classes`) are used to group the code. It could be a code smell because the method or the class maybe is doing too much. Remember: Single Responsability and Separation of Concerns.
- **Magic Numbers** are used to `hardcode` values. It could be a code smell because the value maybe is not modularized. Use constants instead.
- **Duplicated Code** is used to reuse the code, maybe in blocks of code that are similar. It could be a code smell because the code maybe is not modularized. DRY (*don't repeat yourself*) principle.

Examples of Code Smells I

- **Comments** are used to explain the code. It could be a code smell because the code maybe is not self-explanatory. Should have a equilibrium of comments.
- **Long Methods** and **Long Classes** (`Good Classes` or `Black-Hole Classes`) are used to group the code. It could be a code smell because the method or the class maybe is doing too much. Remember: Single Responsability and Separation of Concerns.
- **Magic Numbers** are used to `hardcode` values. It could be a code smell because the value maybe is not modularized. Use constants instead.
- **Duplicated Code** is used to reuse the code, maybe in blocks of code that are similar. It could be a code smell because the code maybe is not modularized. DRY (*don't repeat yourself*) principle.

Examples of Code Smells II

- **Dead Code** is used to keep the code that is not used. It could be a code smell because the code maybe is not maintainable. Remove the code that is not used.
- **Data Classes** are used to group the data. It could be a code smell because the class contains only data and not real functionality. Use encapsulation instead, and not just getters & setters.
- **Feature Envy** consist in a method that uses more the data of another class than its own data. It could be a code smell because it increases the coupling between the classes. Use encapsulation instead, or a design pattern like Observer.
- **Data Clumps** consist in a group of data that is used together. It could be a code smell because the data maybe is not modularized. Use encapsulation instead, or a design pattern like Composite.

Examples of Code Smells II

- **Dead Code** is used to keep the code that is not used. It could be a code smell because the code maybe is not maintainable. Remove the code that is not used.
- **Data Classes** are used to group the data. It could be a code smell because the class contains only data and not real functionality. Use encapsulation instead, and not just getters & setters.
- **Feature Envy** consist in a method that uses more the data of another class than its own data. It could be a code smell because it increases the coupling between the classes. Use encapsulation instead, or a design pattern like Observer.
- **Data Clumps** consist in a group of data that is used together. It could be a code smell because the data maybe is not modularized. Use encapsulation instead, or a design pattern like Composite.

Examples of Code Smells II

- **Dead Code** is used to keep the code that is not used. It could be a code smell because the code maybe is not maintainable. Remove the code that is not used.
- **Data Classes** are used to group the data. It could be a code smell because the class contains only data and not real functionality. Use encapsulation instead, and not just getters & setters.
- **Feature Envy** consist in a method that uses more the data of another class than its own data. It could be a code smell because it increases the coupling between the classes. Use encapsulation instead, or a design pattern like Observer.
- **Data Clumps** consist in a group of data that is used together. It could be a code smell because the data maybe is not modularized. Use encapsulation instead, or a design pattern like Composite.

Examples of Code Smells II

- **Dead Code** is used to keep the code that is not used. It could be a code smell because the code maybe is not maintainable. Remove the code that is not used.
- **Data Classes** are used to group the data. It could be a code smell because the class contains only data and not real functionality. Use encapsulation instead, and not just getters & setters.
- **Feature Envy** consist in a method that uses more the data of another class than its own data. It could be a code smell because it increases the coupling between the classes. Use encapsulation instead, or a design pattern like Observer.
- **Data Clumps** consist in a group of data that is used together. It could be a code smell because the data maybe is not modularized. Use encapsulation instead, or a design pattern like Composite.

Examples of Code Smells III

- **Refused Bequest** occurs when a class inherits from another class but does not use the inherited methods. It could be a code smell because the class maybe is not modularized. Use composition instead, or a design pattern like Template.
- **Switch Statements** occurs when a class has a lot of `switch` statements. It could be a code smell because the class maybe is not modularized. Use polymorphism instead, or a design pattern like Strategy.
- **Long Parameter List** consists in a method that has a lot of parameters. It could be a code smell because the method maybe is doing too much or is hard to call. Use parameter objects instead.
- **Divergent Change** occurs when a class is changed for different reasons. It could be a code smell because the class maybe is not modularized. Use composition instead, or a design pattern like Strategy.

Examples of Code Smells III

- **Refused Bequest** occurs when a class inherits from another class but does not use the inherited methods. It could be a code smell because the class maybe is not modularized. Use composition instead, or a design pattern like Template.
- **Switch Statements** occurs when a class has a lot of `switch` statements. It could be a code smell because the class maybe is not modularized. Use polymorphism instead, or a design pattern like Strategy.
- **Long Parameter List** consists in a method that has a lot of parameters. It could be a code smell because the method maybe is doing too much or is hard to call. Use parameter objects instead.
- **Divergent Change** occurs when a class is changed for different reasons. It could be a code smell because the class maybe is not modularized. Use composition instead, or a design pattern like Strategy.

