Object-Oriented Modelling

Object-Oriented Programming

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

Lecturer Professor
Department of Computer Engineering
School of Engineering
Universidad Distrital Francisco José de Caldas

2025-I

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Outline

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Outline

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Design Before Code

- **Design** should come before coding.
- Jumping into *code without a plan* leads to confusion and rework.
- **Good design** clarifies the `problem` and guides the `solution`.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Design Before Code

- **Design** should come before coding.
- Jumping into *code without a plan* leads to confusion and rework.
- **Good design** clarifies the `problem` and guides the `solution`.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Understanding the Requirements

- **Requirements** must be well understood before design.
- Ask questions, clarify ambiguities, and document all requirements.
- Requirements define the scope and direction of the design.

Understanding the Requirements

- **Requirements** must be well understood before design.
- `Ask questions, clarify ambiguities, and document all requirements.`
- Requirements define the scope and direction of the `design`.

Design Based on the Problem

- **Design** should be driven by the problem, not by `technology`.
- Focus on what needs to be solved, *not just* how to implement it.
- Use the **problem statement** to identify key objects and their *relationships*.

Design Based on the Problem

- **Design** should be driven by the problem, not by `technology`.
- Focus on what needs to be solved, *not just* how to implement it.
- Use the **problem statement** to identify key objects and their *relationships*.

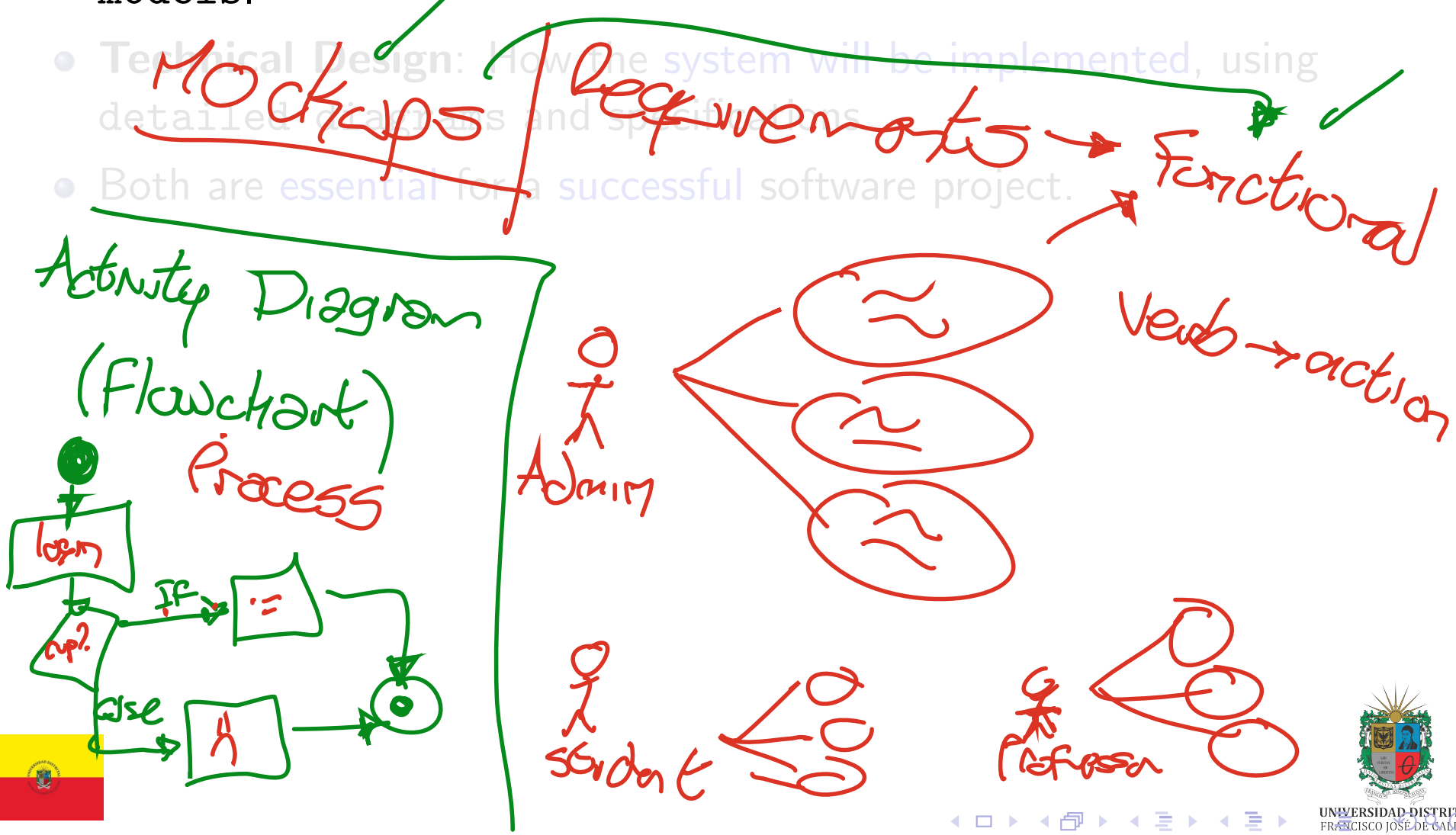
UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Object-Oriented Approach

- The **object-oriented approach** models the system as a collection of interacting objects.
- Each `object represents` a real-world entity or concept.
- **Objects** encapsulate data and behavior.

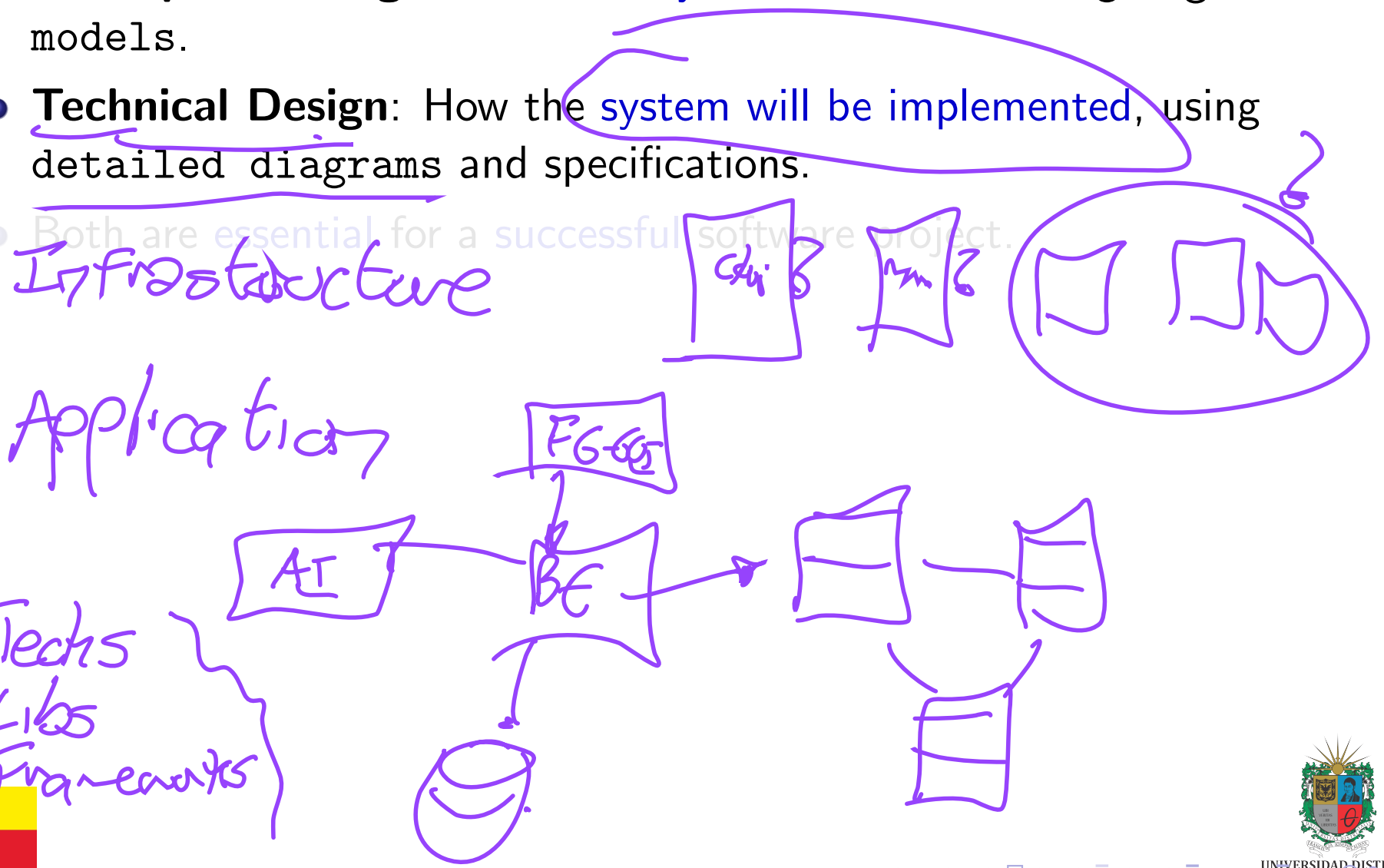
Conceptual Design and Technical Design

- **Conceptual Design**: What the system should do, using `high-level models`.
- **Technical Design**: How the system will be implemented, using detailed diagrams and specifications.
- Both are essential for a successful software project.



Conceptual Design and Technical Design

- **Conceptual Design**: What the system should do, using `high-level models`.
- **Technical Design**: How the system will be implemented, using `detailed diagrams` and specifications.
- Both are essential for a successful software project.



Conceptual Design and Technical Design

- **Conceptual Design**: What the system should do, using `high-level models`.
- **Technical Design**: How the system will be implemented, using `detailed diagrams` and specifications.
- Both are essential for a successful software project.

Conceptual → High Comprehension

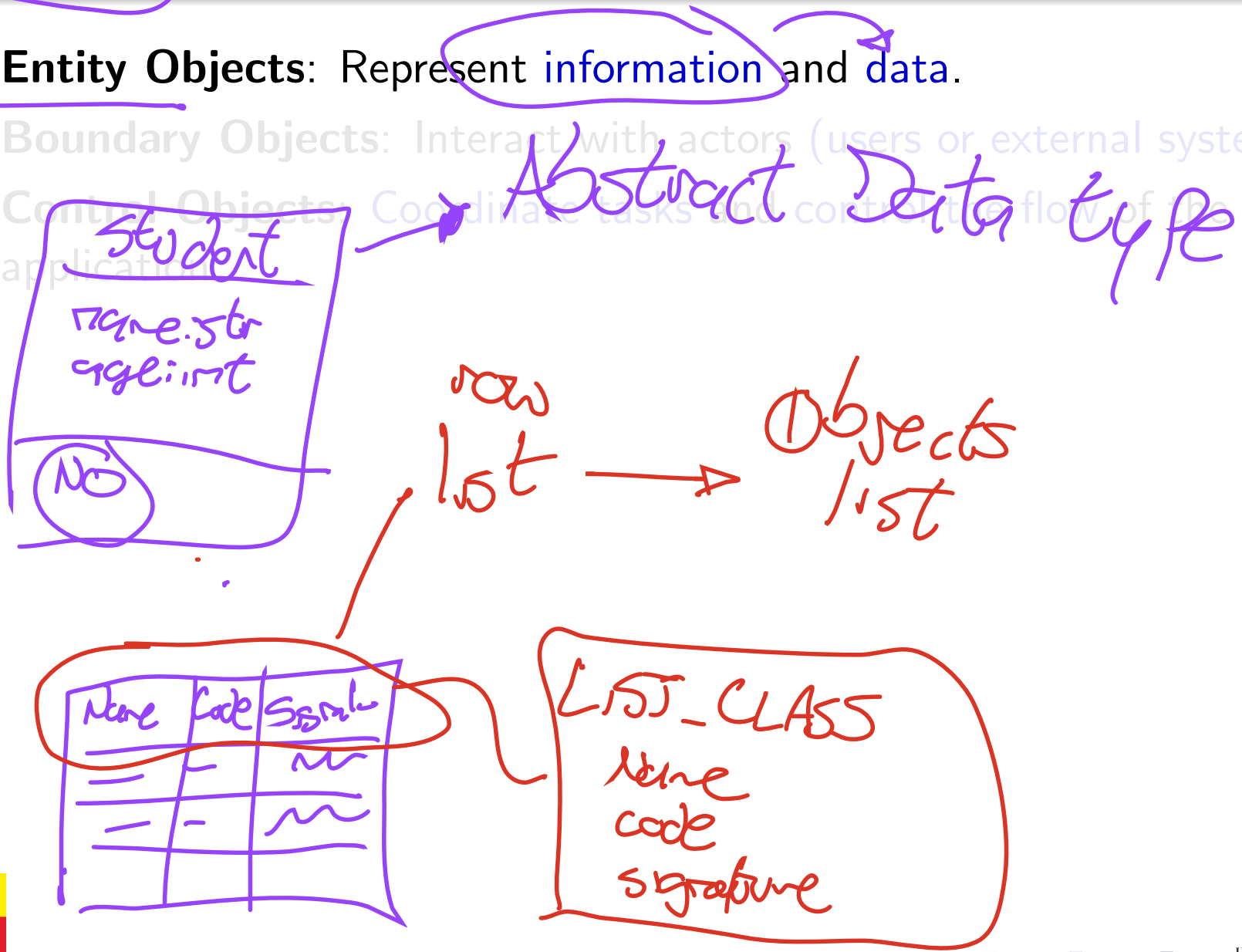
Technical → Make decisions

↳ code

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

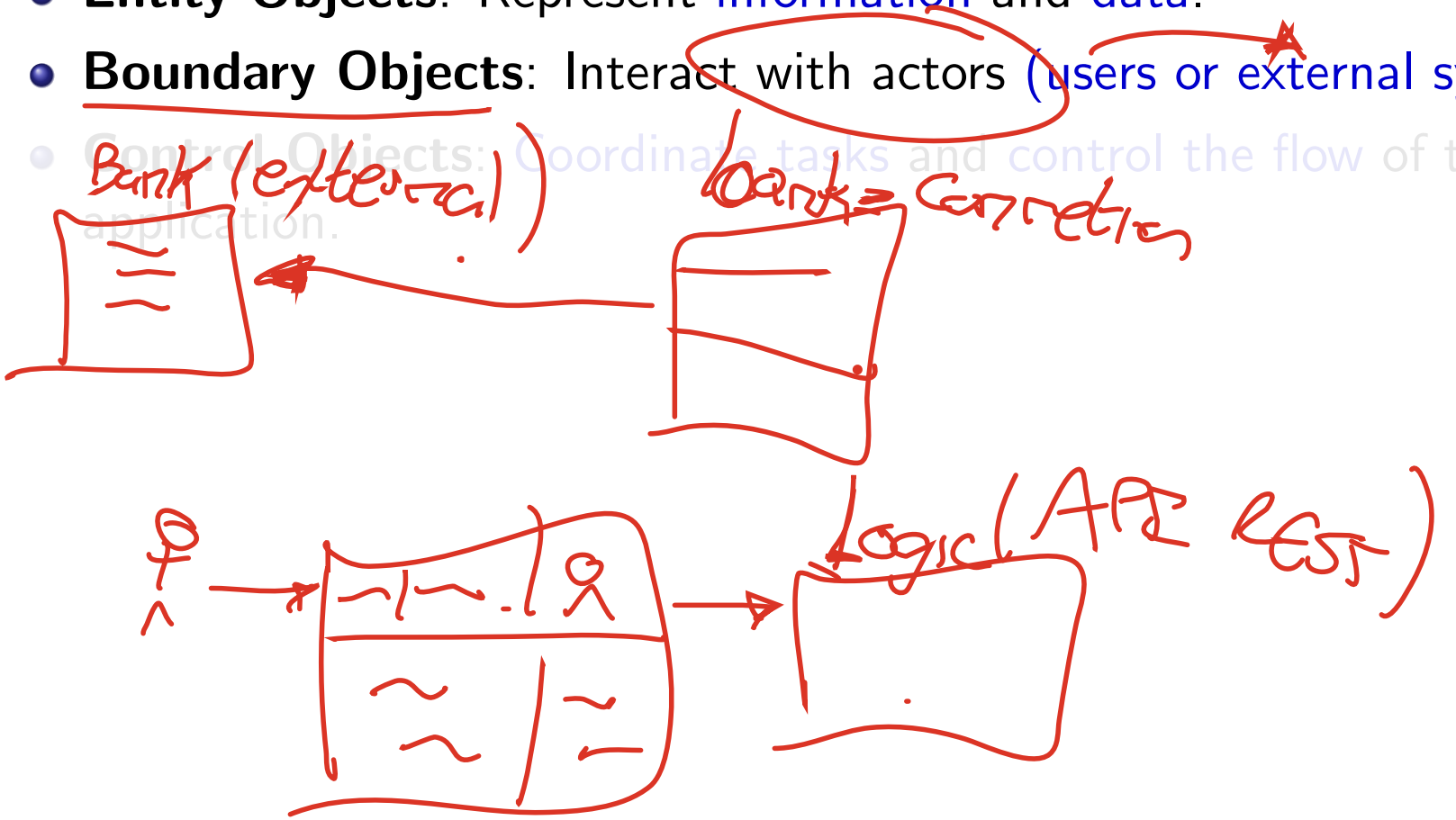
Categories of Objects

- **Entity Objects**: Represent information and data.
- **Boundary Objects**: Interact with actors (users or external systems).
- **Control Objects**: Coordinate tasks and control the flow of the application.



