

# OBJECT-ORIENTED MODELLING

## Object-Oriented Programming

Author: Eng. Carlos Andrés Sierra, M.Sc.  
[cavirguezs@udistrital.edu.co](mailto:cavirguezs@udistrital.edu.co)

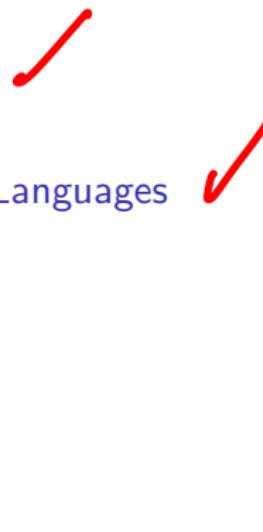
Full-time Adjunct Professor  
Computer Engineering Program  
School of Engineering  
Universidad Distrital Francisco José de Caldas

2025-III



# Outline

- 1 Creating Models in Design
- 2 Evolution of Programming Languages
- 3 Four Design Principles
- 4 SOLID Principles



# Outline

1 Creating Models in Design

2 Evolution of Programming Languages

3 Four Design Principles

4 SOLID Principles



# 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.



# 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.

1

2



# 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.
- list → need*



# 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.

business vocabulary



# Design Based on the Problem

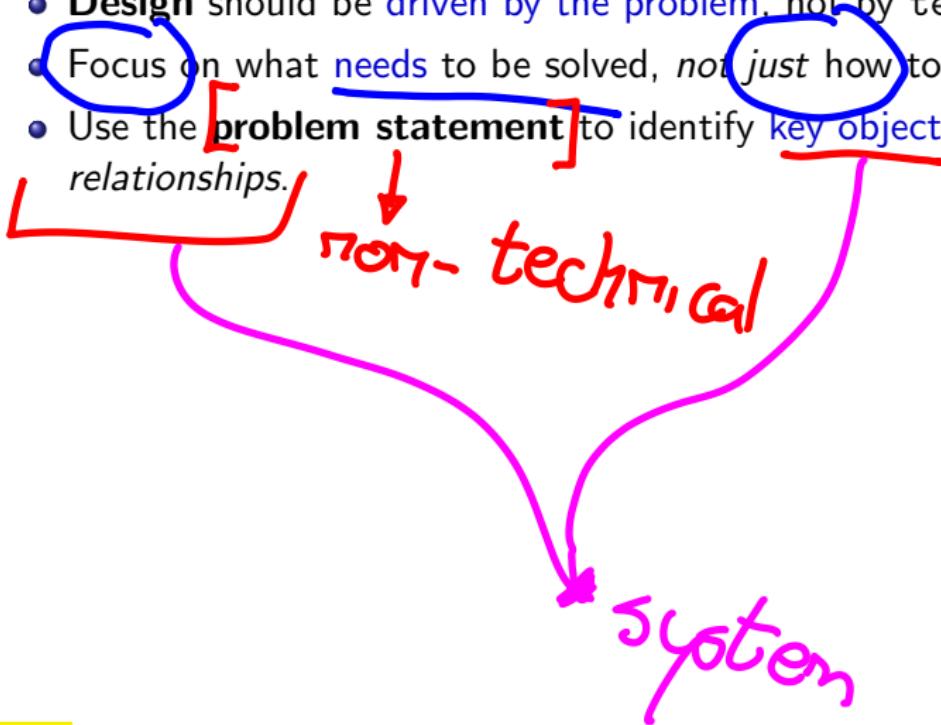


- **Design** should be driven by the problem, not by technology.
- Focus on what needs to be solved, *not just* how to implement it.
- Use the **problem statement** to identify key objects and their *relationships*.



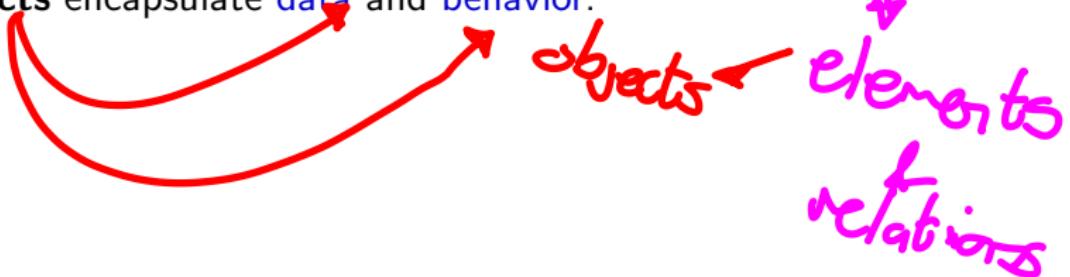
# Design Based on the Problem

- **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.



# 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.

- **Diagrams**
  - Before  
Diagram

UML  
Unified  
modeling  
language  
draw.io



# 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.

+ user

tech team

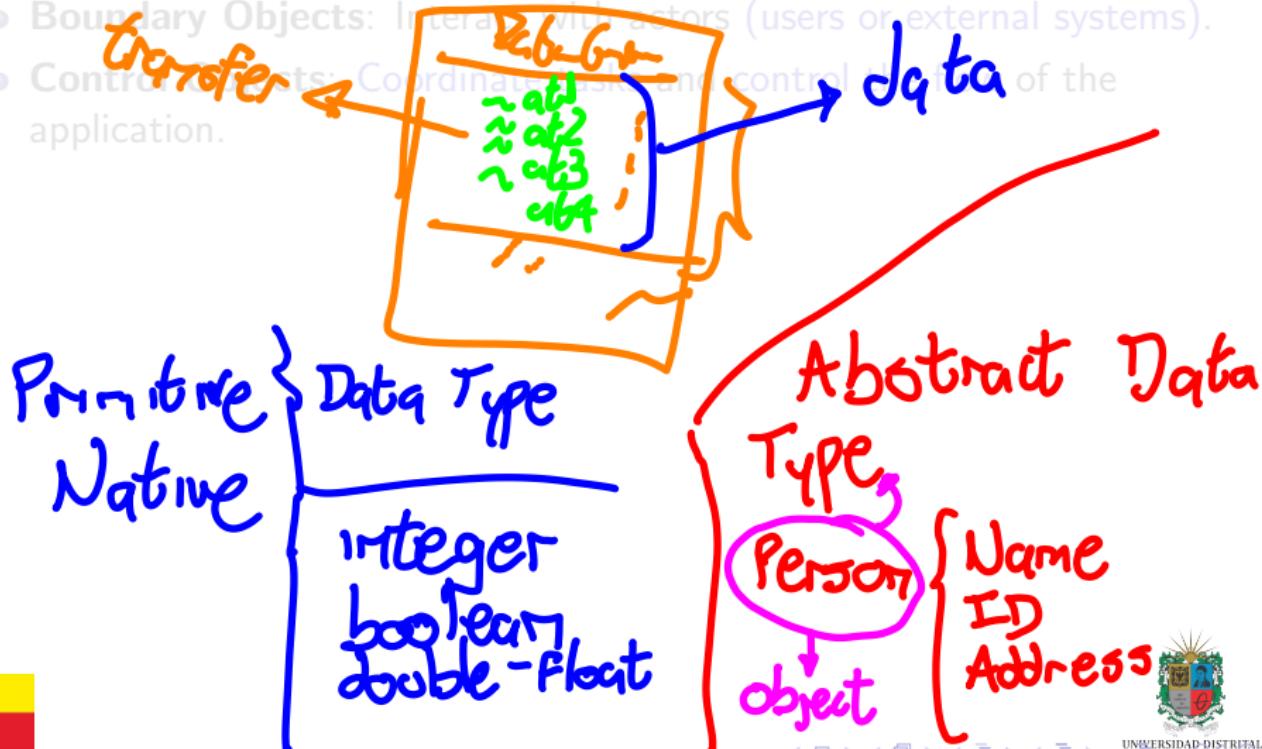
expectations



# Categories of Objects

- **Entity Objects:** Represent **information** and **data**.