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Examples of Code Smells III

- **Refused Bequest** occurs when a class inherits from another class but does not use the inherited methods. It could be a code smell because the class maybe is not modularized. Use composition instead, or a design pattern like Template.
- **Switch Statements** occurs when a class has a lot of `switch` statements. It could be a code smell because the class maybe is not modularized. Use polymorphism instead, or a design pattern like Strategy.
- **Long Parameter List** consists in a method that has a lot of parameters. It could be a code smell because the method maybe is doing too much or is hard to call. Use parameter objects instead.
- **Divergent Change** occurs when a class is changed for different reasons. It could be a code smell because the class maybe is not modularized. Use composition instead, or a design pattern like Strategy.

Examples of Code Smells III

- **Refused Bequest** occurs when a class inherits from another class but does not use the inherited methods. It could be a code smell because the class maybe is not modularized. Use composition instead, or a design pattern like Template.
- **Switch Statements** occurs when a class has a lot of `switch` statements. It could be a code smell because the class maybe is not modularized. Use polymorphism instead, or a design pattern like Strategy.
- **Long Parameter List** consists in a method that has a lot of parameters. It could be a code smell because the method maybe is doing too much or is hard to call. Use parameter objects instead.
- **Divergent Change** occurs when a class is changed for different reasons. It could be a code smell because the class maybe is not modularized. Use composition instead, or a design pattern like Strategy.

UNIVERSIDAD DISTRITAL FRANCISCO JOSÉ DE CALDAS

Examples of Code Smells IV

- **Shotgun Surgery** is a common problem in software design. It occurs when a change in a class requires changes in many other classes. It could be a code smell because the class maybe is not modularized. Use composition instead, or a structural design pattern.
- **Innapropiate Intimacy** occurs when a class has a lot of dependencies with other classes. It could be a code smell because the class maybe is not modularized. Use composition instead, or a design pattern as proxy. Remember the Principle of Least Knowledge.
- **Message Chains** violates the Law of Demeter. It occurs when a class calls a method of another class that calls a method of another class, and so on. It could be a code smell because the class maybe is not modularized. Use encapsulation instead, or a design pattern like Observer.

Examples of Code Smells IV

- **Shotgun Surgery** is a common problem in software design. It occurs when a change in a class requires changes in many other classes. It could be a code smell because the class maybe is not modularized. Use composition instead, or a structural design pattern.
- **Innapropiate Intimacy** occurs when a class has a lot of dependencies with other classes. It could be a code smell because the class maybe is not modularized. Use composition instead, or a design pattern as proxy. Remember the Principle of Least Knowledge.
- **Message Chains** violates the Law of Demeter. It occurs when a class calls a method of another class that calls a method of another class, and so on. It could be a code smell because the class maybe is not modularized. Use encapsulation instead, or a design pattern like Observer.

Examples of Code Smells IV

- **Shotgun Surgery** is a common problem in software design. It occurs when a change in a class requires changes in many other classes. It could be a code smell because the class maybe is not modularized. Use composition instead, or a structural design pattern.
- **Innapropiate Intimacy** occurs when a class has a lot of dependencies with other classes. It could be a code smell because the class maybe is not modularized. Use composition instead, or a design pattern as proxy. Remember the Principle of Least Knowledge.
- **Message Chains** violates the Law of Demeter. It occurs when a class calls a method of another class that calls a method of another class, and so on. It could be a code smell because the class maybe is not modularized. Use encapsulation instead, or a design pattern like Observer.

Examples of Code Smells V

- **Primitive Obsession** consists in the use of primitive types instead of objects. It could be a code smell because the code maybe is not using right obstractions. Use abstract types instead.
- **Speculative Generality** consists in the use of design patterns that are not needed, or to create interfaces thinking maybe those could be useful in the future. It could be a code smell because the code maybe is not modularized. Use design patterns only when needed.

Examples of Code Smells V

- **Primitive Obsession** consists in the use of primitive types instead of objects. It could be a code smell because the code maybe is not using right obstractions. Use abstract types instead.
- **Speculative Generality** consists in the use of design patterns that are not needed, or to create interfaces thinking maybe those could be useful in the future. It could be a code smell because the code maybe is not modularized. Use design patterns only when needed.

Outline

1 Model-View-Controller Pattern

2 Design Principles underlying Design Patterns

3 Anti-Patterns & Code Smells

Thanks!

Questions?

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