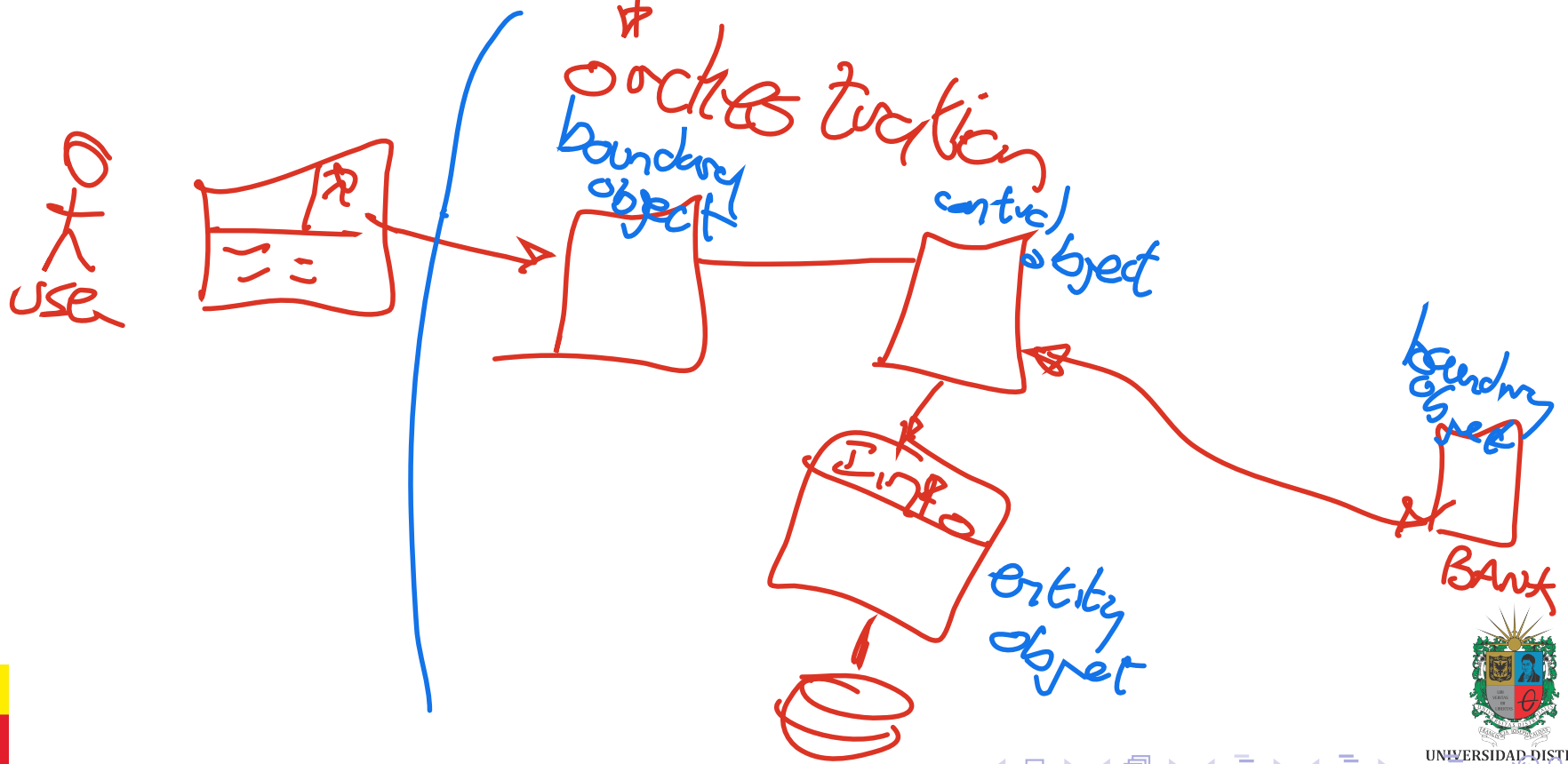
Categories of Objects

- **Entity Objects**: Represent information and data.
- **Boundary Objects**: Interact with actors (users or external systems).
- **Control Objects**: Coordinate tasks and control the flow of the application.



Categories of Objects

- **Entity Objects**: Represent information and data.
- **Boundary Objects**: Interact with actors (users or external systems).
- **Control Objects**: Coordinate tasks and control the flow of the application.



Documentation in Software

- **Documentation** is essential for communication and maintenance.
- Includes requirements, design diagrams, user manuals, and code comments.
- Good **documentation** helps new team members *understand* the *system* quickly.

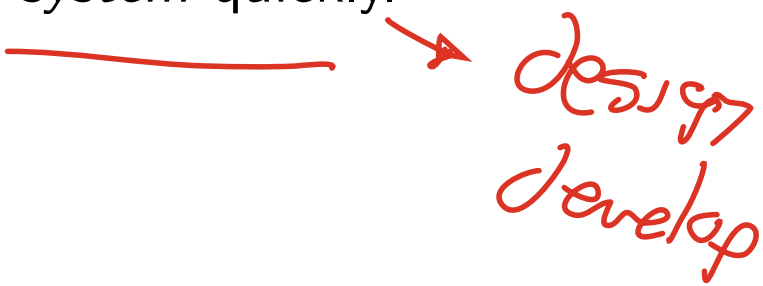
Documentation in Software

- **Documentation** is essential for communication and maintenance.
- Includes requirements, design diagrams, user manuals, and code comments.
- Good **documentation** helps new team members *understand* the *system* quickly.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Documentation in Software

- **Documentation** is essential for communication and maintenance.
- Includes requirements, design diagrams, user manuals, and code comments.
- Good **documentation** helps new team members *understand* the *system* quickly.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Outline

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

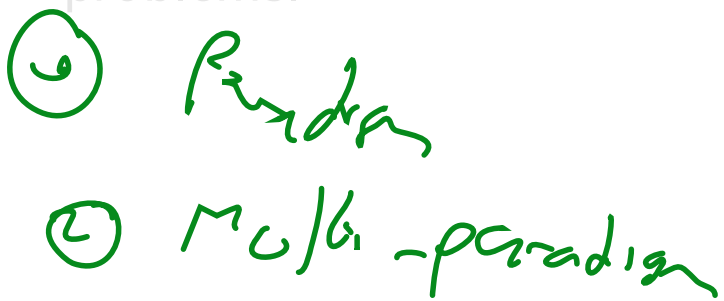
Talk with Machines: Programming Paradigms

- **Programming languages** are tools to communicate with machines.
- **Paradigms**: `Imperative`, `Procedural`, `Object-Oriented`, `Functional`, `Logic`.
- Each paradigm offers a *different way to think* about and solve problems.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

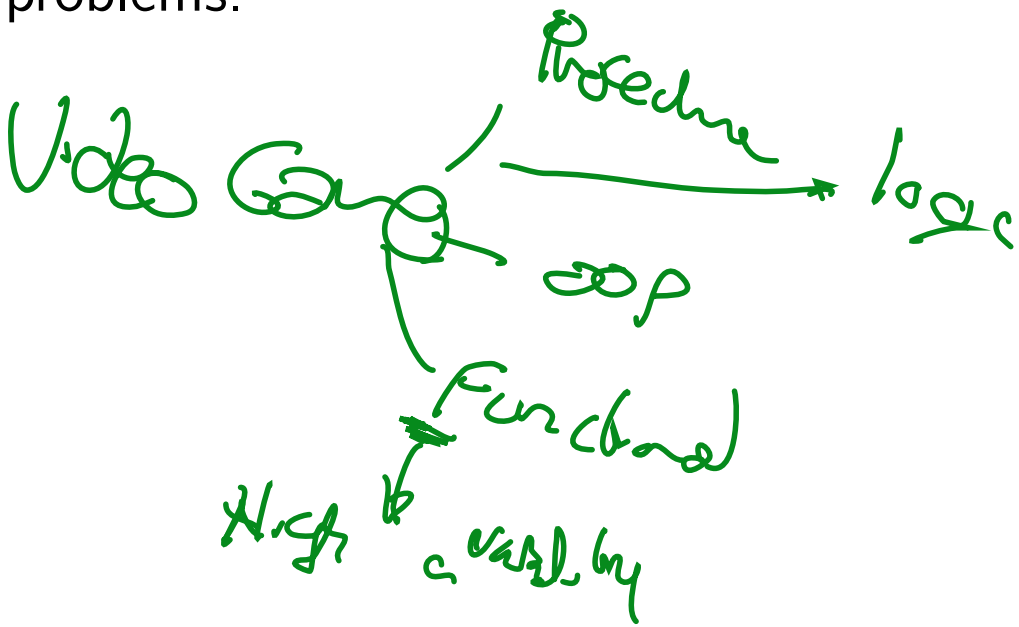
Talk with Machines: Programming Paradigms

- **Programming languages** are tools to communicate with machines.
- **Paradigms**: `Imperative`, `Procedural`, `Object-Oriented`, `Functional`, `Logic`.
- Each paradigm offers a *different way to think* about and solve problems.

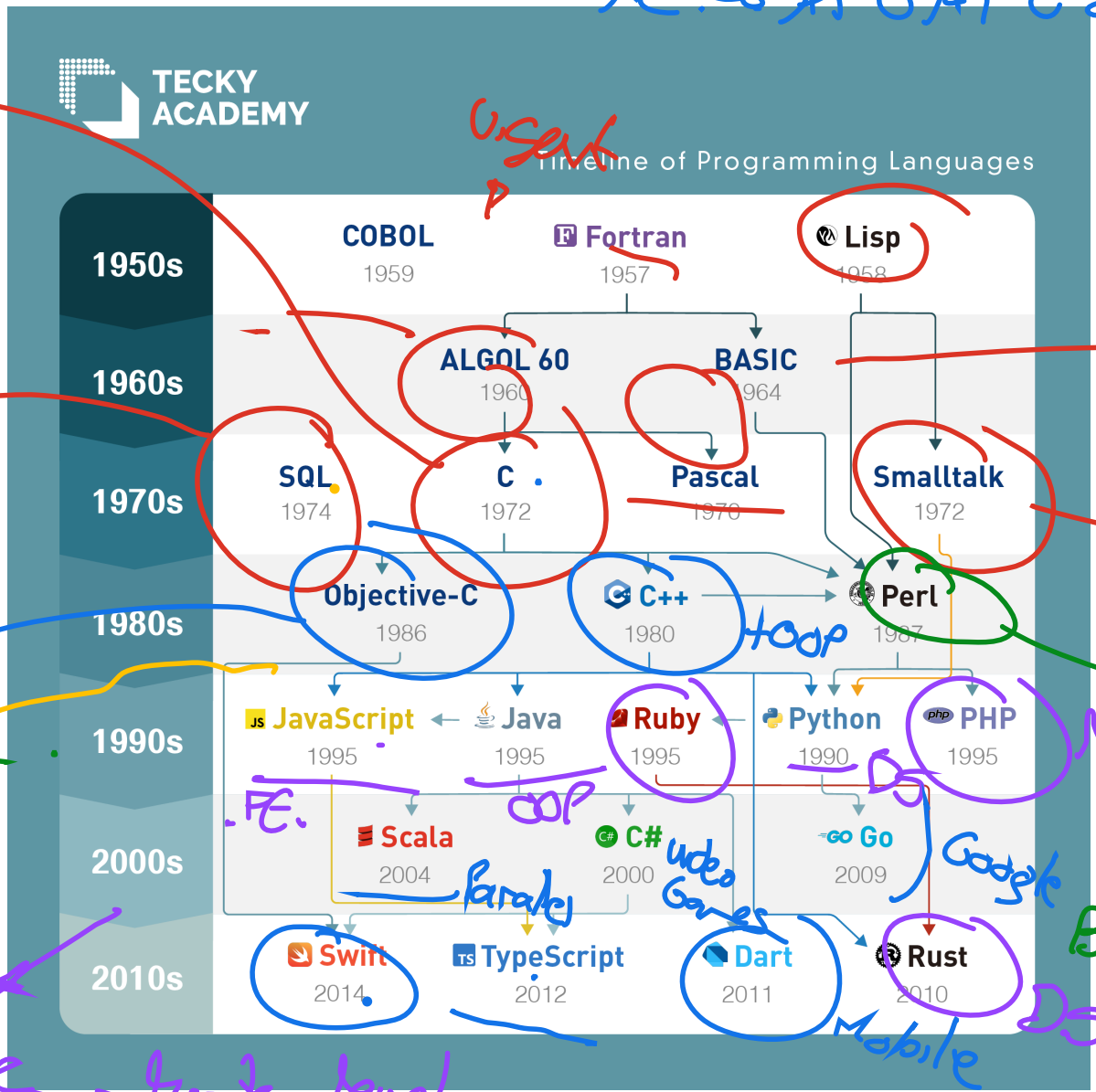


Talk with Machines: Programming Paradigms

- **Programming languages** are tools to communicate with machines.
- **Paradigms**: `Imperative`, `Procedural`, `Object-Oriented`, `Functional`, `Logic`.
- Each paradigm offers a *different way to think* about and solve problems.