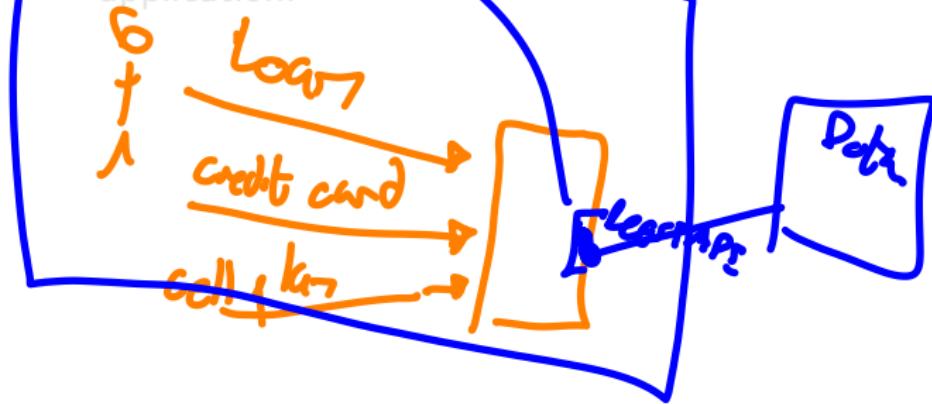
- **Boundary Objects:** Interact with actors (users or external systems).
- **Control Objects:** Coordinate logic and control flow of the application.



# 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.

logic  
most of the objects



# 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.



# 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.

old



solve problems

Findar  
información



# Outline

1 Creating Models in Design

2 Evolution of Programming Languages

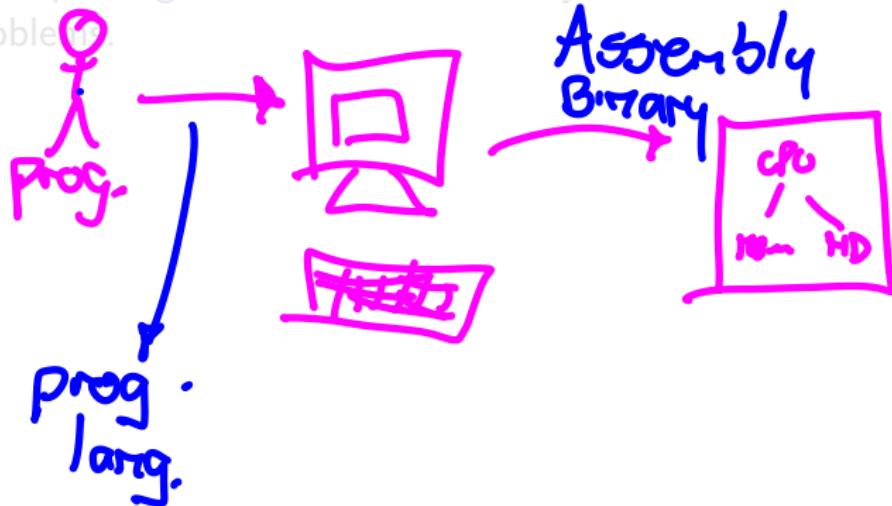
3 Four Design Principles

4 SOLID Principles



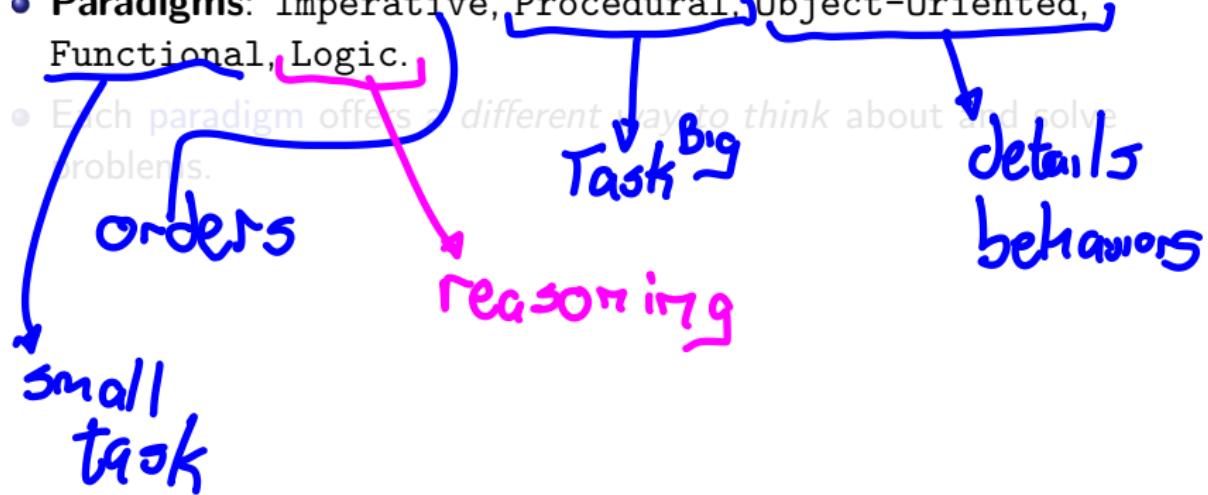
# Talk with Machines: Programming Paradigms

- 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.



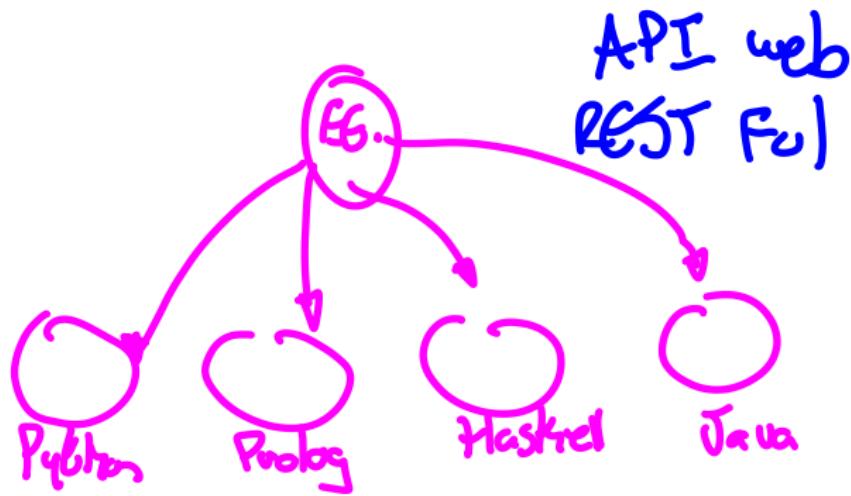
# Talk with Machines: Programming Paradigms

- **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.

