Object-Oriented Modeling

Object-Oriented Programming

Author: Eng. Carlos Andrés Sierra, M.Sc.
casierrav@unal.edu.co

Lecturer
Computer Engineering Program
School of Engineering
Universidad Nacional de Colombia

2025-II

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.

draws

increase understanding

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`.

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.

needs

1. business understanding
   1. impact
      add value
   2. increase clients
2. technical implications

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.

right

boundaries

requirements

redes '97

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*.

actors/roles
grants
dependencies

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.

easy design

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.

- Business Process → BPMN
  ↓
  Flowchart
- User Stories
- Mockups
- CRC cards

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.

engineering

- infrastructure
- objects in code

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.

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.

int double bool

native/primitive

Abstract Data Type

```
class Student:
    name: str
    age: int
    code: int
    papa: Float
```

var1 = Student("pepito", 20, 203040, 3.7)

var1.papa

var1.name

var2 = Stu(...)

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
+
validations

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.

needs → expectations

class diagram

~

// ~

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.

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.

macros

functions

AI.

common task

represent

propos.

```
int suma(int par1, int par2){
    return par1 + par2;
}
```

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

~1930 Assemble

Noam Chomsky

P.O.O.

1991 Internet → web

mathematical

Excel Macros

Chat bot

Bioinformatics

1993

Data Science

mobile

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`.

```
c = Calculator()
print(c.sum(2, 2))
print(c.mult(4,5))
print(c.div(6,3))
```

Outline

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


```
class Calculator:
    def sum(n1, n2):
        return n1 + n2

    def mult(n1: int, n2: int):
        acum = 0
        for i in range(n1):
            acum += n2
        return acum

    def div(n1, n2):
        if n2 != 0:
            return n1/n2
        else:
            return "error"
```

0, 1, 2, 3, ..., n1-1

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.

set

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.

actions

working as expected

no fail

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.

draws → O.O.P.

call an app

- Draw.io
- Lucid Chart

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.

communication between object

system
→ elem: classes
→ relations

Functional 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`.

structure

guide

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*.

Person

Student
- avg-grade
- code

Sportman
- reflex-time
- speed

Pilot
- reflex_time
- Fly-hours
- license

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*.

1. User-centered design

Abstraction & CRC Cards

Chaza	
• sell products • has cash money • has inventory	seller

Class
Responsibilities
Collaborators

seller	
• mera inventory • clean zone • handle client	Chaza

Encapsulation

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

hide

Encapsulation

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

(friends & family)

all (name)

only me

birthday
age
address

Student
- name → public
- code → protected?
- email → public
- avg-grade → protected
- beers-max → protected
- fears → private

Encapsulation

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

```
Student:
  avg-grade: float → private
  def como_va():
      if avg-grade >= 4.0:
          print("todo melo")
      elif avg-grade >= 3.0:
          print("aguanta")
      else:
          print("dam doble")
```

no expose

Encapsulation & UML

+ public
* protected
- private

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!

define

```
private fears;
public get_fears():
    return fears

public set_fears(new_fears):
    fears = new_fears

public get_fears():
    if role == 'admin':
        return fears
```

```
class NameClass:
    def __init__(self):
        self.atti = 5
        self.atti2 = None
```

object

var = NameClass()

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!

```
Class Student:
    def __init__(self):
        self.age = None

    def set_age(new_age):
        if new_age > 0:
            self.age = new_age
        else: raise
```

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

- **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

- **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.

Demo time!

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**: 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`.

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.

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.

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.

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.

UNIVERSIDAD NACIONAL DE COLOMBIA

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?