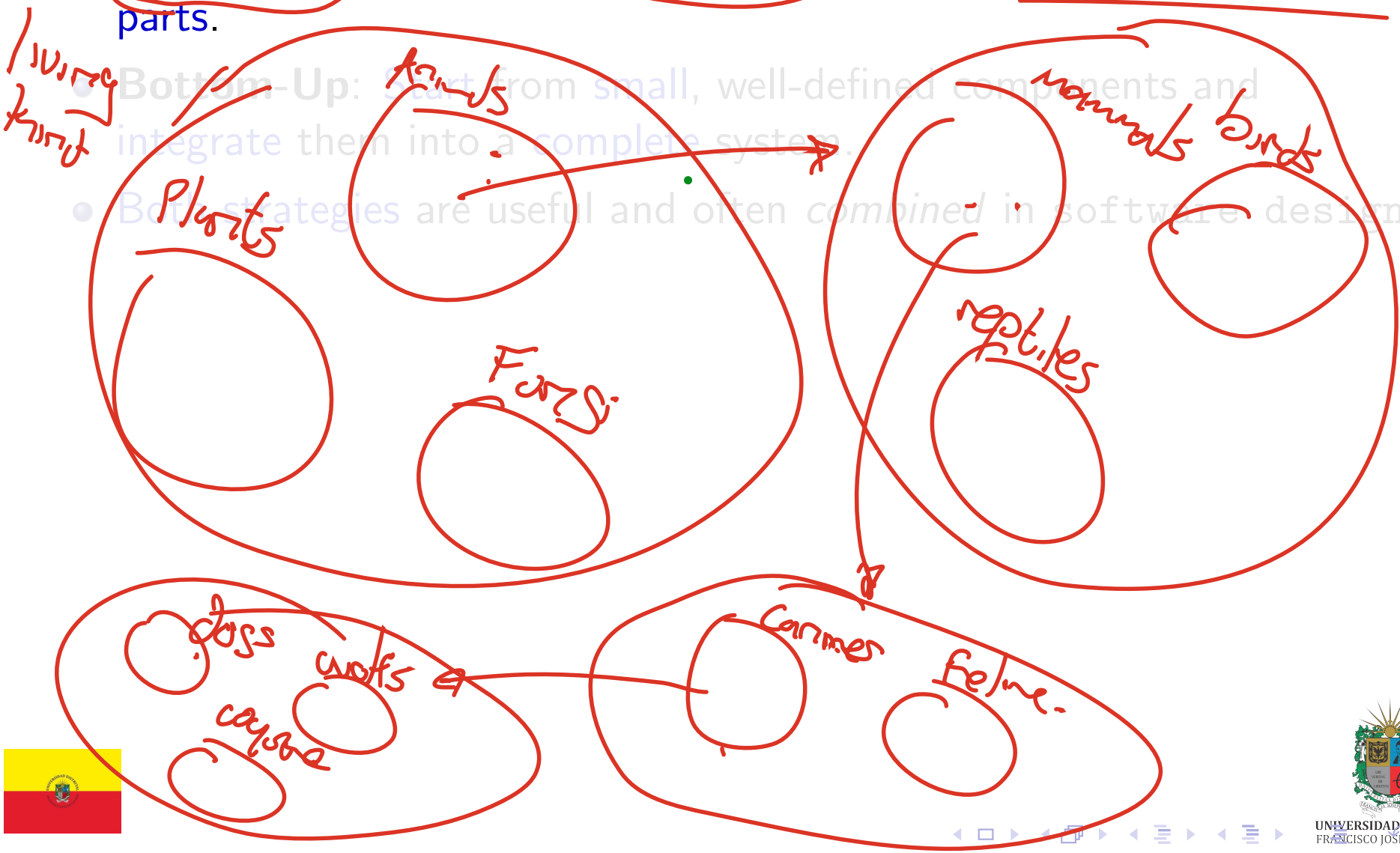


History of Programming Languages



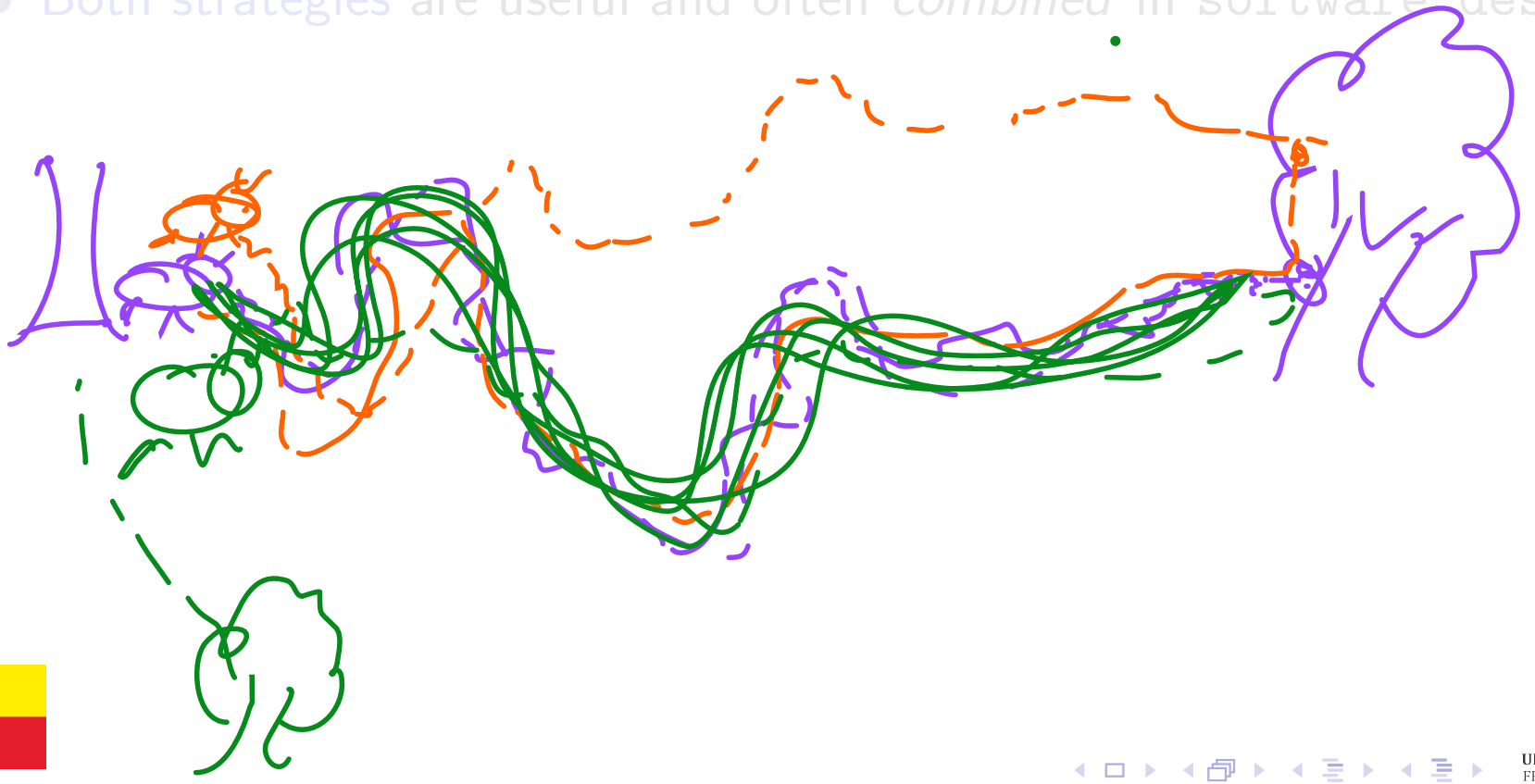
Strategies to Solve Problems

- **Top-Down**: Start from the big picture and break it down into smaller parts.
- **Bottom-Up**: Start from small, well-defined components and integrate them into a complete system.
- Both strategies are useful and often *combined* in software design.



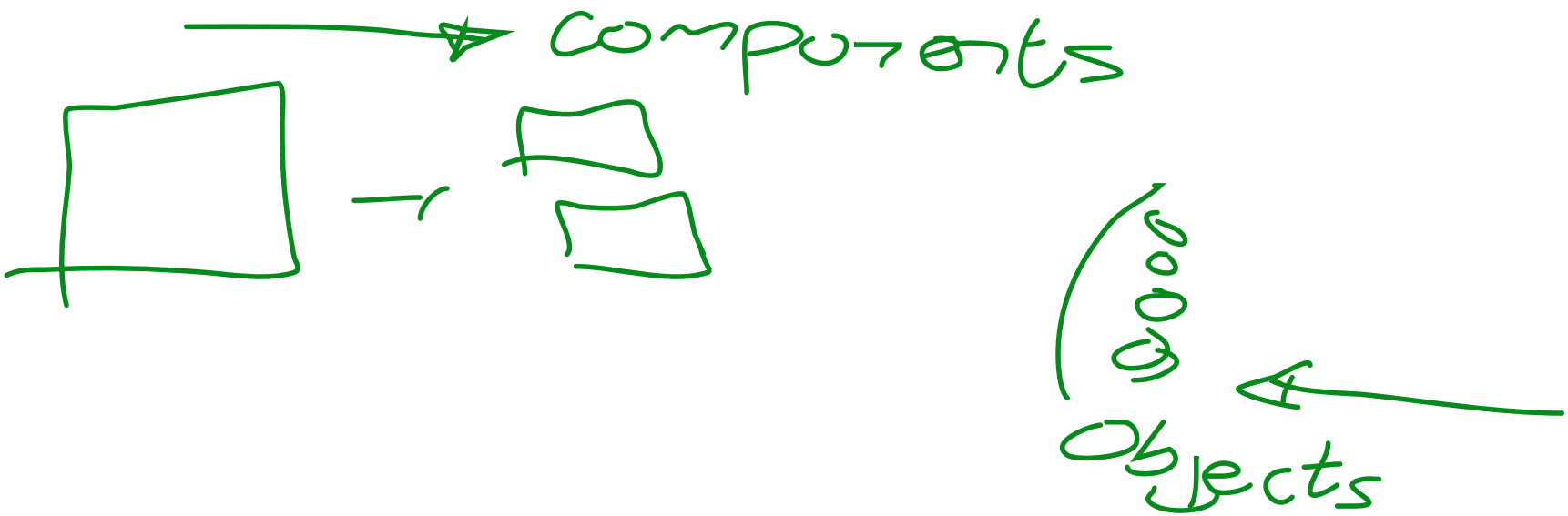
Strategies to Solve Problems

- **Top-Down**: Start from the big picture and break it down into smaller parts.
- **Bottom-Up**: Start from small, well-defined components and integrate them into a complete system.
- Both strategies are useful and often *combined* in `software design`.



Strategies to Solve Problems

- **Top-Down**: Start from the big picture and break it down into smaller parts.
- **Bottom-Up**: Start from small, well-defined components and integrate them into a complete system.
- Both strategies are useful and often *combined* in `software design`.



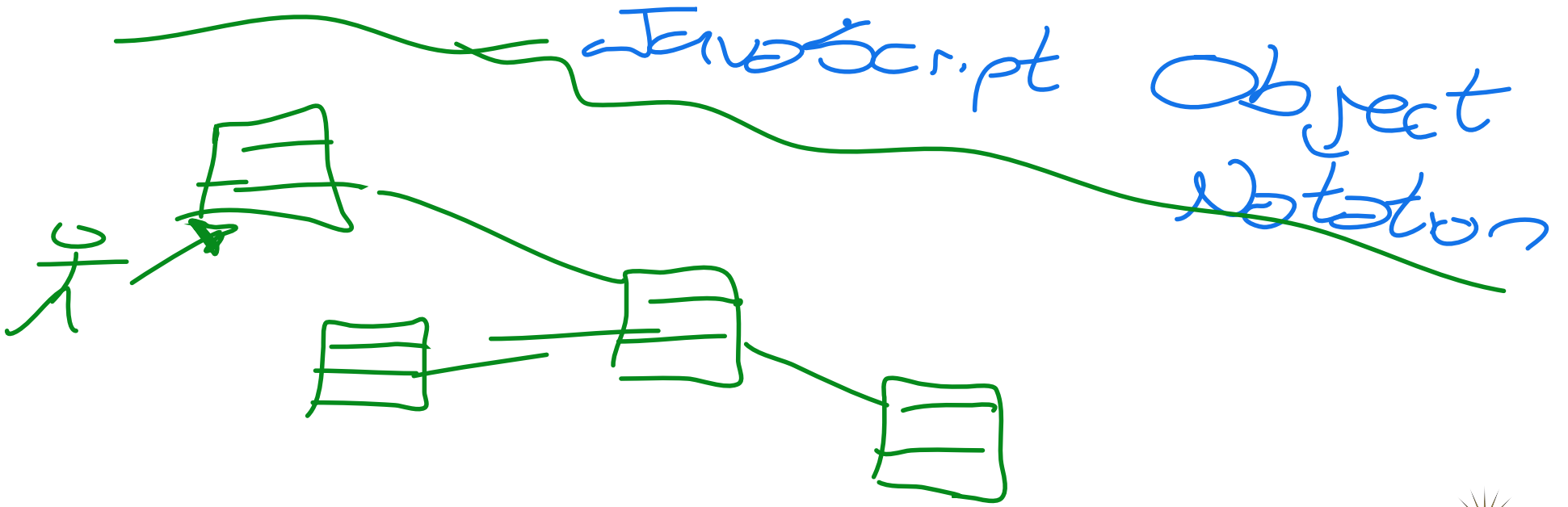
UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Outline

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

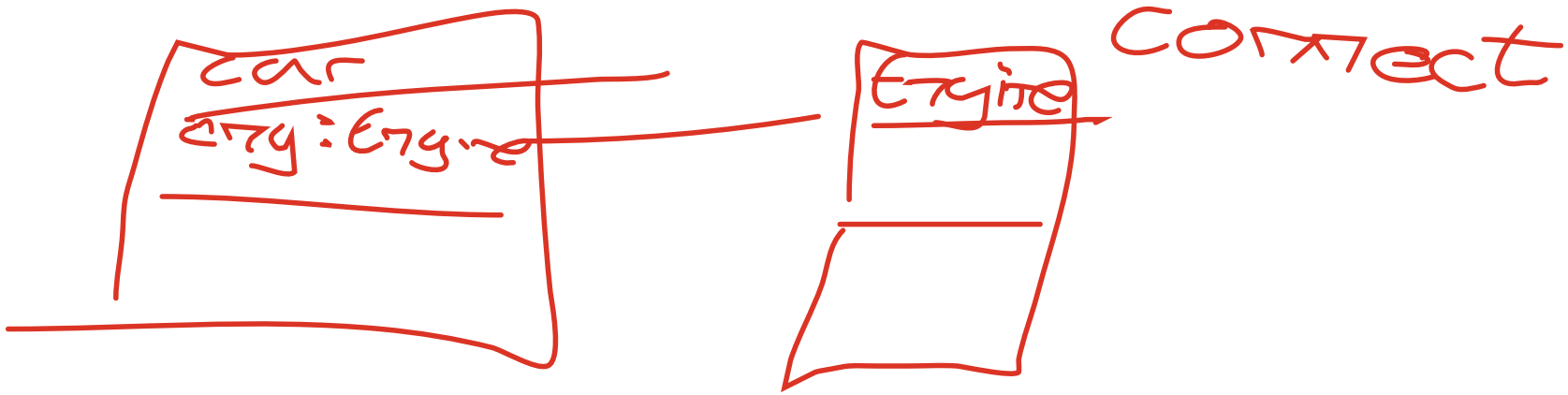
Object-Oriented Design and Contracts

- **Object-Oriented Design (OOD)** organizes software as a collection of objects.
- **Contracts**: Define responsibilities and expectations between objects.
- **Contracts** help ensure correctness and robustness.



Object-Oriented Design and Contracts

- **Object-Oriented Design (OOD)** organizes software as a collection of objects.
- **Contracts**: Define responsibilities and expectations between objects.
- **Contracts** help ensure correctness and robustness.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

UML Diagrams

- **UML** (*Unified Modeling Language*) is a standard way to visualize system design.
- Common diagrams: Class diagrams, Sequence diagrams, and Use case diagrams.
- UML helps communicate design ideas clearly.

Static

use-cases

⋮

Dynamic

activity

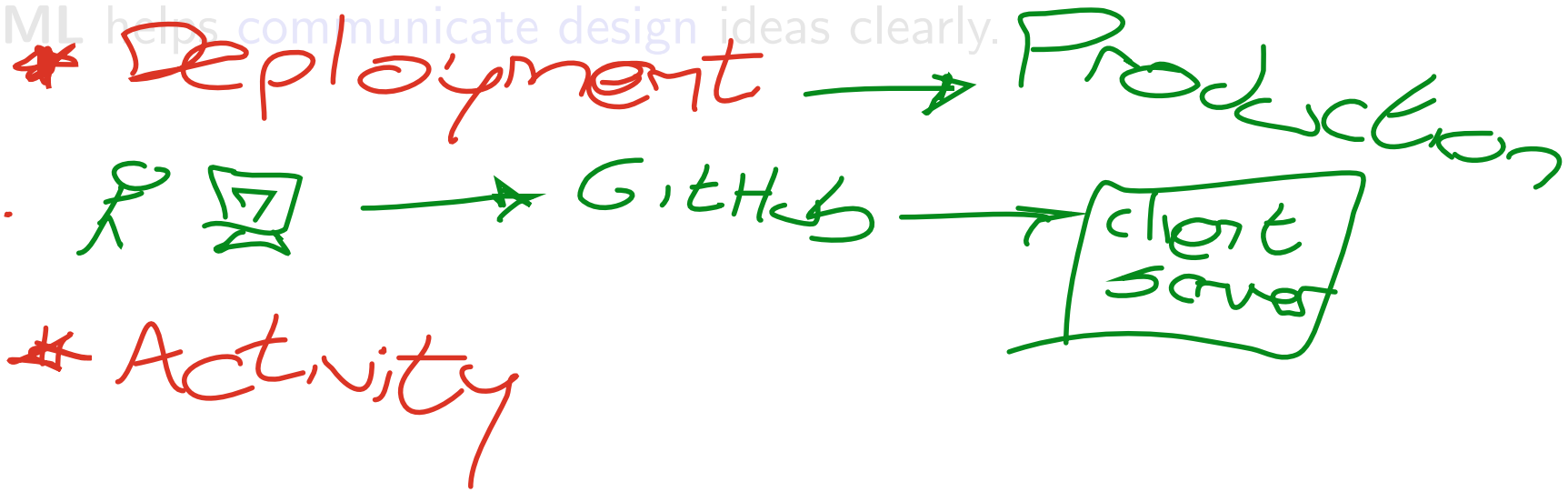
⋮

No code

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

UML Diagrams

- **UML** (*Unified Modeling Language*) is a standard way to visualize system design.
- Common diagrams: `Class diagrams`, `Sequence diagrams`, and `Use case diagrams`.
- UML helps communicate design ideas clearly.