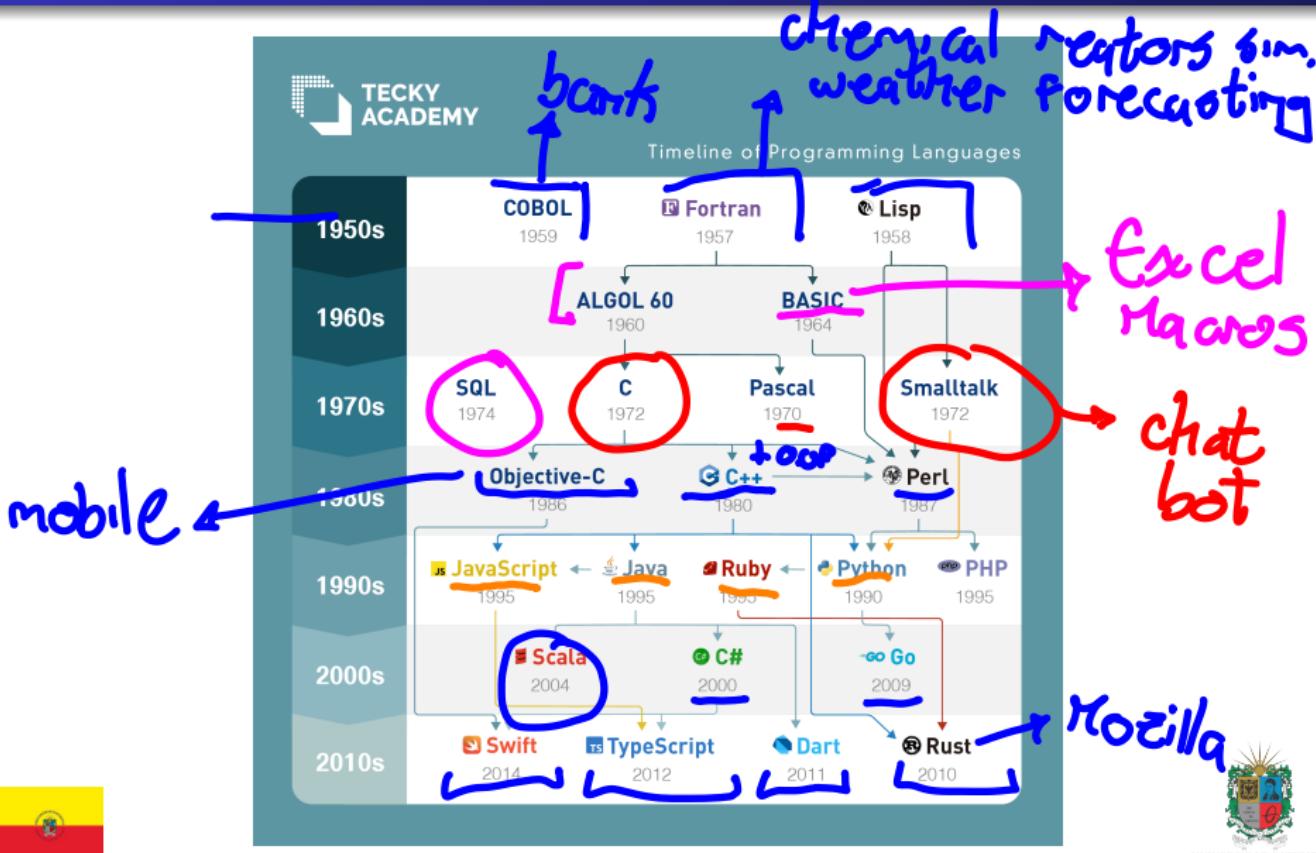


# Talk with Machines: Programming Paradigms

- **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



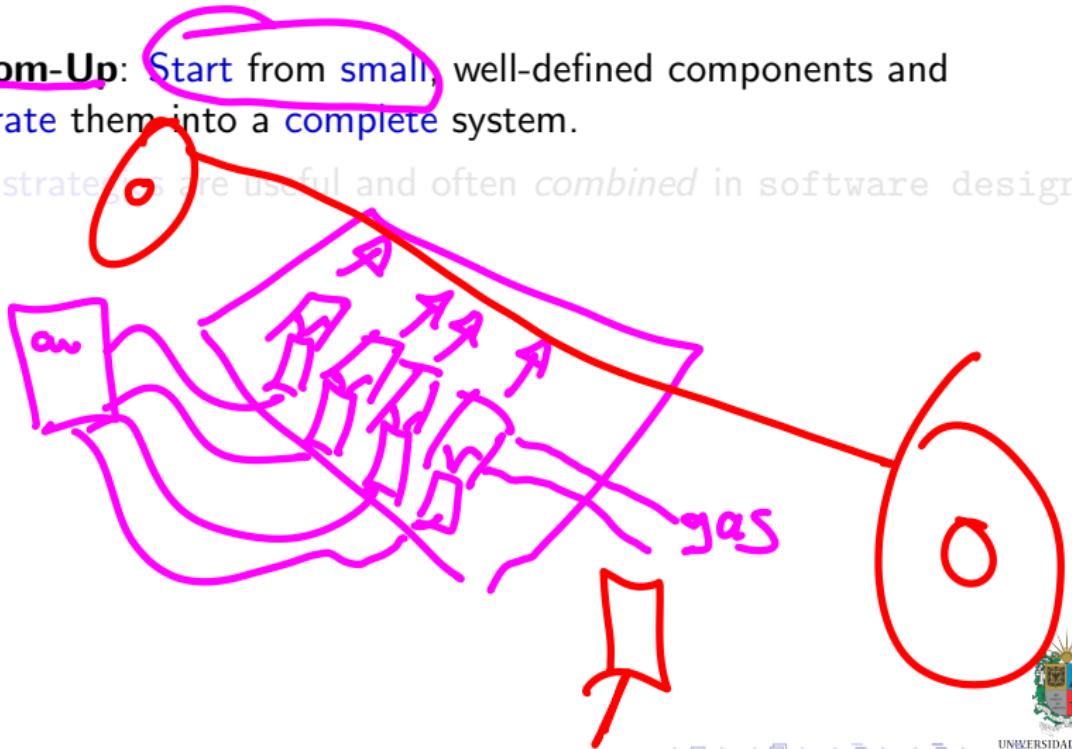
# Strategies to Solve Problems

- **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.



# Outline

1 Creating Models in Design

2 Evolution of Programming Languages

3 Four Design Principles

4 SOLID Principles



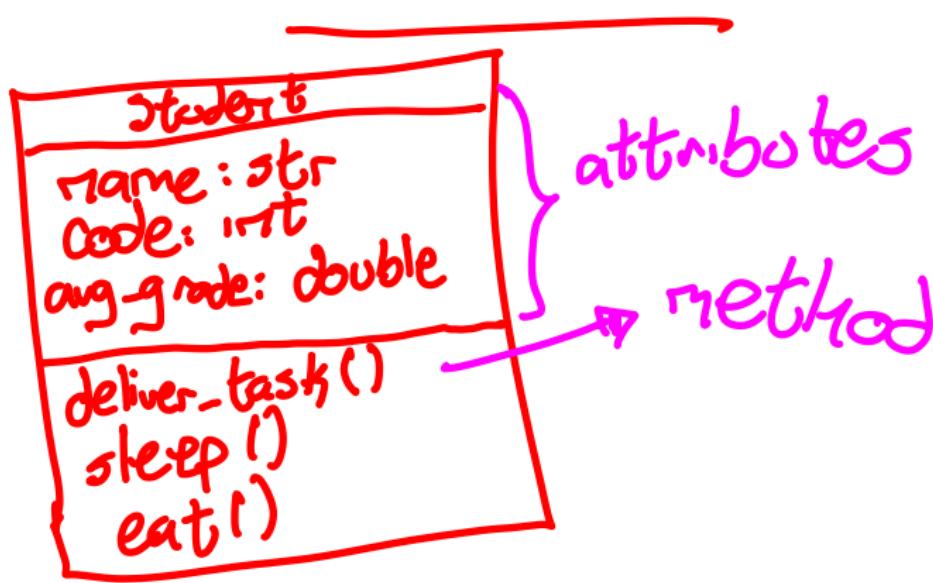
# Object-Oriented Design and Contracts

- **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.



# 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 communication by having clearly defined standards

Diagram illustrating the components of UML:

- Class Diagram
- Deployment Diagram
- Activity Diagram
- Use Case Diagram

A large bracket on the right side groups "Deployment Diagram", "Activity Diagram", and "Use Case Diagram" under the heading "standard".