UML Diagrams

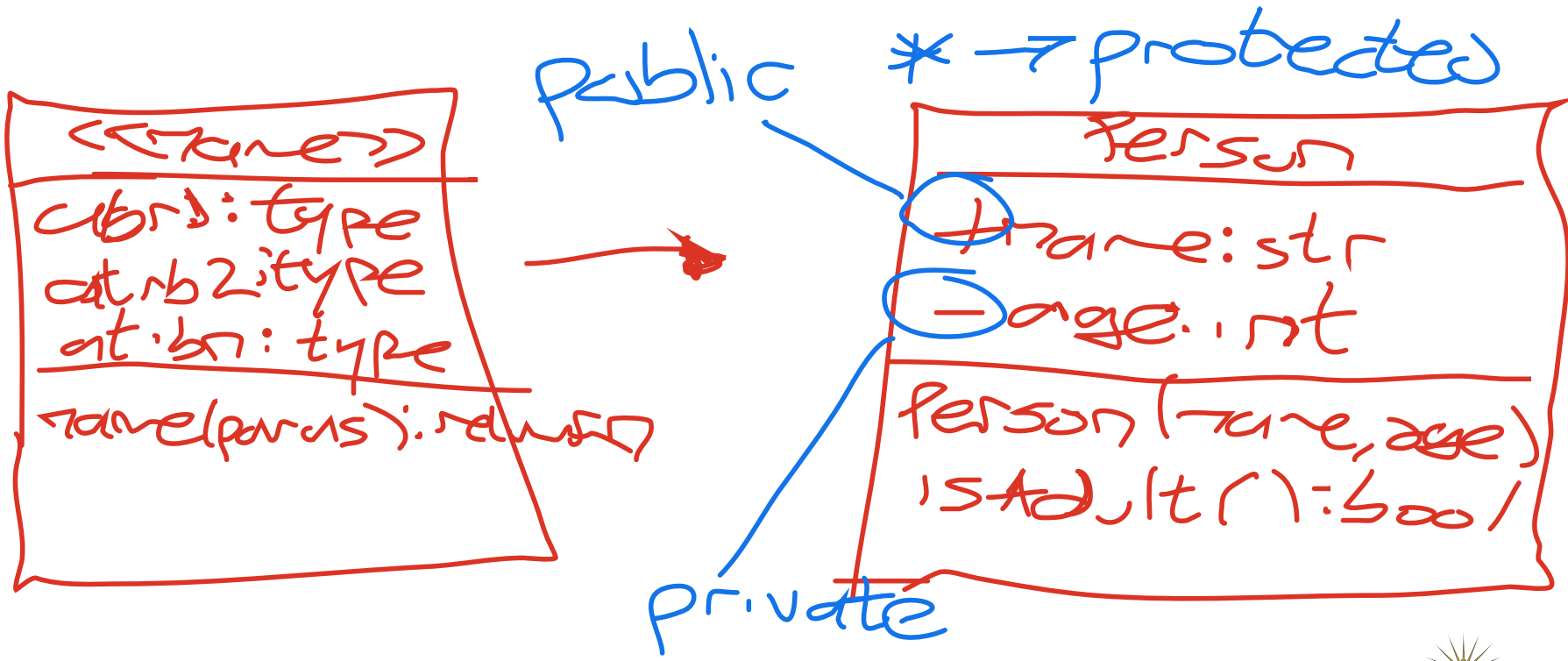
- **UML** (*Unified Modeling Language*) is a standard way to visualize system design.
- Common diagrams: `Class diagrams`, `Sequence diagrams`, and `Use case diagrams`.
- **UML** helps communicate design ideas clearly.

→ simple, free from tech

→ flexible

Class Diagrams

- **Class diagrams** show the structure of the system.
- They display **classes**, their attributes, methods, and relationships.
- Useful for both conceptual and technical design.



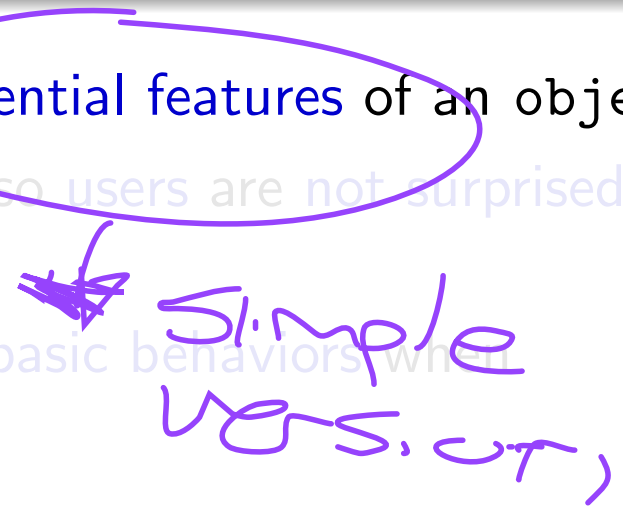
Class Diagrams

- **Class diagrams** show the structure of the system.
- They display **classes**, their attributes, methods, and relationships.
- Useful for `both conceptual and technical design`.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Abstraction

- **Abstraction** means focusing on the essential features of an `object`.
- **Rule of Least Astonishment**: Design so users are not surprised by behavior.
- Consider context, basic attributes, and basic behaviors when *designing abstractions*.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Abstraction

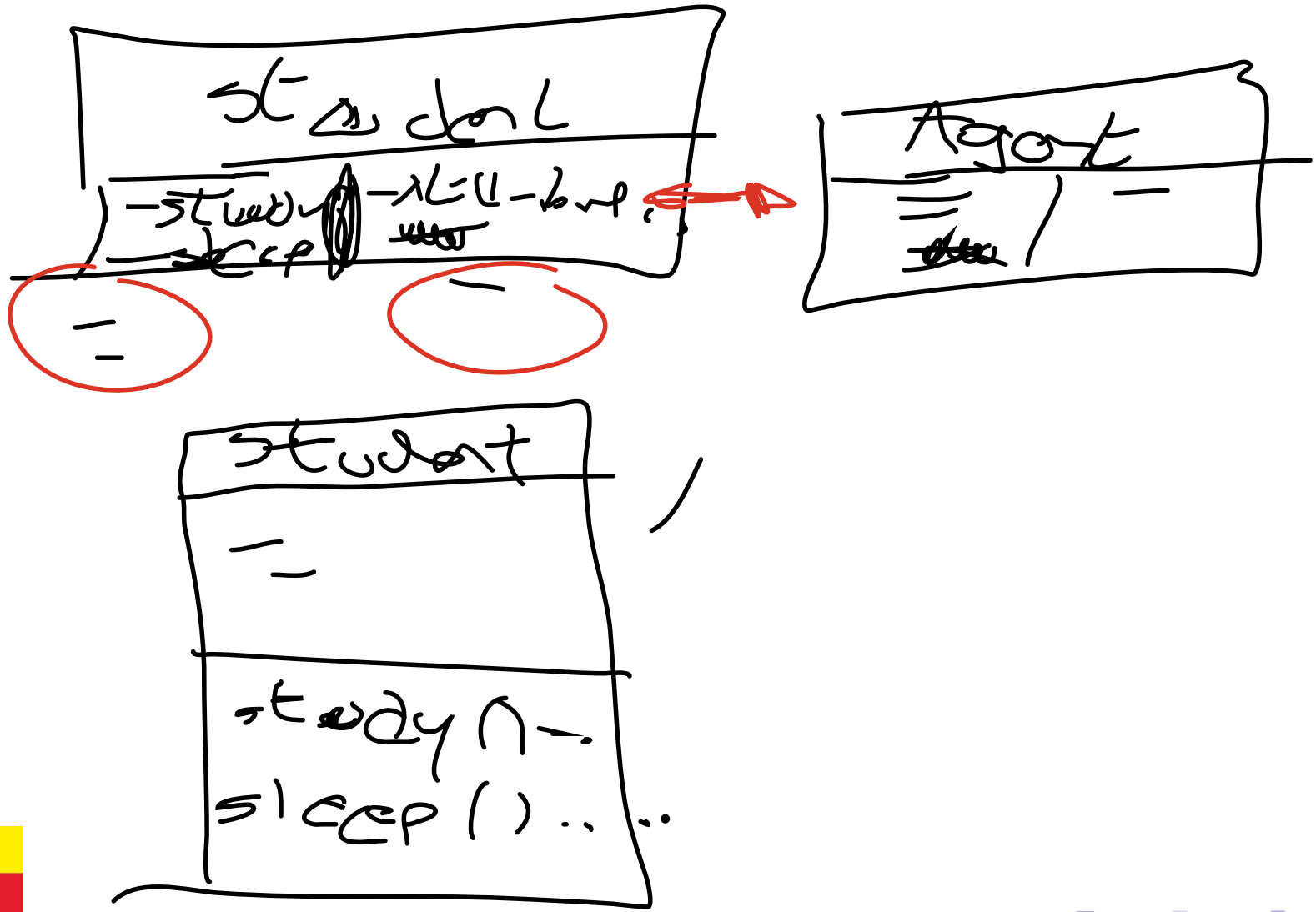
- **Abstraction** means focusing on the essential features of an `object`.
- **Rule of Least Astonishment**: Design so users are not surprised by behavior.
- Consider context, basic attributes, and basic behaviors when *designing abstractions*.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Abstraction

- **Abstraction** means focusing on the essential features of an `object`.
- **Rule of Least Astonishment**: Design so users are not surprised by behavior.
- Consider context, basic attributes, and basic behaviors when *designing abstractions*.

Abstraction & CRC Cards

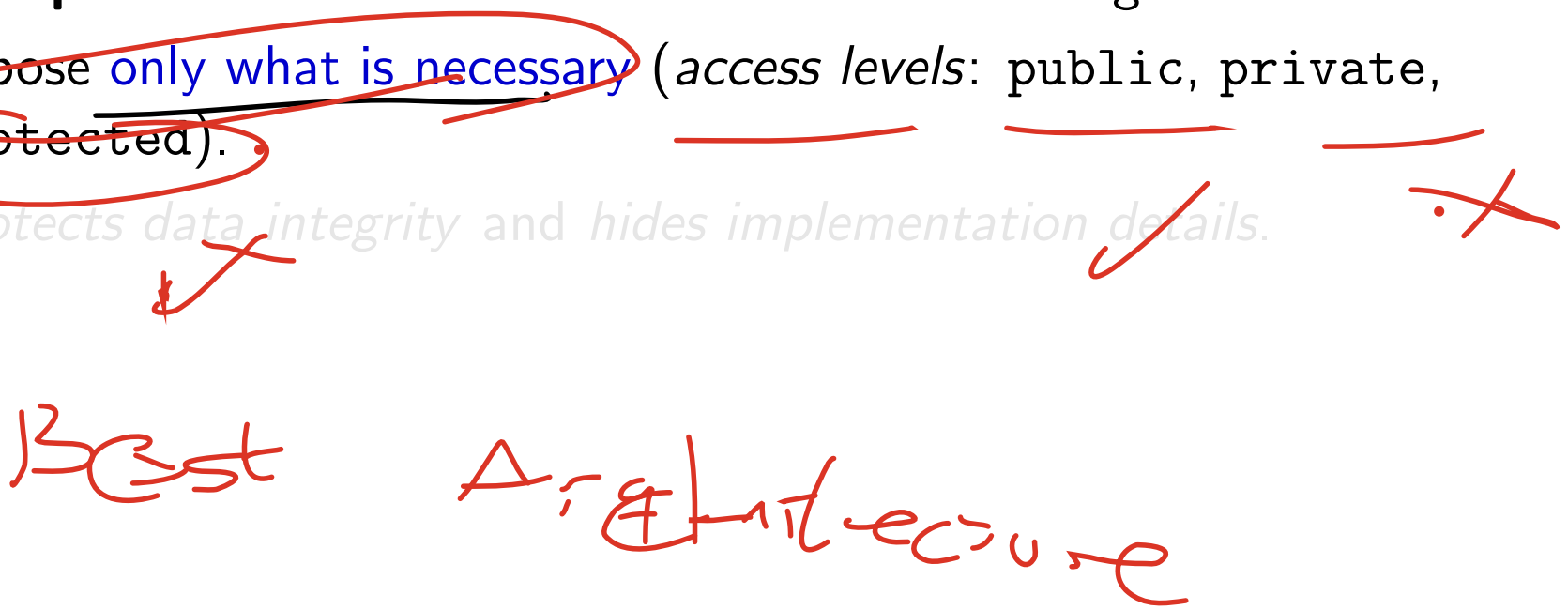


Encapsulation

- **Encapsulation** bundles attributes and methods together.
- Expose only what is necessary (*access levels*: `public`, `private`, `protected`).
- *Protects data integrity* and *hides implementation details*.

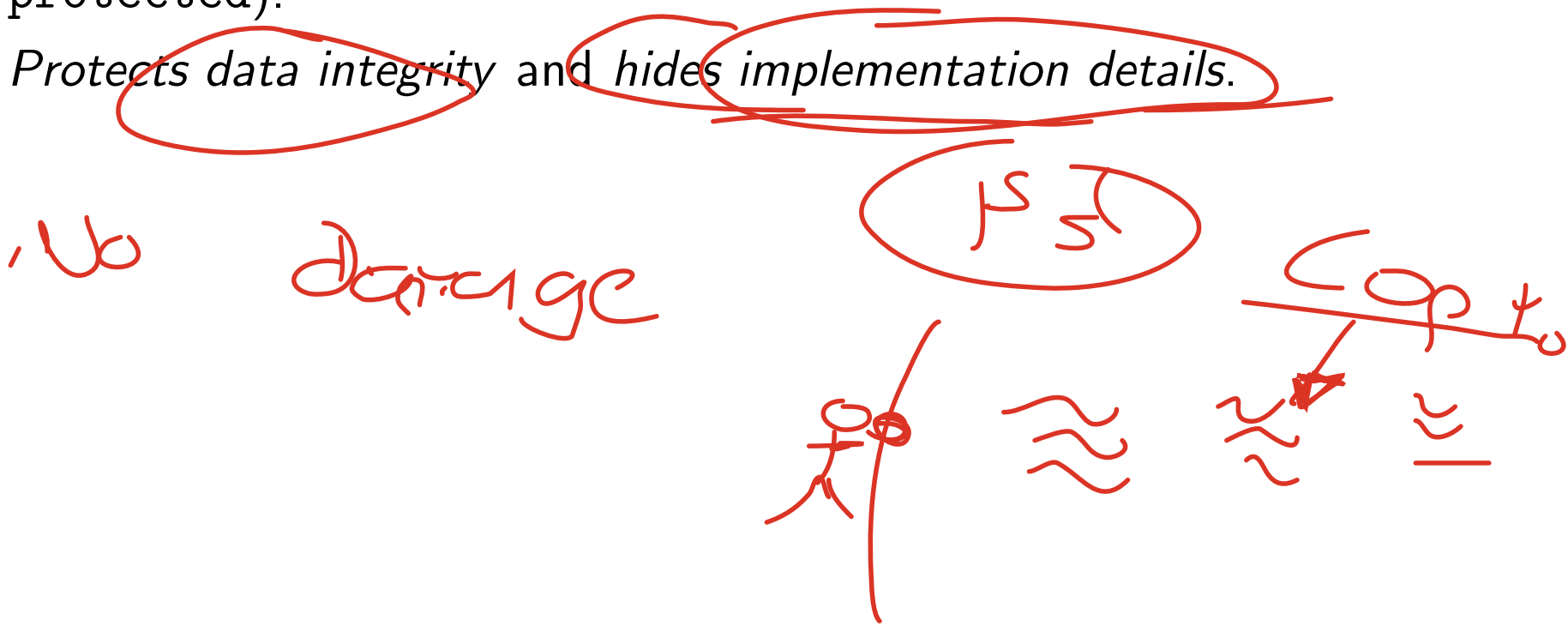
Encapsulation

- **Encapsulation** bundles attributes and methods together.
- Expose only what is necessary (*access levels*: `public`, `private`, `protected`).
- *Protects data integrity and hides implementation details.*



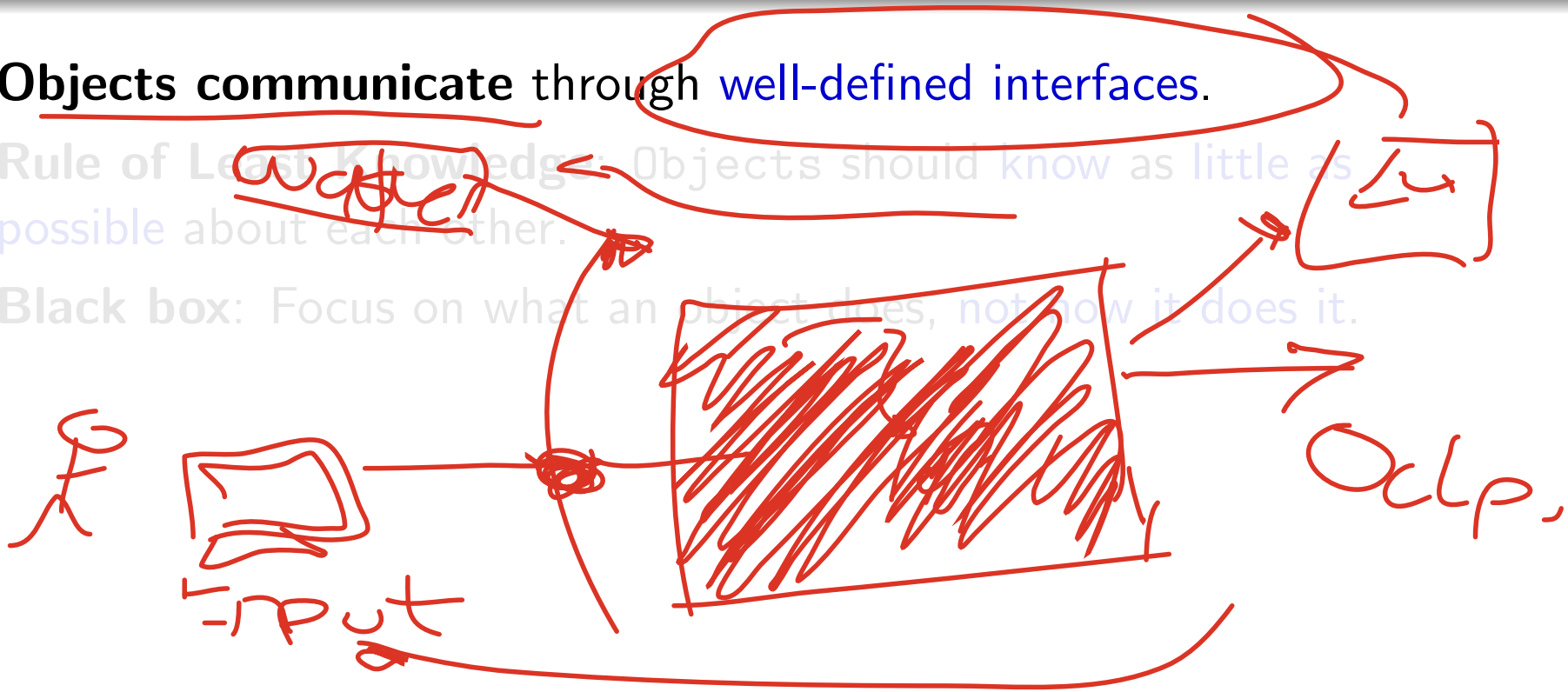
Encapsulation

- **Encapsulation** bundles attributes and methods together.
- Expose only what is necessary (*access levels*: `public`, `private`, `protected`).
- *Protects data integrity* and *hides implementation details*.



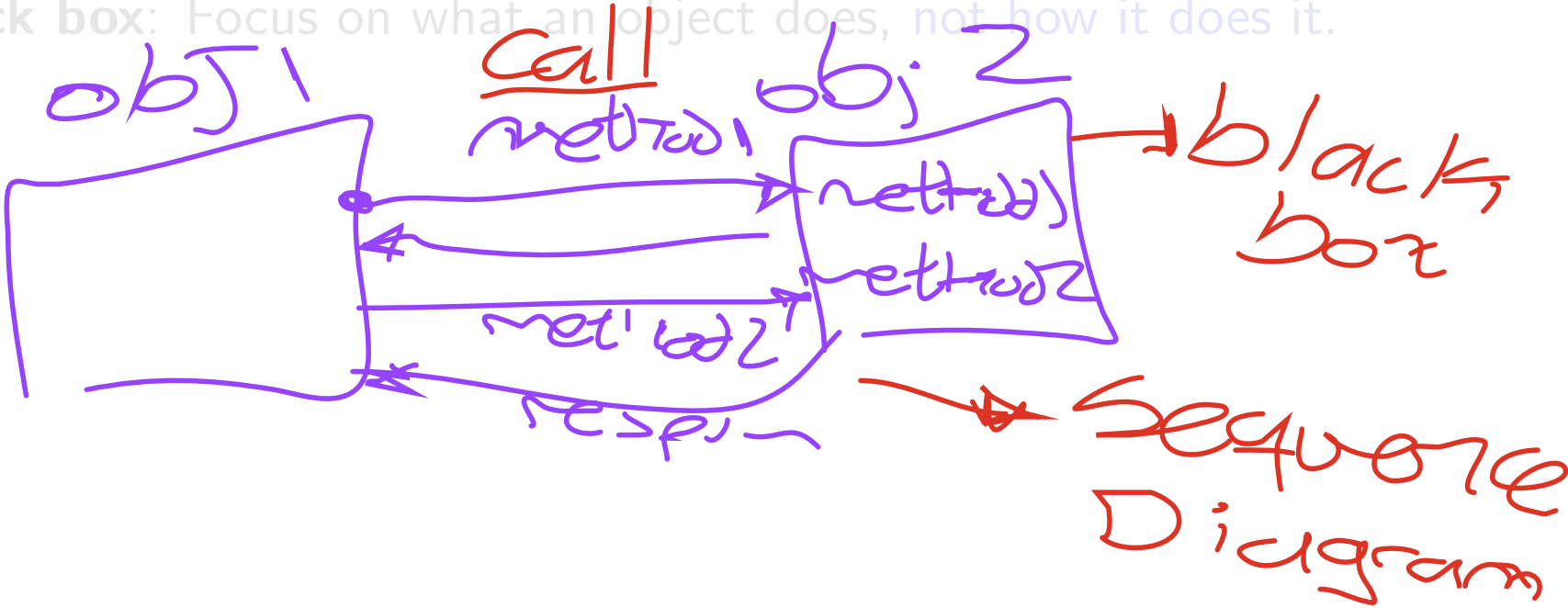
Black Box Thinking

- **Objects communicate** through well-defined interfaces.
- **Rule of Least Knowledge**: Objects should know as little as possible about each other.
- **Black box**: Focus on what an object does, not how it does it.



Black Box Thinking

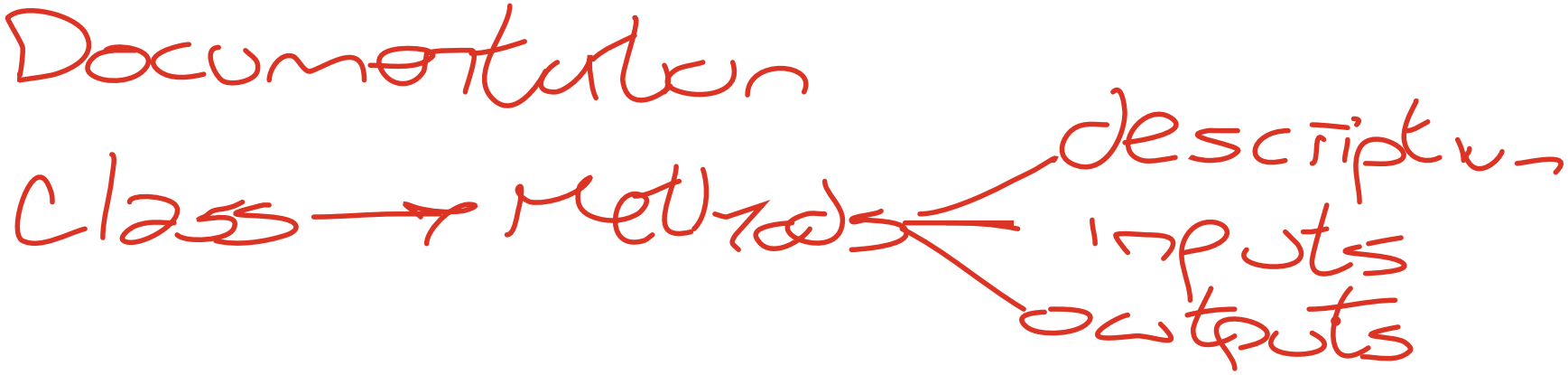
- **Objects communicate** through well-defined interfaces.
- **Rule of Least Knowledge**: Objects should know as little as possible about each other.
- **Black box**: Focus on what an object does, not how it does it.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Black Box Thinking

- **Objects communicate** through well-defined interfaces.
- **Rule of Least Knowledge**: `Objects` should know as little as possible about each other.
- **Black box**: Focus on what an object does, not how it does it.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

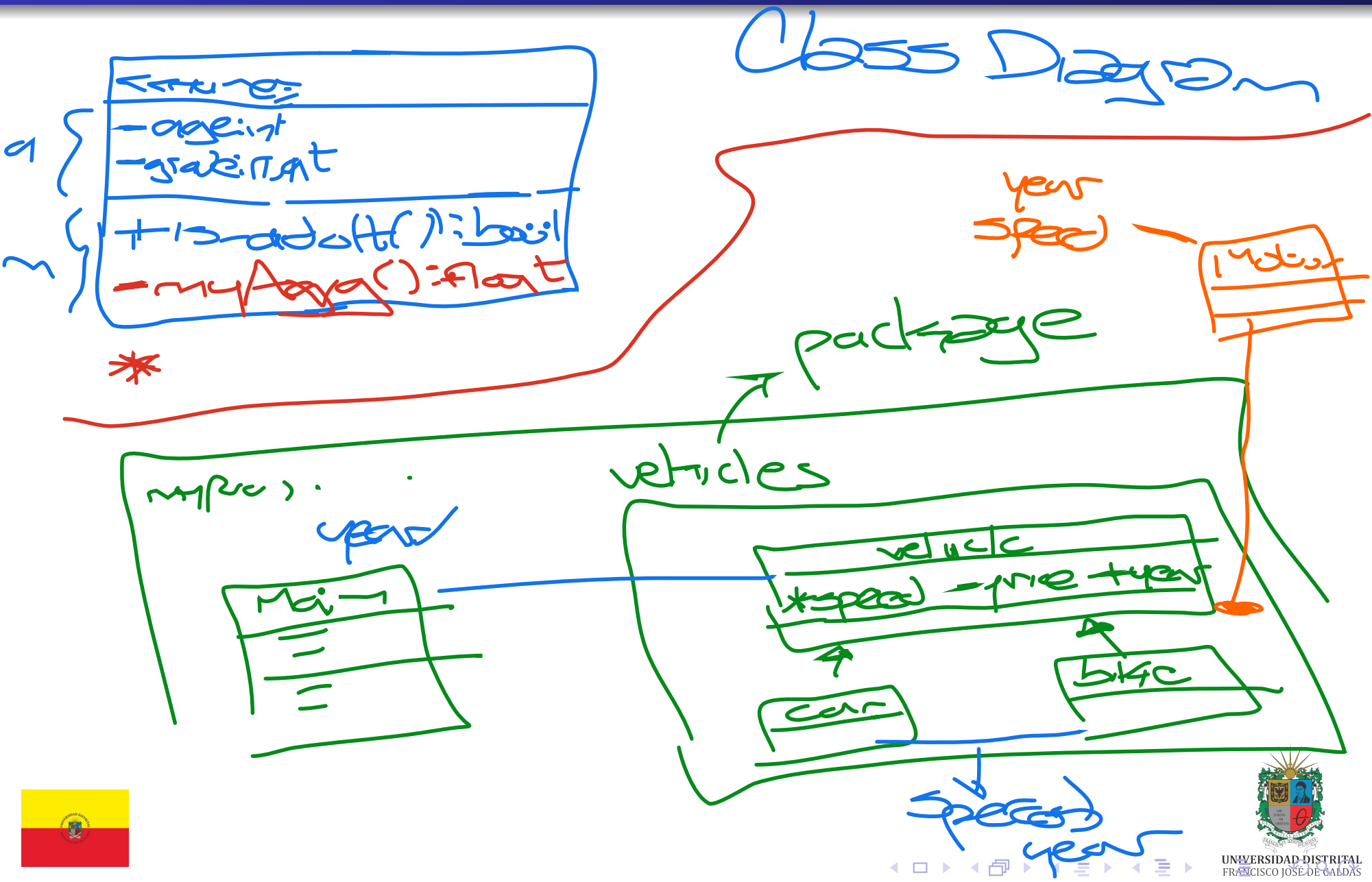
Data Integritry: Getters and Setters

```
public getName(){
    //Ask; grants → Appune
                   → reject
    return name;
}

public setAge(int newAge){
    //Ask grant, value range
    age = newAge; // f newAge<0
                  → error
}
```

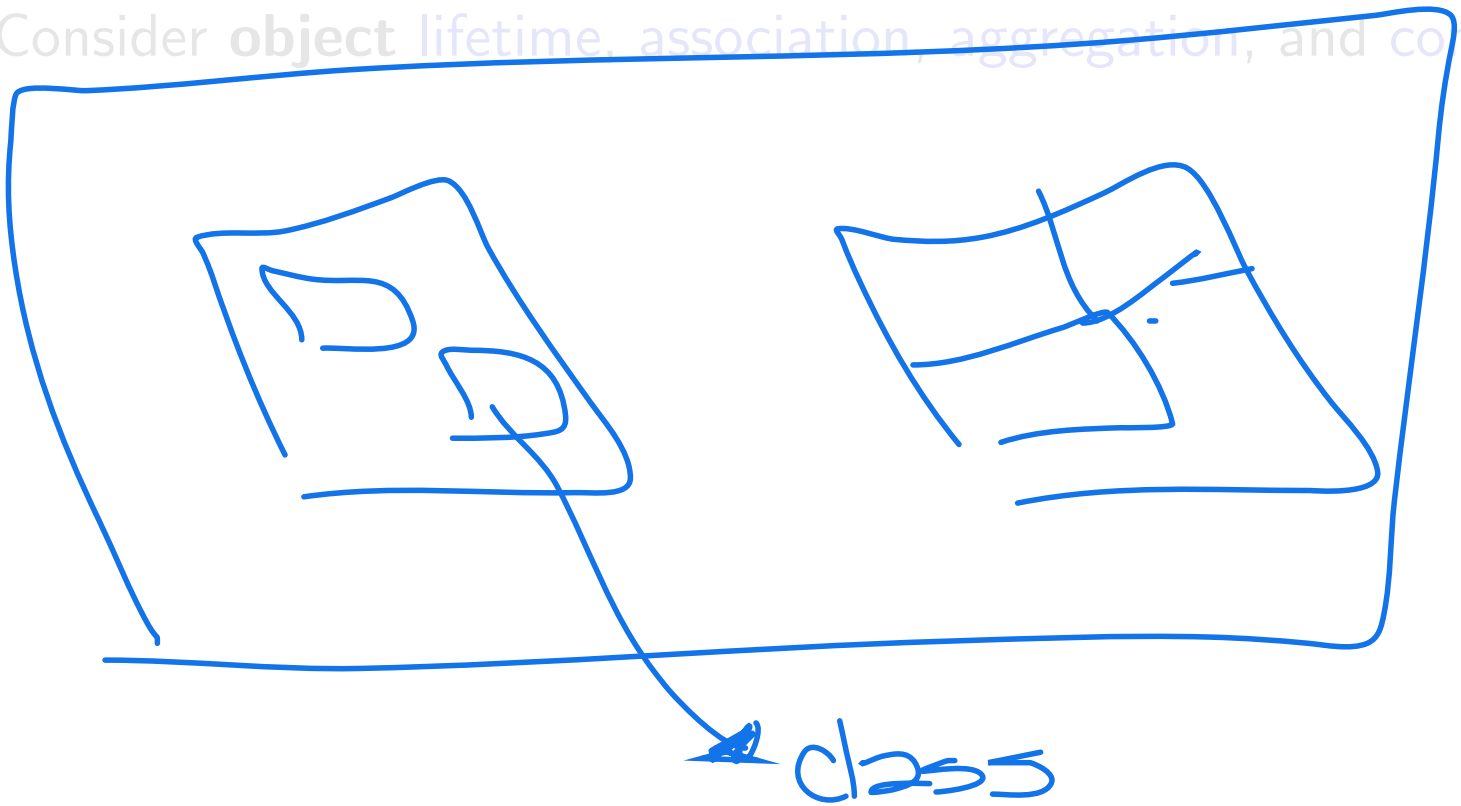
UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Encapsulation & UML



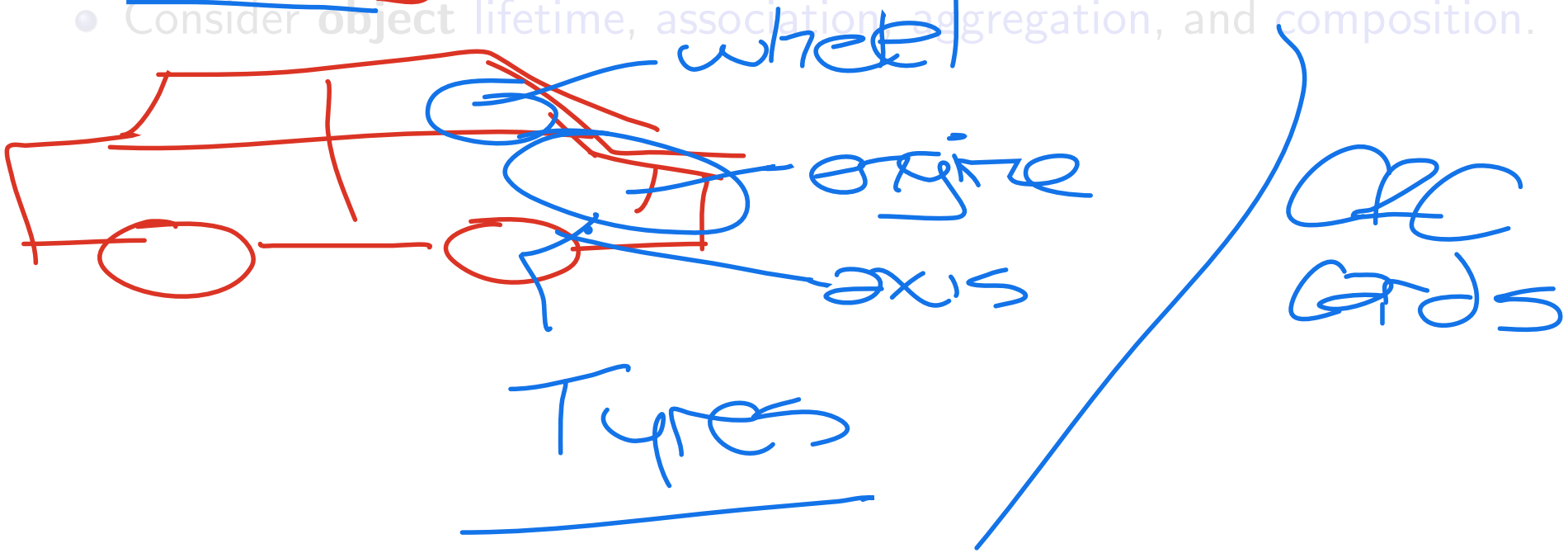
Decomposition

- **Decomposition**: Divide and conquer by breaking the system into smaller parts.
- **Separation of Concerns**: Each part should have a clear responsibility.
- Consider **object** lifetime, association, aggregation, and composition.