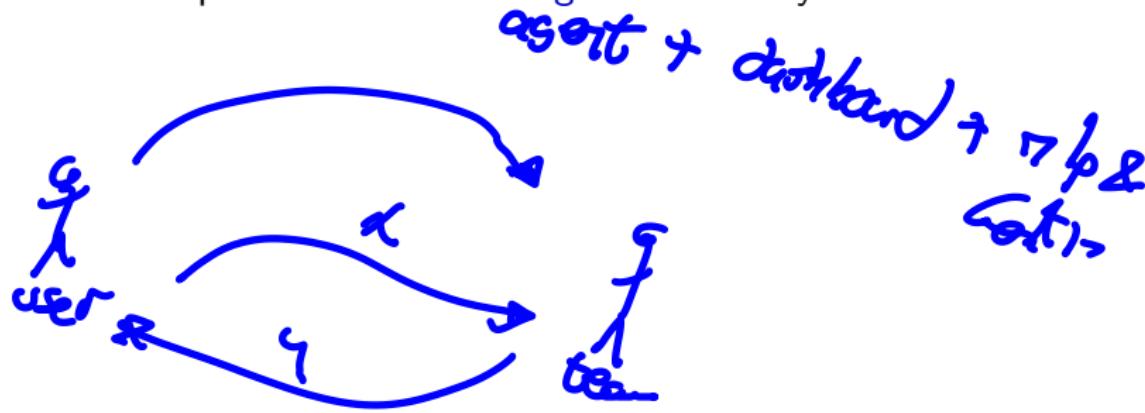
# UML Diagrams

- 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.
- oop*      *calls*
- Requirements*



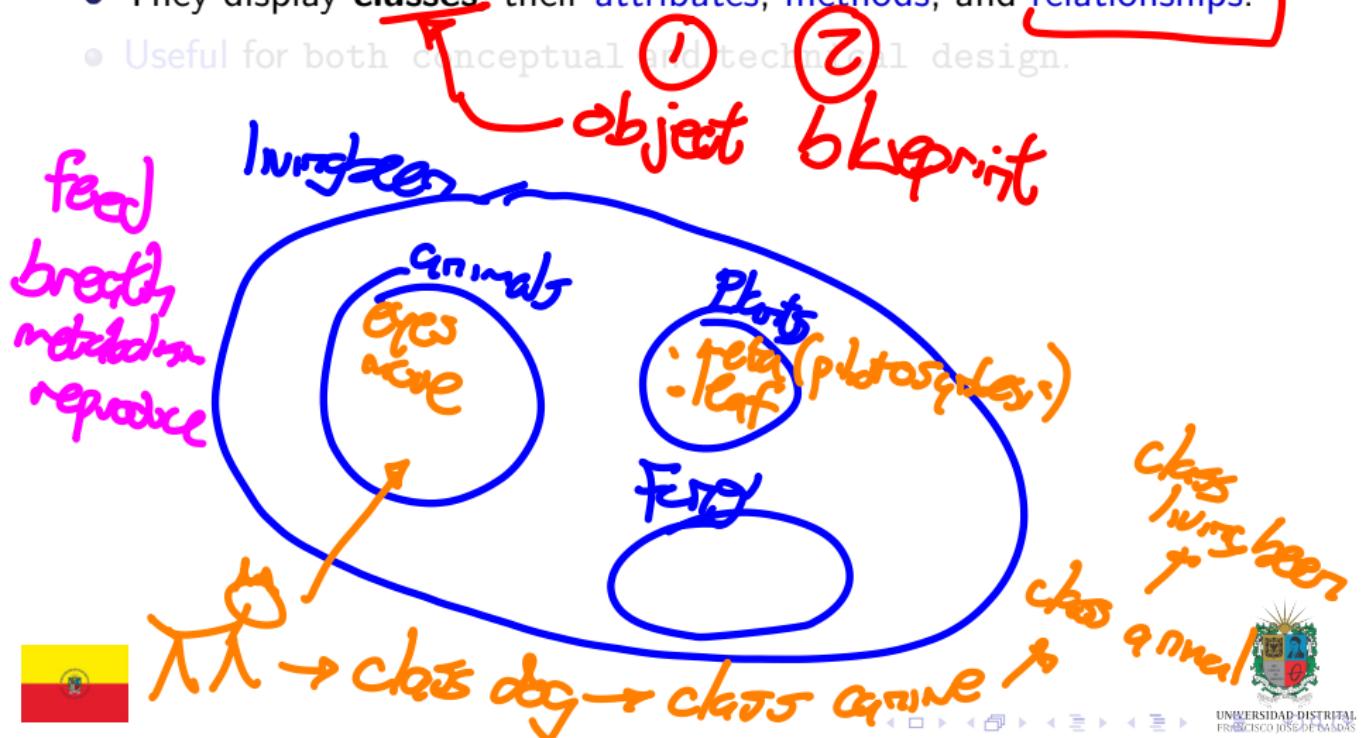
# UML Diagrams

- **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.



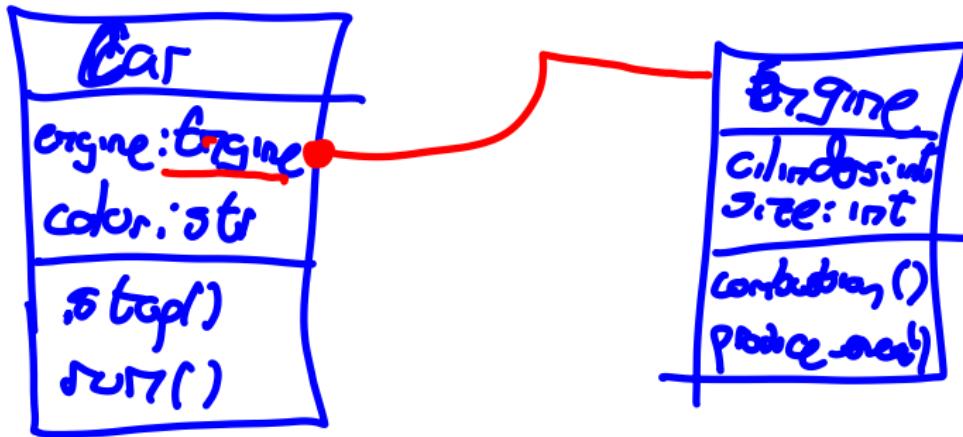
# Class Diagrams

- 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.



# 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*.

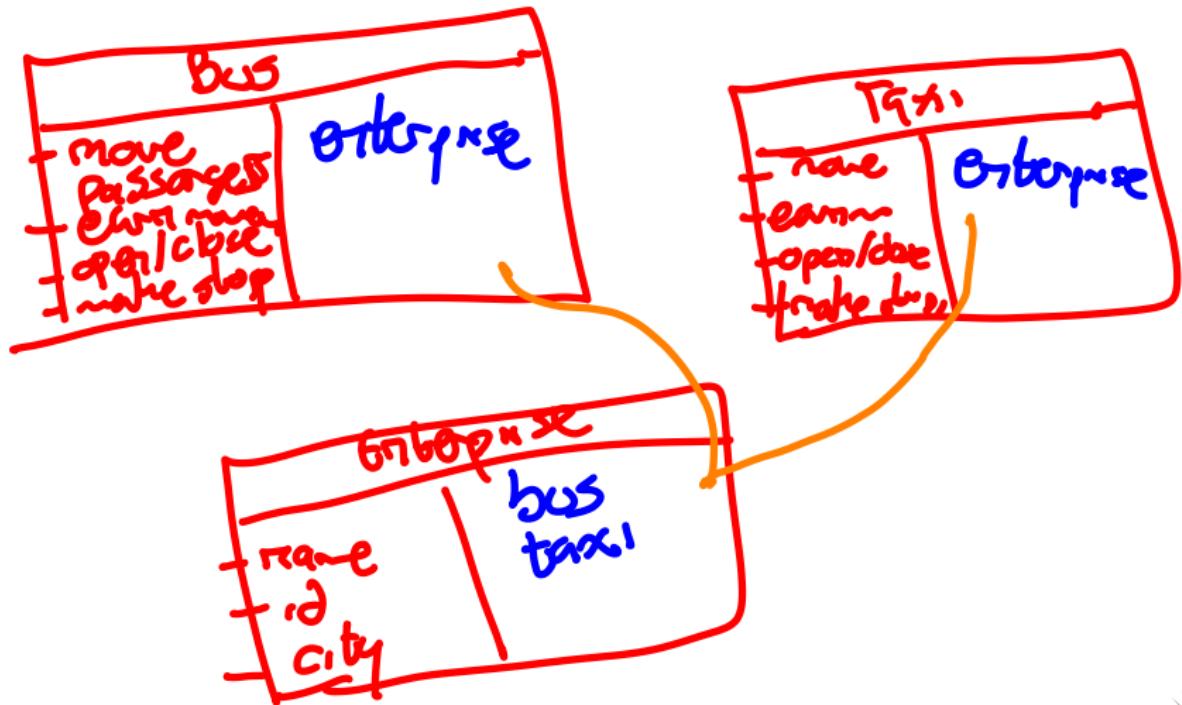


# Abstraction

- **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).

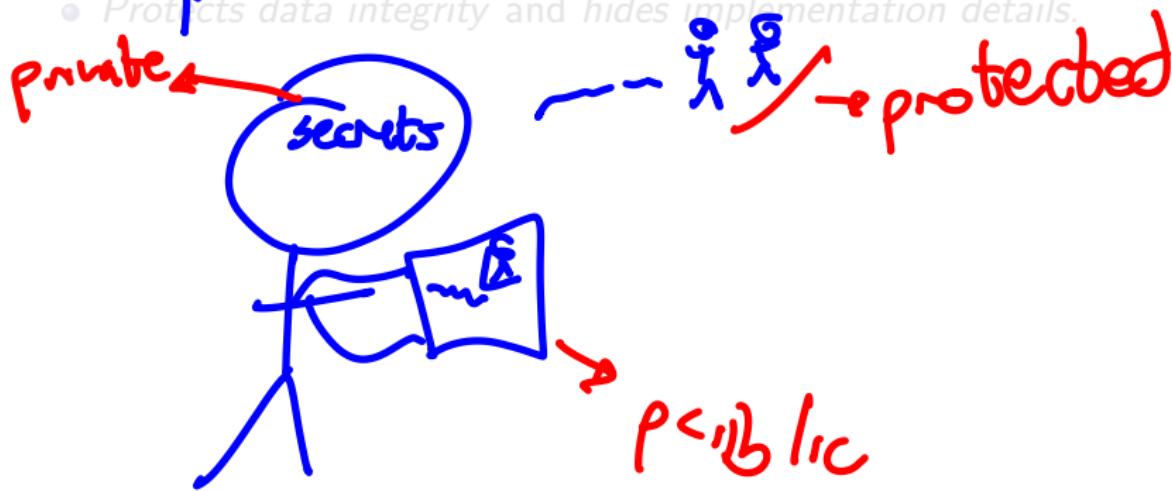
Hide implementation details.  
Protects data integrity and hides implementation details.

Hide implementation details.  
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.



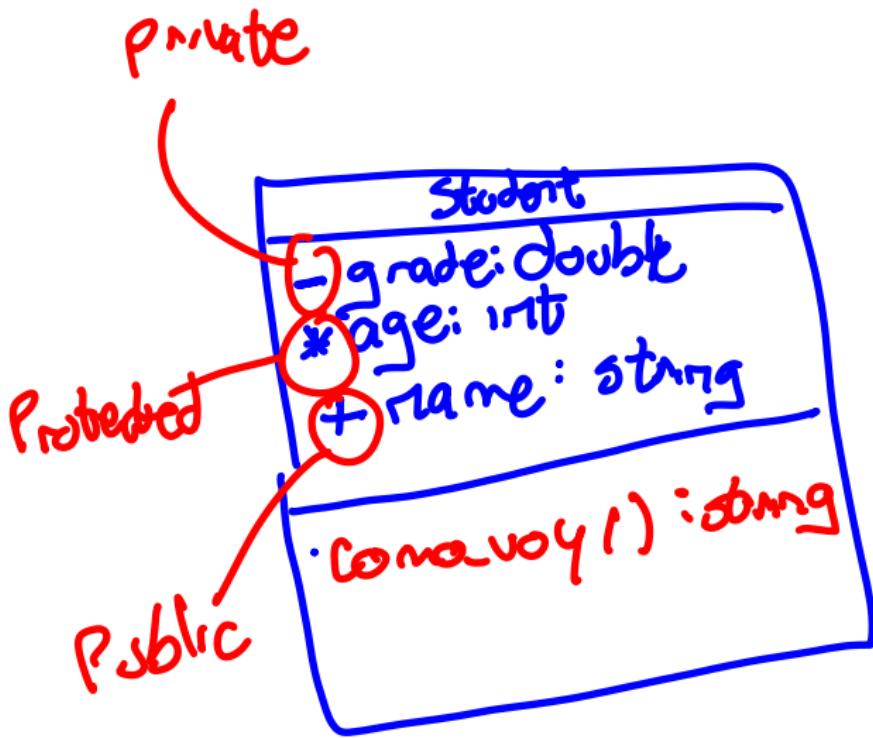
# Encapsulation

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

```
public class Student {  
    private double grade;  
    protected int age;  
  
    public String cond-way() {  
        if (grade >= 3.0) {  
            return "Bien";  
        } else if (grade >= 2.0) {  
            return "Aprobado";  
        } else return "Alta vacante";  
    }  
}
```



# Encapsulation & UML



# Black Box Thinking

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

Demo time!



# Black Box Thinking

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

## Demo time!



# Data Integrity: Getters and Setters

- **Getters** and **Setters** are methods to access and modify object attributes.
- They help maintain data integrity by controlling how attributes are accessed and modified.
- Use them to enforce validation rules and business logic.

## Demo time!



# Data Integrity: Getters and Setters

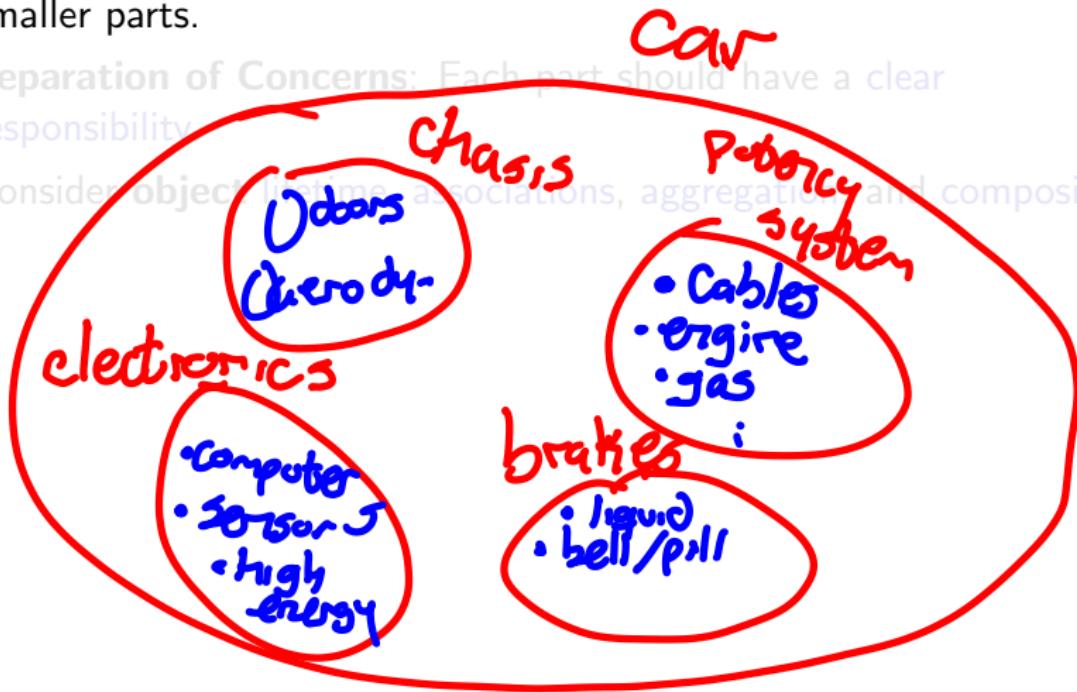
- **Getters** and **Setters** are methods to access and modify object attributes.
- They help maintain **data integrity** by controlling how attributes are accessed and modified.
- Use them to enforce **validation rules** and **business logic**.

## Demo time!



# Decomposition

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

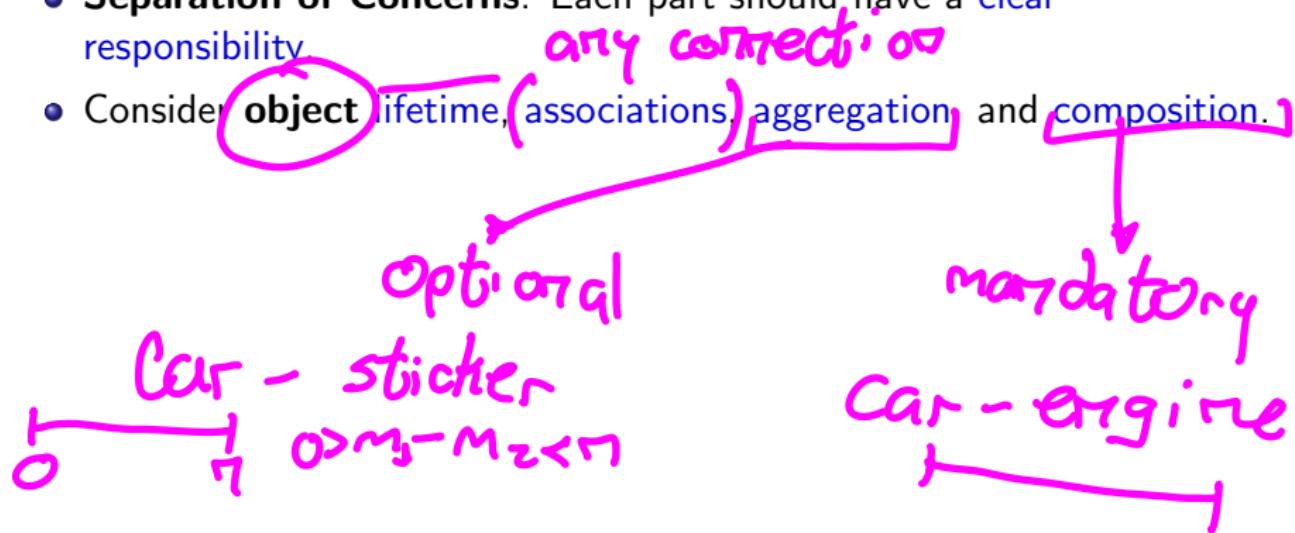
*object*

*role*

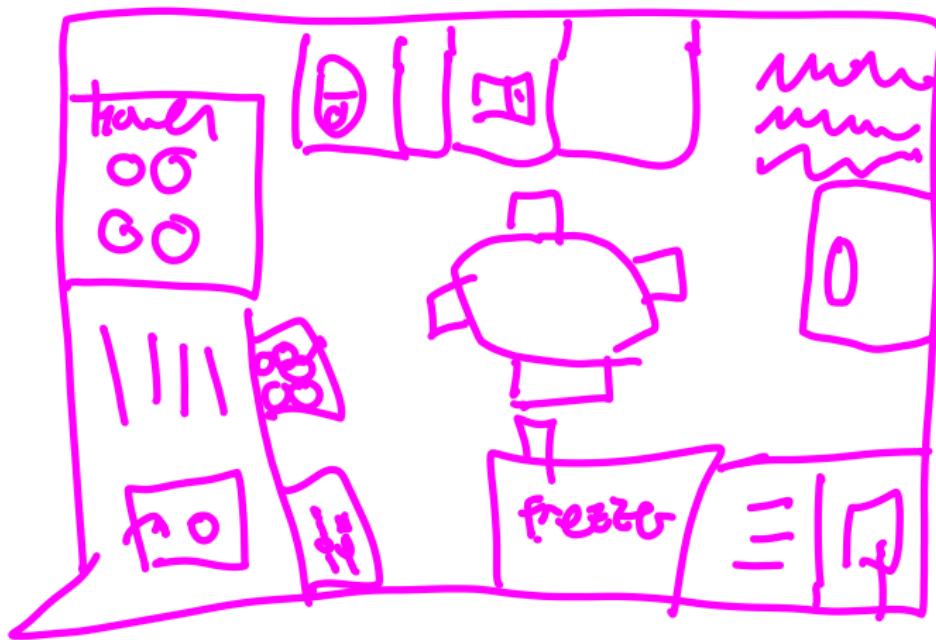


# Decomposition

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

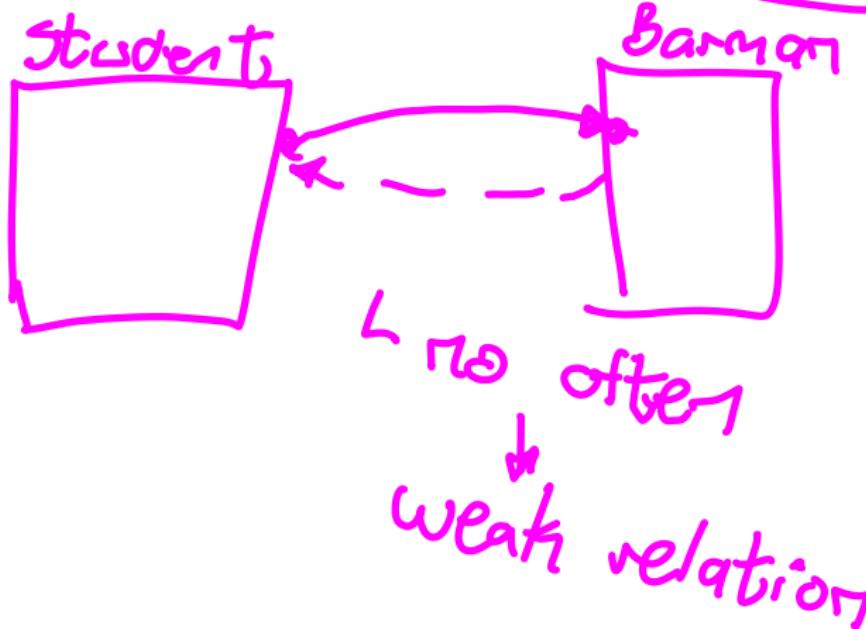


# Decomposition Example: Kitchen in a House



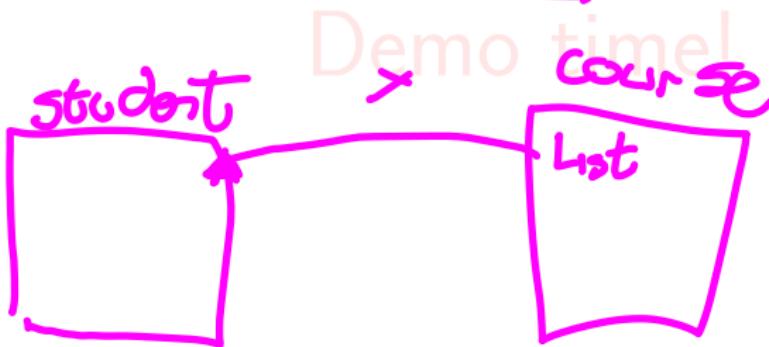
# Association

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



# Aggregation

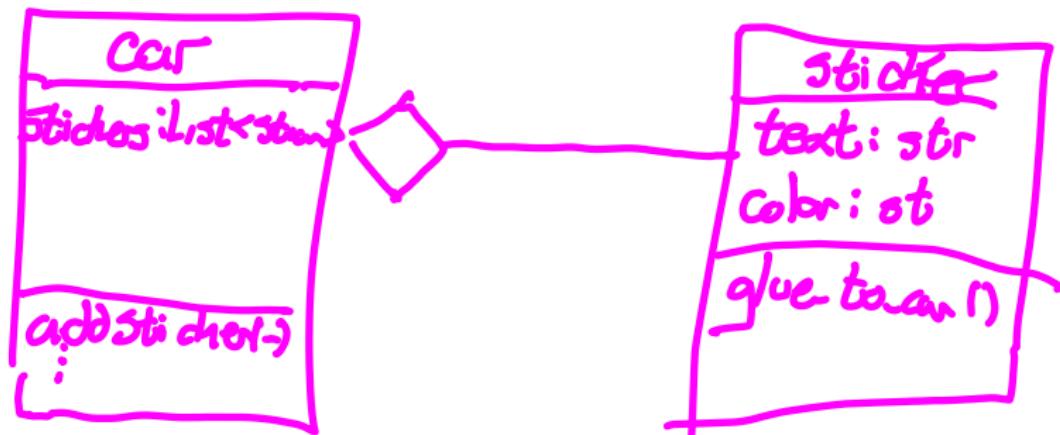
An **aggregation** is a **whole-part relationship** where one class is a **part** of another class, but **can exist independently**.



# Aggregation

An **aggregation** is a **whole-part relationship** where one class is a **part** of another class, but can exist independently \*

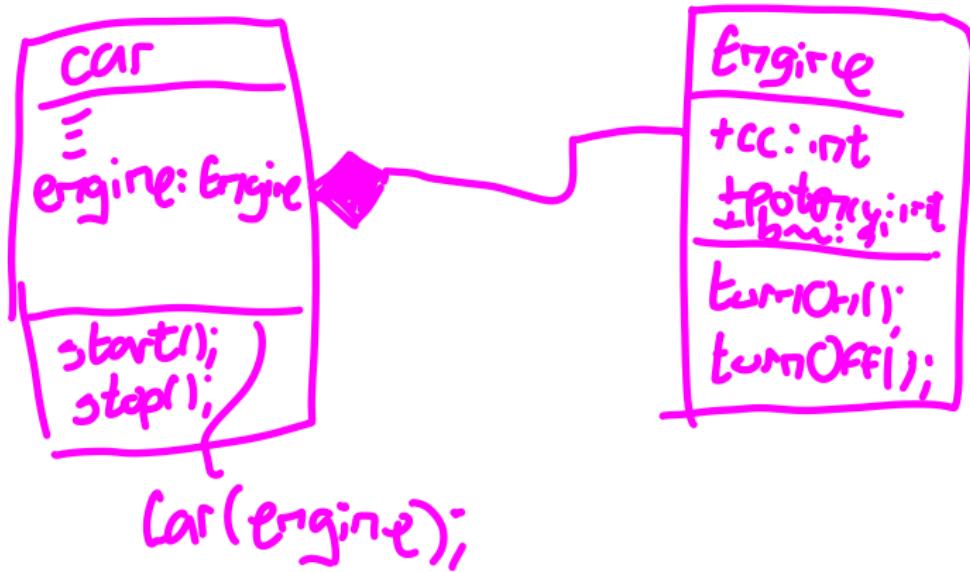
## Demo time!



# Composition

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

Demo time!



# Composition

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

## Demo time!



# 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: Subclasses 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**, and **abstract classes**.

- Polymorphism: Objects can be treated as instances of their parent class.

- Types of inheritance: single, multiple, interface-based.

/ abstract

/ concrete →

implemented logic

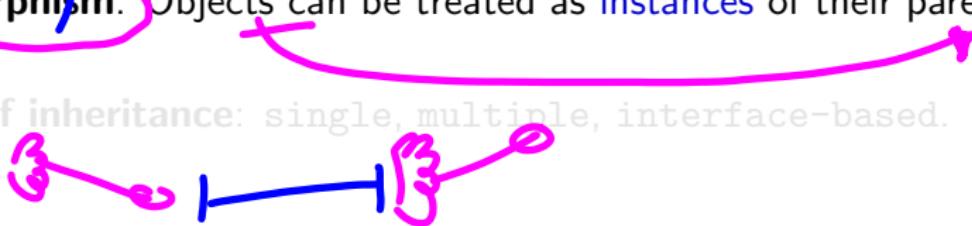


Transfer



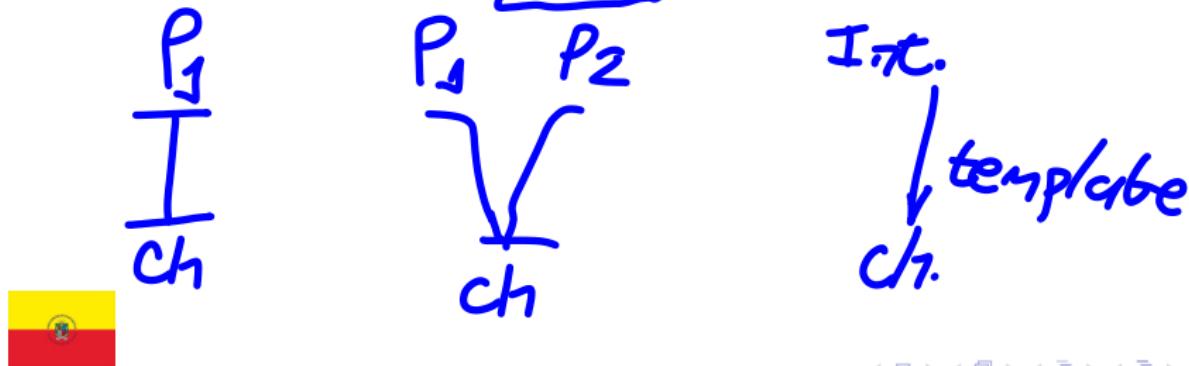
# Generalization

- **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.
- data + behaviour*

## Demo time!



# Inheritance

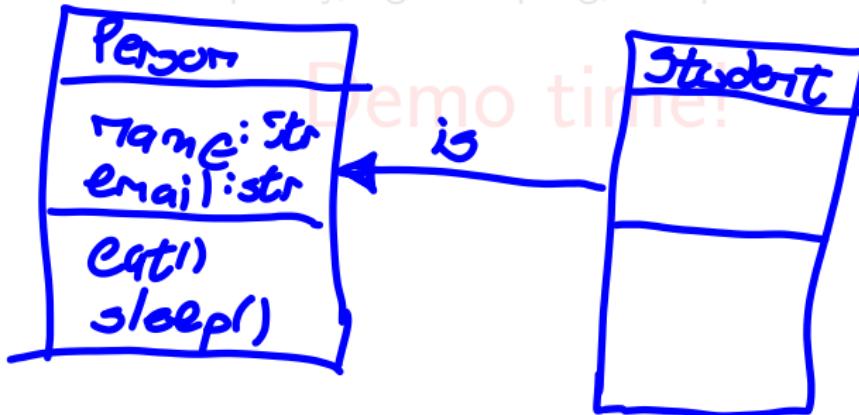
- **Inheritance** allows a class to inherit properties and methods from another class.