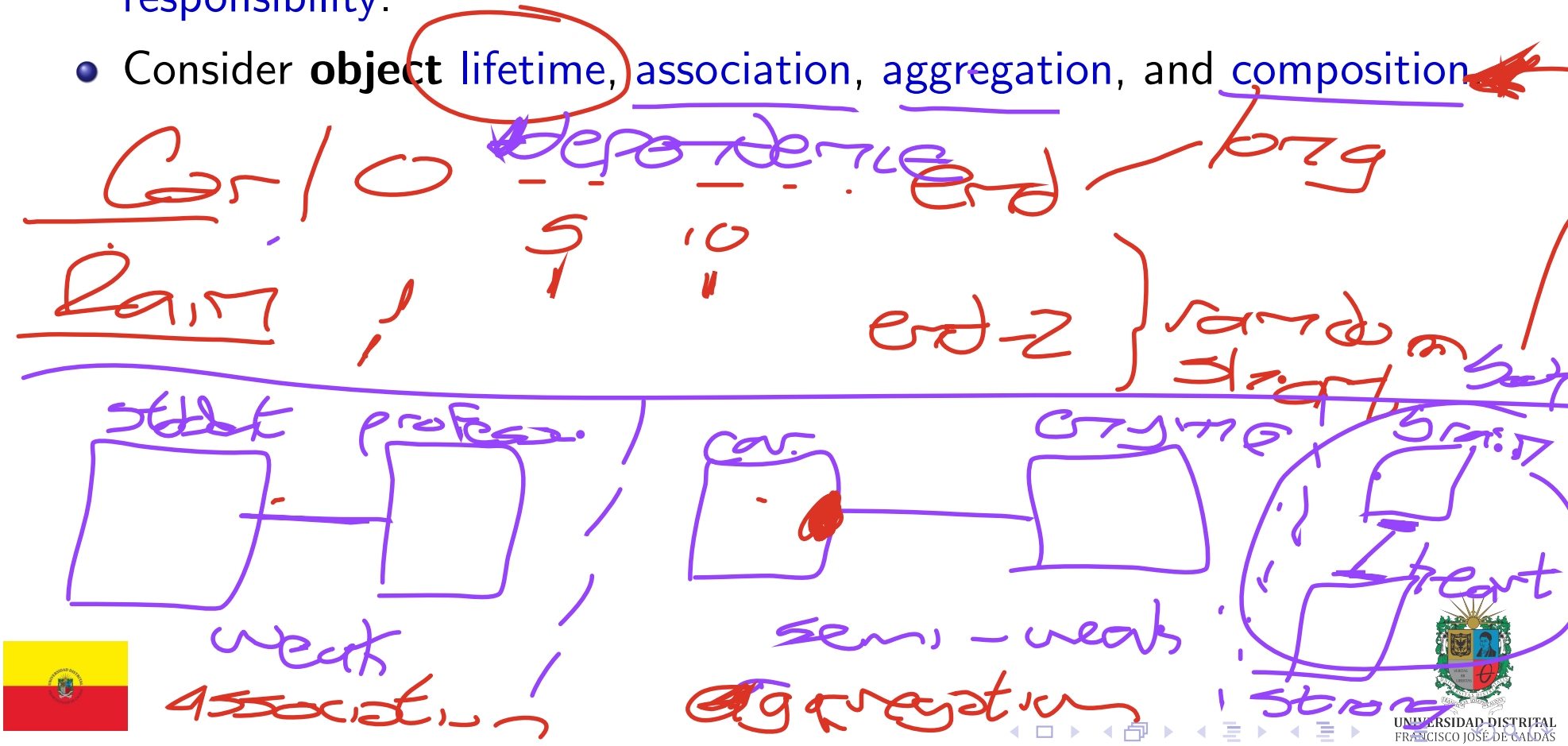
Decomposition

- **Decomposition**: Divide and conquer by breaking the system into smaller parts.
- **Separation of Concerns**: Each part should have a clear responsibility.
- Consider **object** lifetime, association, aggregation, and composition.

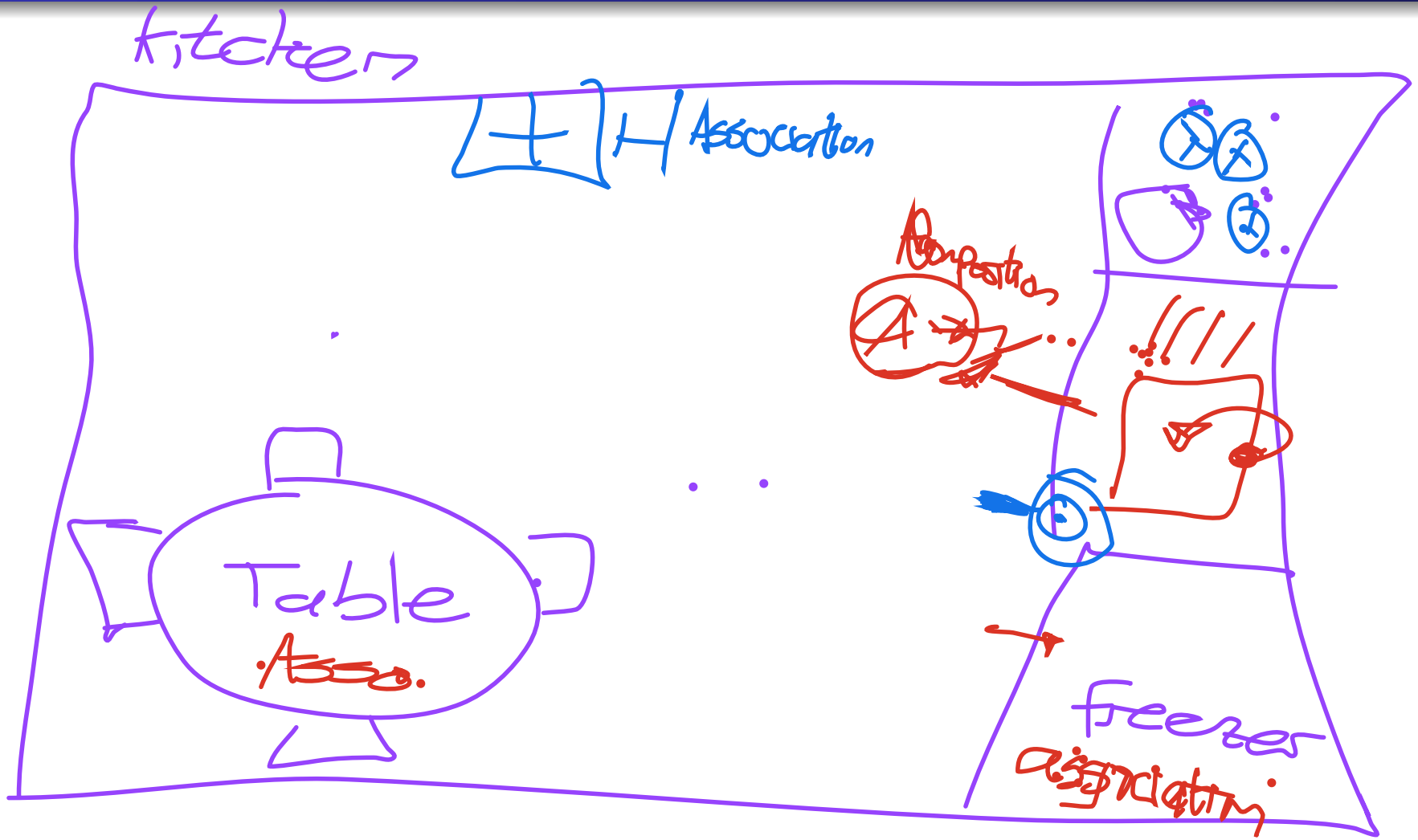


Decomposition

- **Decomposition**: Divide and conquer by breaking the system into smaller parts.
- **Separation of Concerns**: Each part should have a clear responsibility.
- Consider **object** lifetime, association, aggregation, and composition.

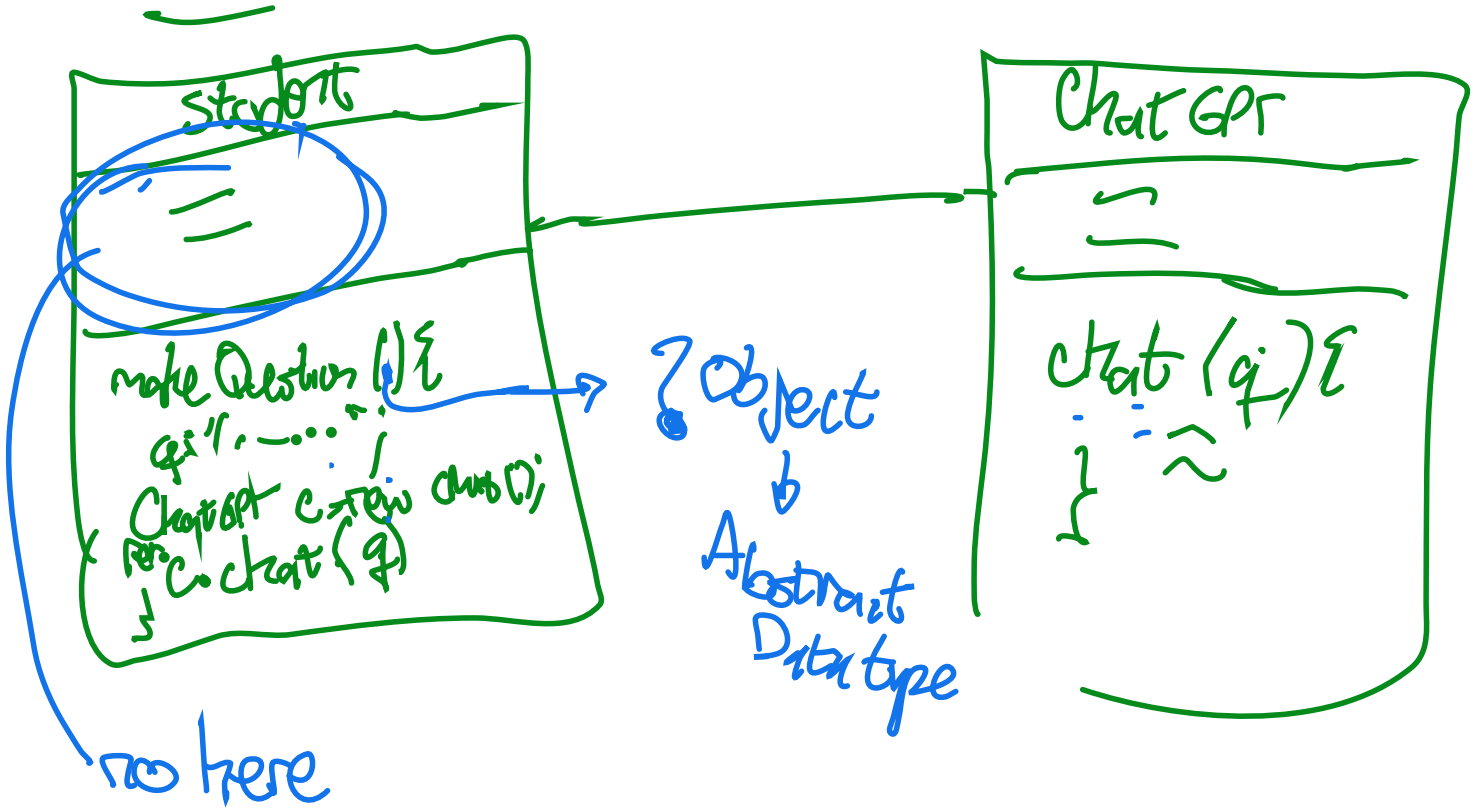


Decomposition Example: Kitchen in a House



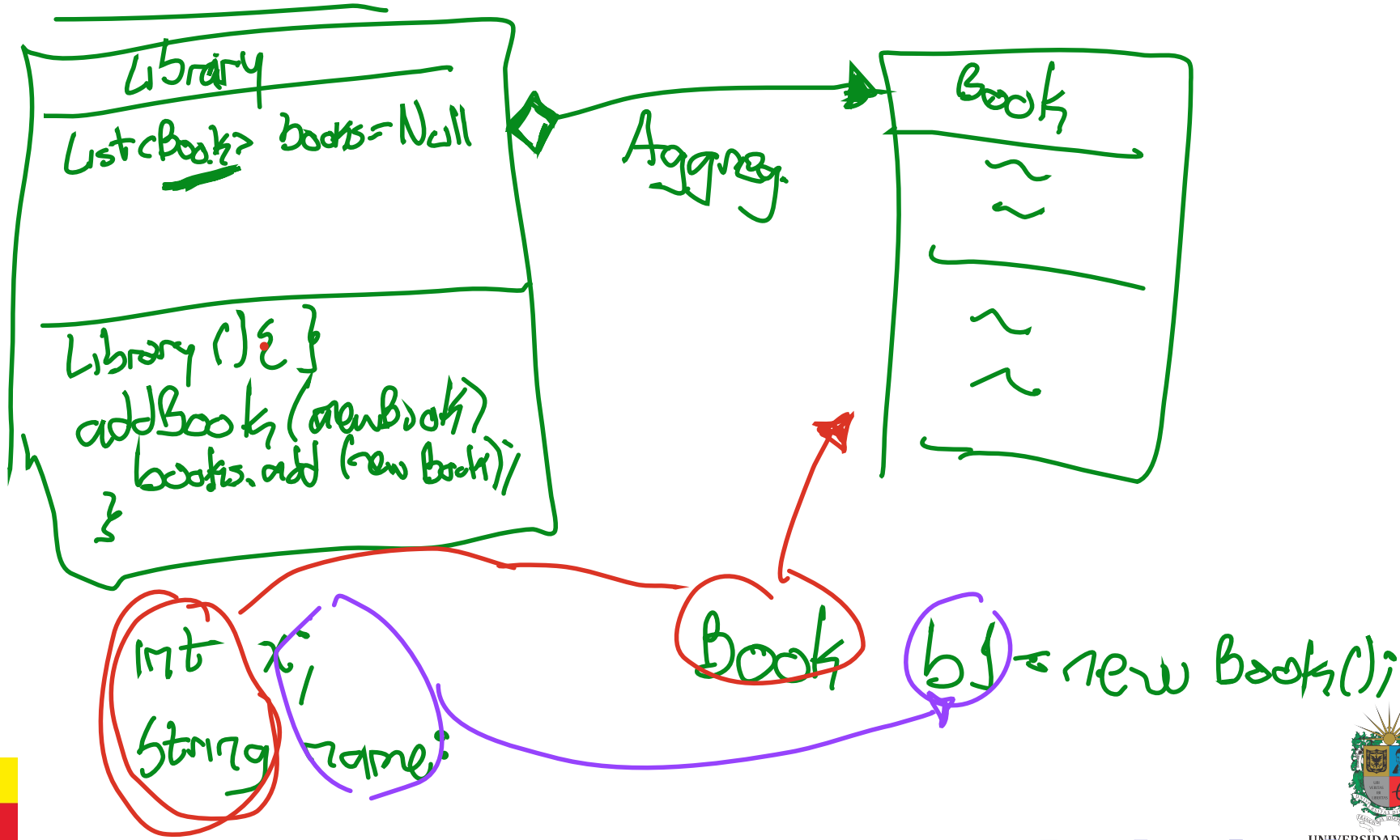
Association

A **relationship** between two classes where one class uses or interacts with another class.



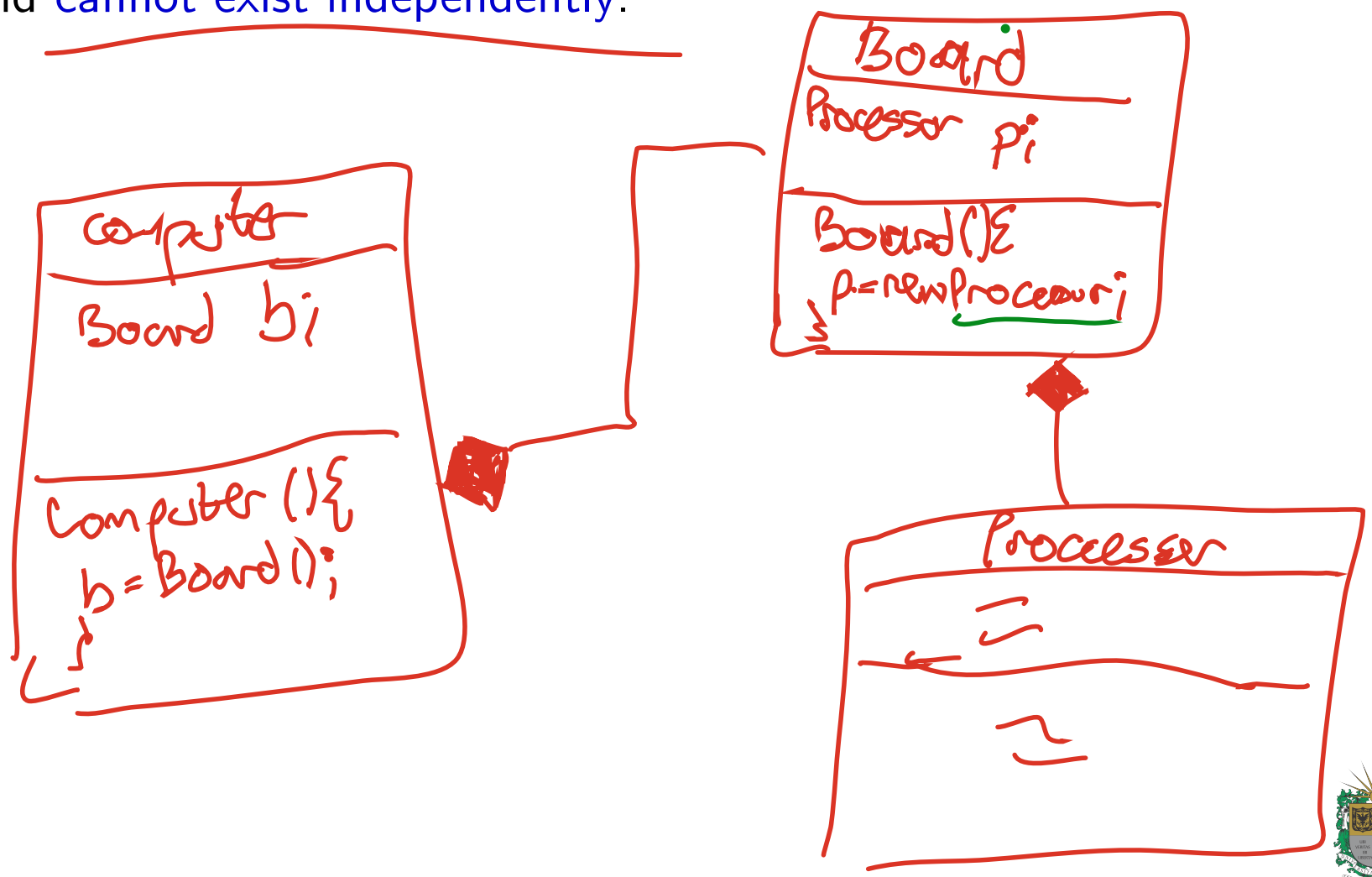
Aggregation

A **whole-part relationship** where one class is a part of another class, but can exist independently.



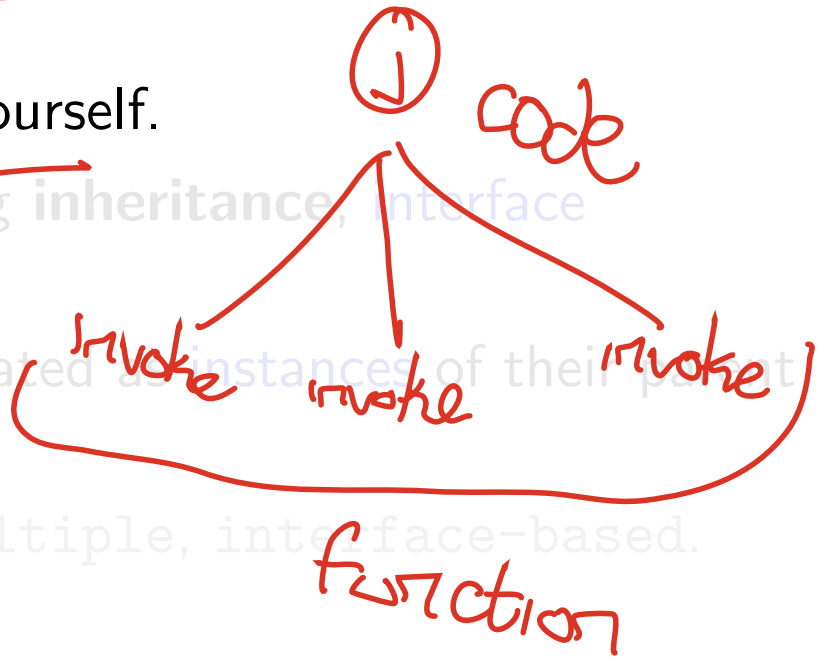
Composition

A **stronger whole-part relationship** where one class is a part of another class and cannot exist independently.



Generalization

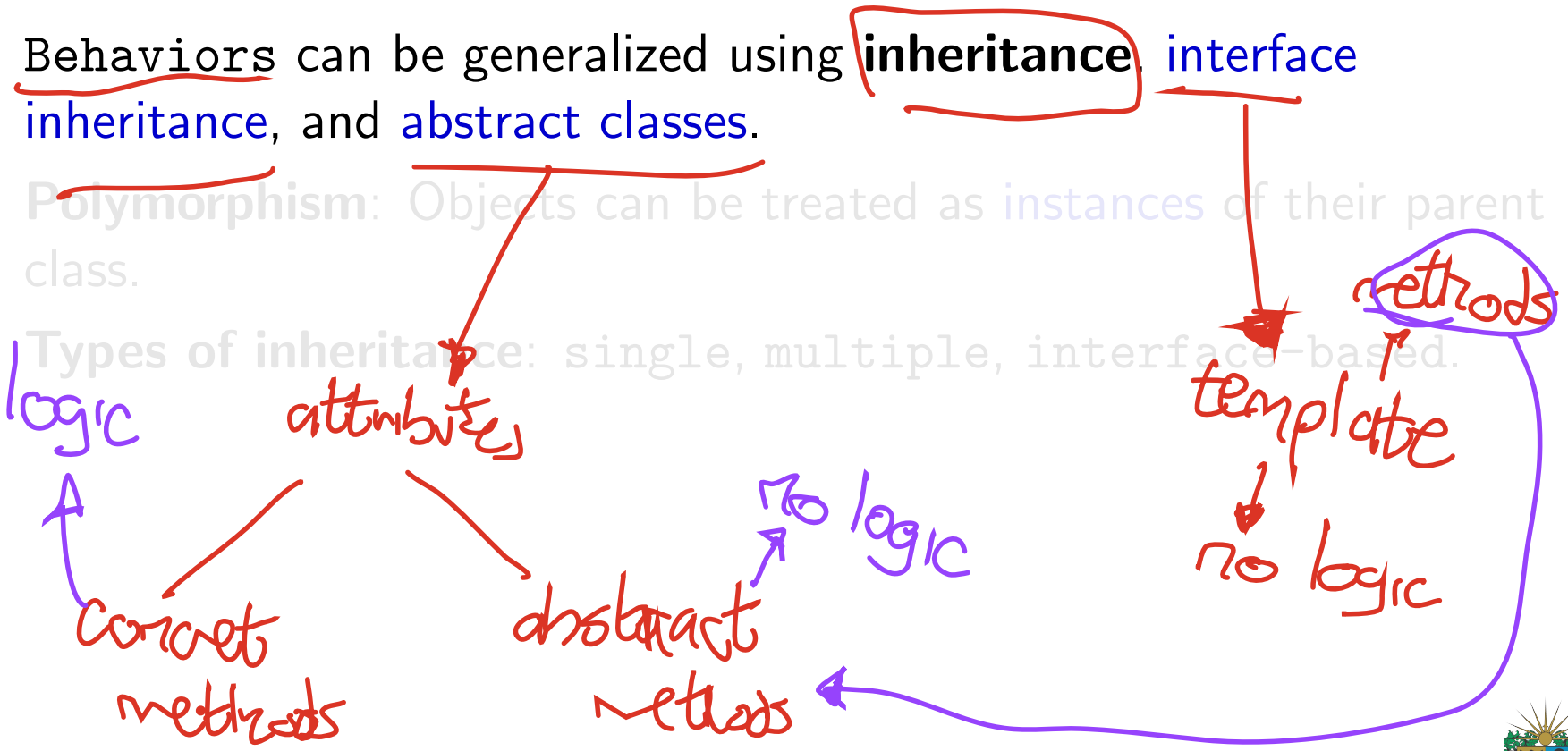
- **Generalization** eliminates redundancy by extracting common features.
- **D.R.Y. Principle**: Don't Repeat Yourself.
- Behaviors can be generalized using **inheritance**, interface inheritance, and abstract classes.
- **Polymorphism**: Objects can be treated as instances of their parent class.
- **Types of inheritance**: `single`, `multiple`, `interface-based`.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Generalization

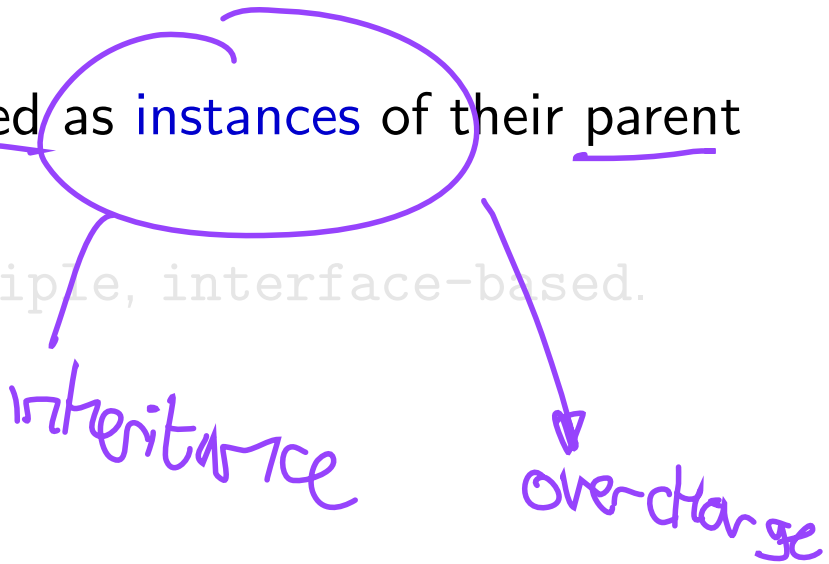
- **Generalization** eliminates redundancy by extracting common features.
- **D.R.Y. Principle**: Don't Repeat Yourself.
- `Behaviors` can be generalized using **inheritance**, interface inheritance, and abstract classes.
- **Polymorphism**: Objects can be treated as instances of their parent class.
- **Types of inheritance**: `single`, `multiple`, `interface-based`.



UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

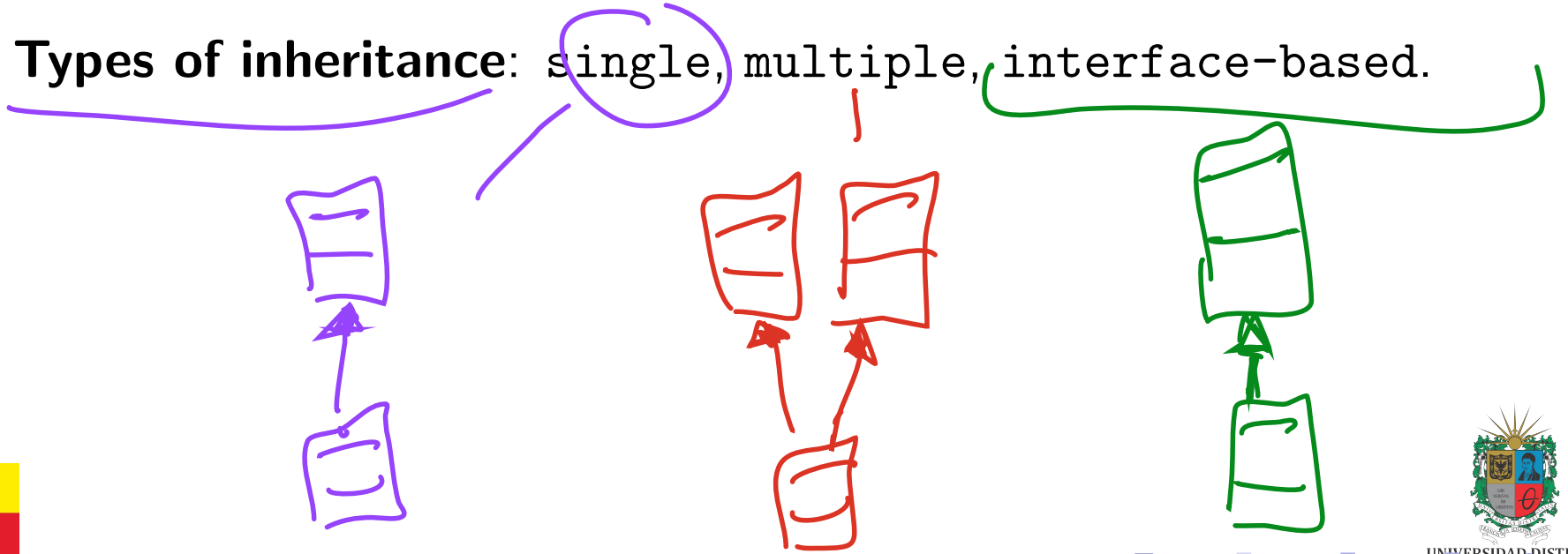
Generalization

- **Generalization** eliminates redundancy by extracting common features.
- **D.R.Y. Principle**: Don't Repeat Yourself.
- `Behaviors` can be generalized using **inheritance**, interface inheritance, and abstract classes.
- **Polymorphism**: Objects can be treated as instances of their parent class.
- **Types of inheritance**: `single`, `multiple`, `interface-based`.