- **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.



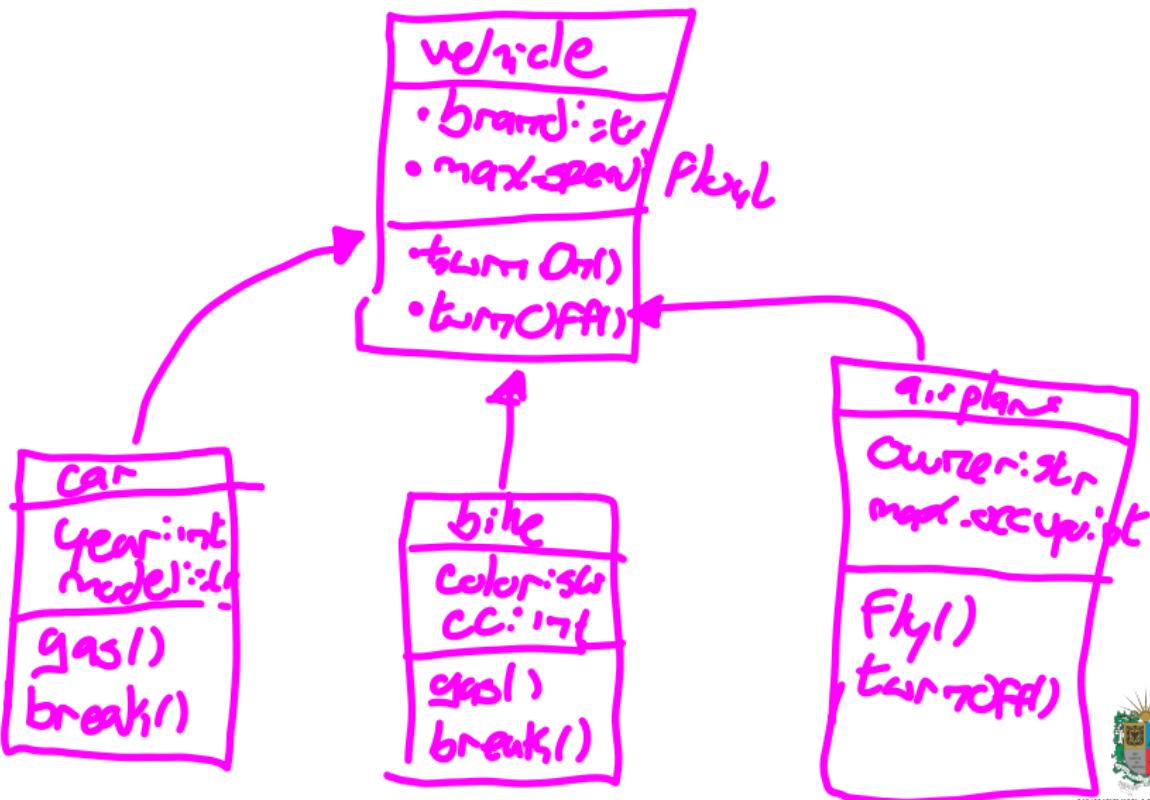
# Inheritance

- **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**.
- by inheritance*

## Demo time!



# 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.

## Demo time!



# 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.

## Demo time!



# 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.

## Demo time!



# 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.

## Demo time!



# 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.

## Demo time!



# Polymorphism by Overloading

- **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.

## Demo time!



# Polymorphism by Overloading

- **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.

## Demo time!



# Outline

1 Creating Models in Design

2 Evolution of Programming Languages

3 Four Design Principles

4 SOLID Principles



# SOLID Principles

- **Single Responsibility Principle (SRP)**: A class should have only one reason to change.
- **Open/Closed Principle (OCP)**: A class should be open for extension, but closed for modification.
- **Liskov Substitution Principle (LSP)**: Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- **Interface Segregation Principle (ISP)**: A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.
- **Dependency Inversion Principle (DIP)**: High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.



# SOLID Principles

- **Single Responsibility Principle (SRP)**: A class should have only one reason to change.
- **Open/Closed Principle (OCP)**: A class should be **open** for extension, but **closed** for modification.
- **Liskov Substitution Principle (LSP)**: Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- **Interface Segregation Principle (ISP)**: A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.
- **Dependency Inversion Principle (DIP)**: High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.



# SOLID Principles

- **Single Responsibility Principle (SRP)**: A class should have only one reason to change.
- **Open/Closed Principle (OCP)**: A class should be **open** for extension, but **closed** for modification.
- **Liskov Substitution Principle (LSP)**: Objects in a program should be replaceable with instances of their **subtypes** without altering the correctness of that program.
- **Interface Segregation Principle (ISP)**: A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.
- **Dependency Inversion Principle (DIP)**: High-level modules should not depend on low-level modules. Both should **depend on abstractions**. Abstractions should not depend on details. Details should depend on abstractions.



# SOLID Principles

- **Single Responsibility Principle (SRP)**: A class should have only one reason to change.
- **Open/Closed Principle (OCP)**: A class should be **open** for extension, but **closed** for modification.
- **Liskov Substitution Principle (LSP)**: Objects in a program should be replaceable with instances of their **subtypes** without altering the correctness of that program.
- **Interface Segregation Principle (ISP)**: A client **should never be forced** to implement an interface that it doesn't use or clients **shouldn't be forced** to depend on methods they do not use.
- **Dependency Inversion Principle (DIP)**: High-level modules should not depend on low-level modules. Both should **depend on abstractions**. Abstractions should not depend on details. Details **should depend on abstractions**.



# SOLID Principles

- **Single Responsibility Principle (SRP)**: A class should have only one reason to change.
- **Open/Closed Principle (OCP)**: A class should be **open** for extension, but **closed** for modification.
- **Liskov Substitution Principle (LSP)**: Objects in a program should be replaceable with instances of their **subtypes** without altering the correctness of that program.
- **Interface Segregation Principle (ISP)**: A client **should never be forced** to implement an interface that it doesn't use or clients **shouldn't be forced** to depend on methods they do not use.
- **Dependency Inversion Principle (DIP)**: High-level modules should not depend on low-level modules. Both should **depend on abstractions**. Abstractions should not depend on details. **Details should depend on abstractions**.



# Good Practices

- **Composition over Inheritance:** Inheritance should be used **only when there is a clear relationship** between the base class and the derived class. In other cases, **composition** should be used. Inheritance is a powerful tool, but it is not always the best tool for the job. Inheritance is a way to achieve polymorphism, but it is **not the only way to achieve polymorphism**.
- **Code to Interfaces, not Implementations:** This principle is about designing your classes so that they depend on interfaces rather than concrete classes.

## Demo time!



# Good Practices

- **Composition over Inheritance:** Inheritance should be used **only when there is a clear relationship** between the base class and the derived class. In other cases, **composition** should be used. Inheritance is a powerful tool, but it is not always the best tool for the job. Inheritance is a way to achieve polymorphism, but it is **not the only way to achieve polymorphism**.
- **Code to Interfaces, not Implementations:** This principle is about designing your classes so that they **depend on interfaces** rather than concrete classes.

## Demo time!



# Outline

1 Creating Models in Design

2 Evolution of Programming Languages

3 Four Design Principles

4 SOLID Principles



# Thanks!

## Questions?



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