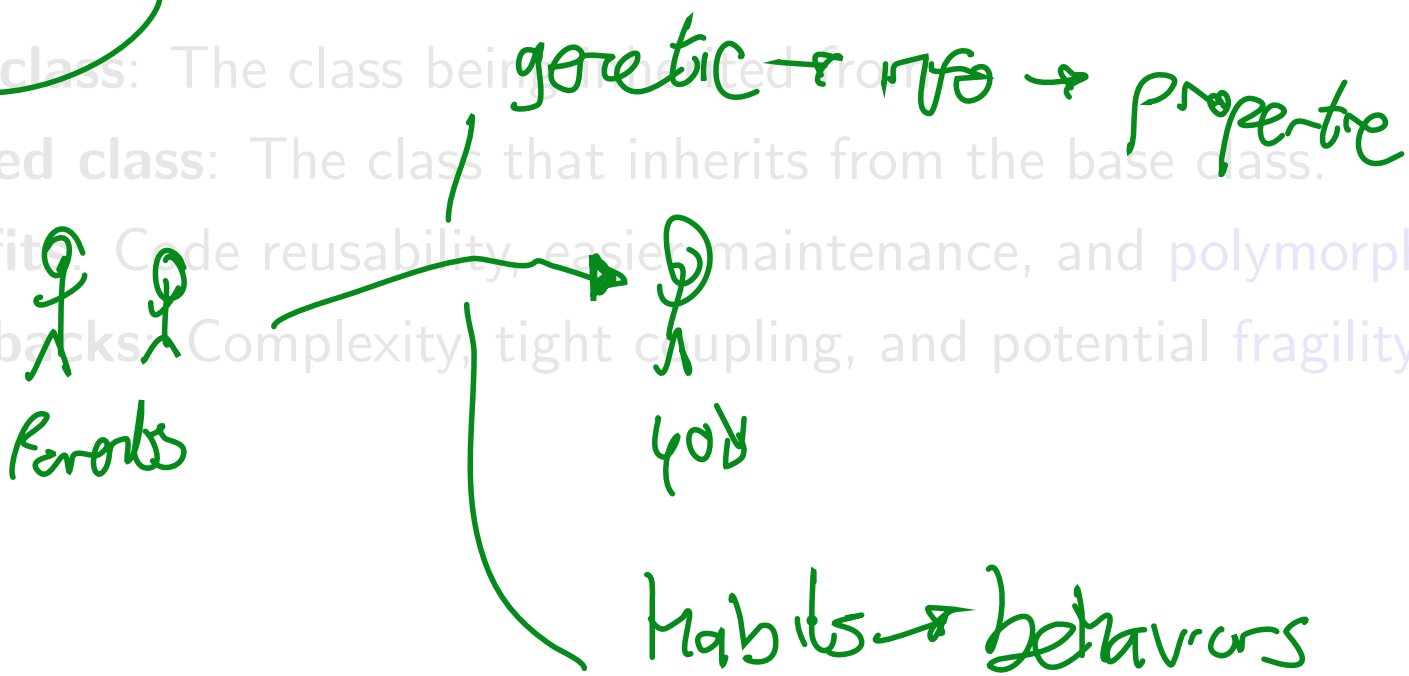
Generalization

- **Generalization** eliminates redundancy by extracting common features.
- **D.R.Y. Principle**: Don't Repeat Yourself.
- `Behaviors` can be generalized using **inheritance**, interface inheritance, and abstract classes.
- **Polymorphism**: Objects can be treated as instances of their parent class.
- **Types of inheritance**: `single`, `multiple`, `interface-based`.



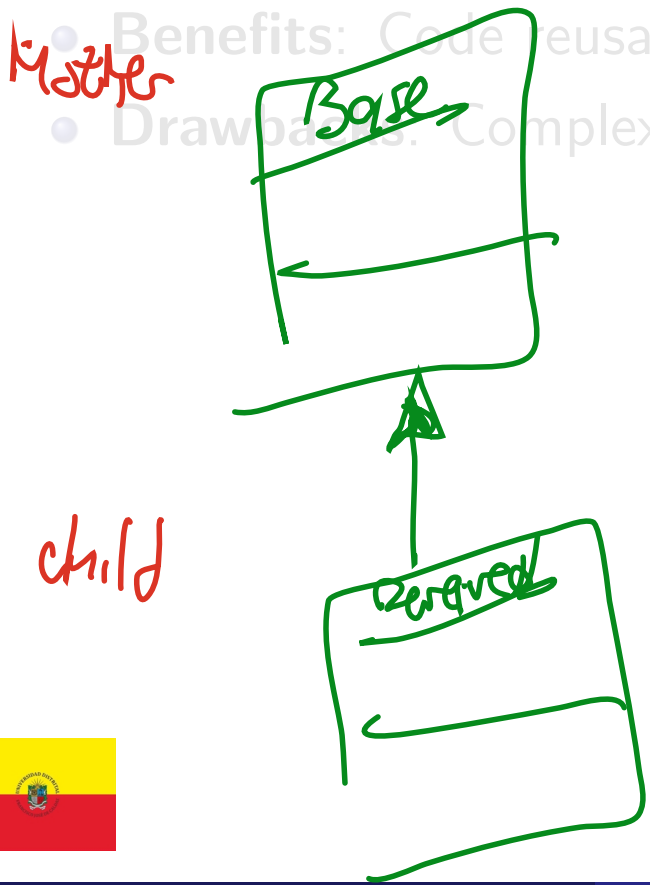
Inheritance

- **Inheritance** allows a class to inherit properties and methods from another class.
- **Base class**: The class being inherited from.
- **Derived class**: The class that inherits from the base class.
- **Benefits**: Code reusability, easier maintenance, and polymorphism.
- **Drawbacks**: Complexity, tight coupling, and potential fragility.



Inheritance

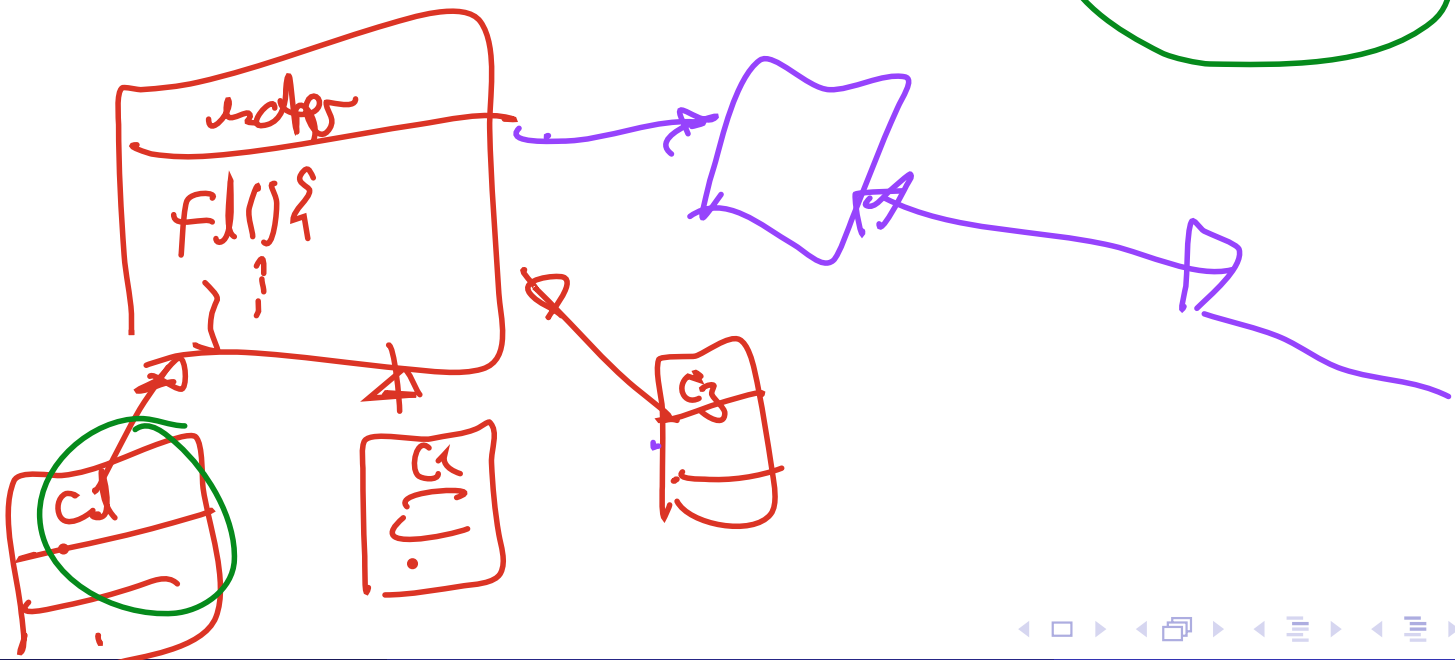
- **Inheritance** allows a class to inherit properties and methods from another class.
- **Base class**: The class being inherited from.
- **Derived class**: The class that inherits from the base class.
- **Benefits**: Code reusability, easier maintenance, and polymorphism.
- **Drawbacks**: Complexity, tight coupling, and potential fragility.



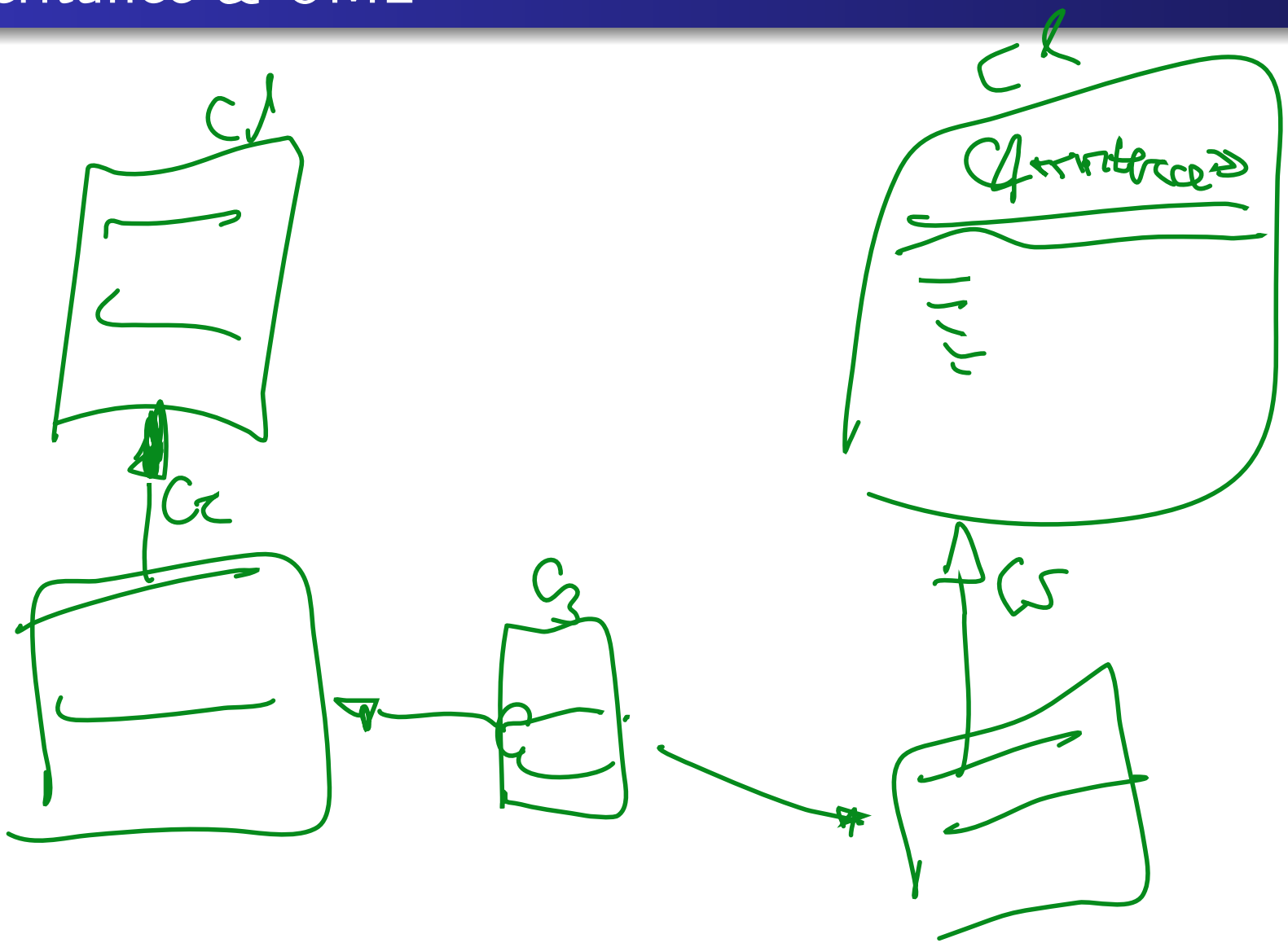
UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Inheritance

- **Inheritance** allows a class to inherit properties and methods from another class.
- **Base class**: The class being inherited from.
- **Derived class**: The class that inherits from the base class.
- **Benefits**: Code reusability, easier maintenance, and polymorphism.
- **Drawbacks**: Complexity, tight coupling, and potential fragility.



Inheritance & UML



Interface Inheritance

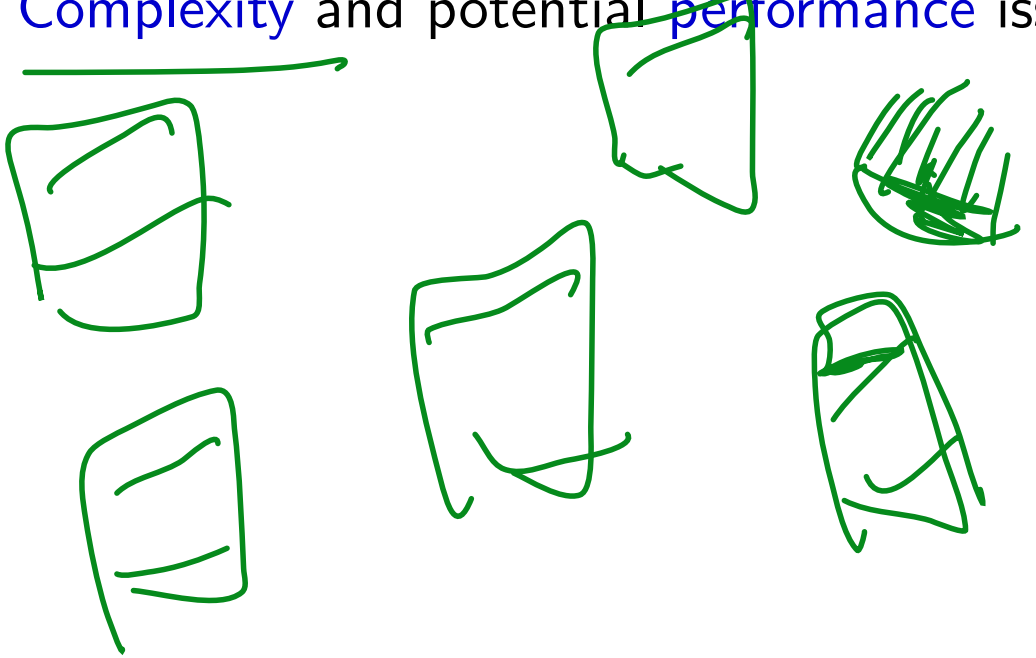
- **Interface inheritance** allows a class to implement an interface without inheriting its implementation.
- **Interfaces** define a contract that classes must adhere to.
- **Benefits**: `Flexibility`, `code reusability`, and `easier maintenance`.
- **Drawbacks**: Complexity and potential performance issues.

Interface Inheritance

- **Interface inheritance** allows a class to implement an interface without inheriting its implementation.
- **Interfaces** define a contract that classes must adhere to.
- **Benefits**: Flexibility, code reusability, and easier maintenance.
- **Drawbacks**: Complexity and potential performance issues.

Interface Inheritance

- **Interface inheritance** allows a class to implement an interface without inheriting its implementation.
- **Interfaces** define a contract that classes must adhere to.
- **Benefits**: `Flexibility`, `code reusability`, and `easier maintenance`.
- **Drawbacks**: Complexity and potential performance issues.



Polymorphism by Inheritance

- **Polymorphism** allows objects of different classes to be treated as objects of a common superclass.
- **Benefits**: `Code reusability`, `flexibility`, and easier maintenance.
- **Drawbacks**: Complexity and potential performance issues.

Polymorphism by Inheritance

- **Polymorphism** allows objects of different classes to be treated as objects of a common superclass.
- **Benefits**: `Code reusability`, `flexibility`, and easier maintenance.
- **Drawbacks**: Complexity and potential performance issues.

Polymorphism by Inheritance

- **Polymorphism** allows objects of different classes to be treated as objects of a common superclass.
- **Benefits**: `Code reusability`, `flexibility`, and easier maintenance.
- **Drawbacks**: Complexity and potential performance issues.

Polymorphism by Overcharge

- **Polymorphism by overloading** allows multiple methods with the same name but different parameters.
- **Benefits**: Improves readability and reduces complexity.
- **Drawbacks**: Can lead to confusion if not used carefully.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Polymorphism by Overcharge

- **Polymorphism by overloading** allows multiple methods with the same name but different parameters.
- **Benefits**: Improves readability and reduces complexity.
- **Drawbacks**: Can lead to confusion if not used carefully.

Polymorphism by Overcharge

- **Polymorphism by overloading** allows multiple methods with the same name but different parameters.
- **Benefits**: Improves readability and reduces complexity.
- **Drawbacks**: Can lead to confusion if not used carefully.

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Outline

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Thanks!

Questions?

Repo: *https://github.com/EngAndres/ud-public/tree/main/courses/object-oriented-programming*

UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS