Design Patterns(SE-408)

Course Teacher
Assistant Professor
Engr. Mustafa Latif

Topics Cover

TOPICS
Introduction to Object-Oriented Software Development, The Object-Oriented Paradigm
UML-The Unified Modeling Language
The Principles and Strategies of Design Patterns
Introduction to Design Patterns, The Singleton Pattern
The Factory Method Pattern, The Abstract Factory Method Pattern
The Builder Pattern
The Facade Pattern,
The Adapter Pattern,
The Decorator Pattern,
The Bridge Pattern, Midterm
The Strategy Pattern
The Observer Pattern
The Template Method Pattern

Reading Materials

- "Dive Into DESIGN PATTERNS" Alexander Shvets
- "Design Patterns Explained: A New Perspective on Object-Oriented Design", Alan Shalloway and James R. Trott, Addison-Wesley Professional, 2nd Edition
- "Java Design Patterns: A tour of 23 gang of four design patterns in Java", Vaskaran Sarcar , Apress, 2016

3

Marks Distribution

- Assignment(15 Marks) in 12th week
- Test(05 Marks) in 5th week
- Mid-term(20 Marks) in 8th week
- Final Exam (60 Marks) in 16th week

4

Software Required

- JDK (java development kit)
- IntelliJ IDEA (java IDE)
- StarUML

5

- Introduction to Object-Oriented Software Development

6

Software Engineering

- **Software**
- **Types of Software Systems**
- **Successful Software System**
  - Deliver required functionality, adaptable, response time, performance (less use of resources such as memory and processor time), usable; acceptable for the type of the user it's built for, reliable, secure, safe, ..etc
- **Software Engineering**
  - Software engineering is an engineering discipline that's applied to the development of software in a *systematic* approach (called a software process).
- **Computer Science Vs Software Engineering**
- **Job of a software engineer** (Dealing with users, Dealing with technical people, Dealing with management**)**

7

Software Process

- A software process (also knows as software **methodology**) is a set of related activities that leads to the production of the software. These activities may involve the development of the software from the scratch, or, modifying an existing system.
  - **Software specification** (or requirements engineering): Define the main functionalities of the software and the constrains around them.
  - S**oftware design and implementation**: The software is to be designed and programmed.
  - **Software verification and validation**: The software must conforms to it's specification and meets the customer needs.
  - **Software evolution** (software maintenance): The software is being modified to meet customer and market requirements changes.

8

Software Process Models

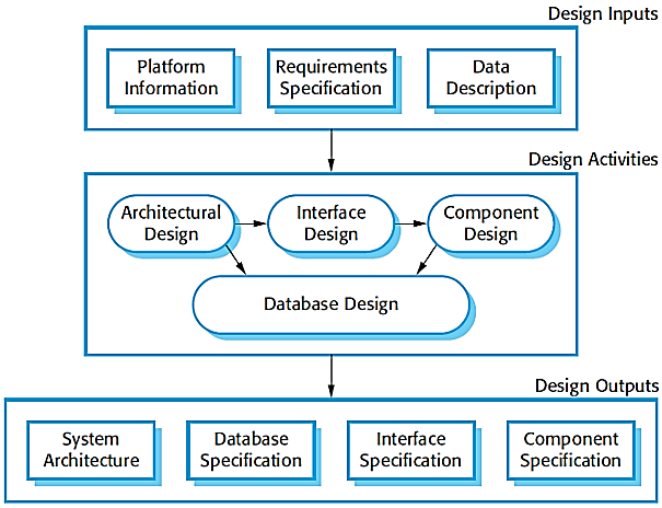
- A software process model is a simplified representation of a software process. Each model represents a process from a specific perspective.
- **Waterfall**
  - It's useful when the requirements are clear, or following a very structured process as in critical systems which needs a detailed, precise, and accurate documents describes the system to be produced.
- **Prototype**
  - This is very useful when requirements aren't clear, and the interactions with the customer and experimenting an initial version of the software results in high satisfaction and a clearance of what to be implemented.
- **Incremental & Iterative**
  - They're suited for large projects, less expensive to the change of requirements as they support customer interactions with each increment. Initial versions of the software are produced early, which facilitates customer evaluation and feedback.
- **Spiral**
  - It's good for high risky or large projects where the requirements are ambiguous. The risks might be due to cost, schedule, performance, user interfaces, etc.
- **Agile**
  - It suits small-medium size project, with rapidly changes in the requirements as customer is involved during each phase.

Requirements Engineering

- **Software specification or requirements engineering** is the process of understanding and defining what services are required and identifying the constraints on these services.
- **Requirements engineering processes** ensures your software will meet the user expectations, and ending up with a high quality software.
- It's a critical stage of the software process as **errors at this stage** will reflect later on the next stages, which definitely will cause you a **higher costs**.
- At the end of this stage, a **requirements document** that specifies the requirements will be produced and validated with the stockholders.

Software Design And Implementation

- Software design and implementation is the stage in the software engineering at which an executable software system is developed



11

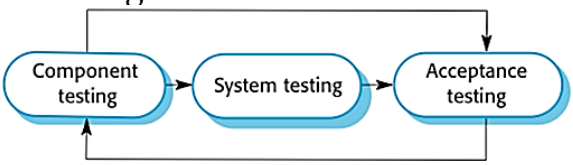
Activities can vary depending on the type of the system needs to be developed. Four main activities that may be part of the design process for **information systems**, and they are:

- **Architectural design:**
  - Defines the overall structure of the system, the main components, their relationships.
- **Interface design:**
  - **D**efines the interfaces between these components.
  - The interface specification must be clear, so that component can be used without having to know it's implemented.
  - Once the interface specification are agreed, the components can be designed and developed concurrently.
- **Component design:**
  - Take each component and design how it will operate.
- **Database design:**
  - The system data structures are designed and their representation in a database is defined.

12

Software Verification And Validation

- Software verification and validation (V&V) is intended to show that a system both **conforms to its specification** and that it meets the **expectations of the customer**.
- Testing has three main stages:



  - Development (or component) testing
  - System testing
  - Acceptance testing

13

Software Maintenance

- **Software Maintenance** is the process of modifying a software product after it has been delivered to the customer.

- Need for Maintenance –

  Software Maintenance must be performed in order to:
  - Correct faults.
  - Improve performance.
  - Improve the design.
  - Implement enhancements.
  - Interface with other systems.
  - Accommodate programs so that different hardware, software, system features, and telecommunications facilities can be used.

14

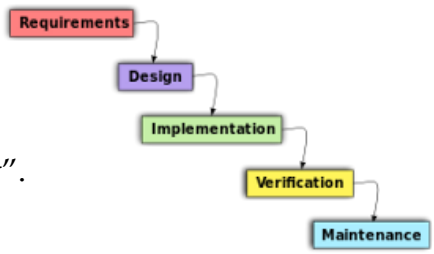
The Concept Of Object-Orientation

- Object-orientation is what's referred to as a **programming paradigm**. It's **not a language itself** but a set of concepts that is supported by many languages.
- Everything done in these languages is object-oriented, i.e. we are oriented or focused around **objects**.
- In an object-oriented language, there is not a **single large program**, instead of this it is splits into **self contained objects**, each object representing a different part of the application.
- Each object contains its **own data and its own logic**, and they communicate between themselves.
- Each object represent things like employees, images, bank accounts, spaceships, asteroids, video segment, audio files, or whatever exists in your program.

15

Object-Oriented Analysis And Design (OOAD)

- It's a **structured method for analyzing, designing** a system by applying the **object-orientated concepts,** and develop a set of graphical system models during the development **life cycle of the software**.
- The software life cycle is typically divided into various stages.
  - The earliest stages of this process are analysis (**requirements**) and **design**.
  - The distinction between analysis and design is often described as "**what vs how**".



16

Object-Oriented Analysis

- In the object-oriented analysis, we …
  - **Elicit** *requirements:* Define what does the software need to do, and what's the problem the software trying to solve.
  - **Specify** *requirements*: Describe the requirements, usually, using use cases (and scenarios) or user stories.
  - **Conceptual** *model*: Identify the important objects, refine them, and define their relationships and behavior and draw them in a simple diagram.

17

Object-Oriented Design

- In the object-oriented design, we …
  - **Describe the classes** and their relationships using class diagram.
  - **Describe the interaction** between the objects using sequence diagram.
  - **Apply** software design principles and design patterns.

18

Unified Modelling Language (UML)

- The unified modeling language become the standard modeling language for object-oriented modeling.
- **Use case diagram**: Shows the interaction between a system and it's environment (users or systems) within a particular situation.
- **Class diagram**: Shows the different objects, their relationship, their behaviors, and attributes.
- **Sequence diagram**: Shows the interactions between the different objects in the system.
- **State machine diagram**: Models the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.

19

Conceptual Model

- After having some use cases or user stories, the next thing we can do is to create a conceptual model of our system.

Steps:

- 1. Identifying Objects
  - Through use cases, user stories, and any other written requirements identify objects.
  - Objects will be in form of nouns. Those are the candidate objects, some of them will be actual objects in the system, and the rest won't, as you'll see later.

Use Case Scenario: Customer confirms items in shopping cart. Customer provides payment and address to process sale. System validates payment and responds by confirming order, and provides order number that Customer can use to check on order status. System will send Customer a copy of order details by email.

20

Conceptual Model

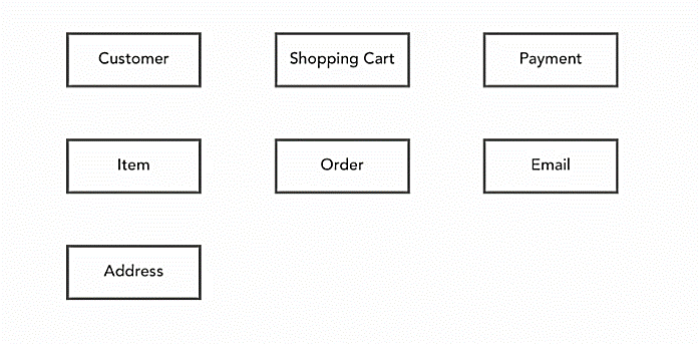
- 2. Refining Objects
  - After underlying on your candidate objects, you start refining them by
    - Remove any **duplicates.**
    - **Combine** some objects, or, even **splitting** some objects.

Customer	Order
Item	~~Order Number~~
Shopping Cart	~~Order Status~~
Payment	~~Order Details~~
Address	Email
~~Sale~~	~~System~~

  - You may identify an **attribute** as an object instead.
  - You may identify a **behavior** as an object instead.

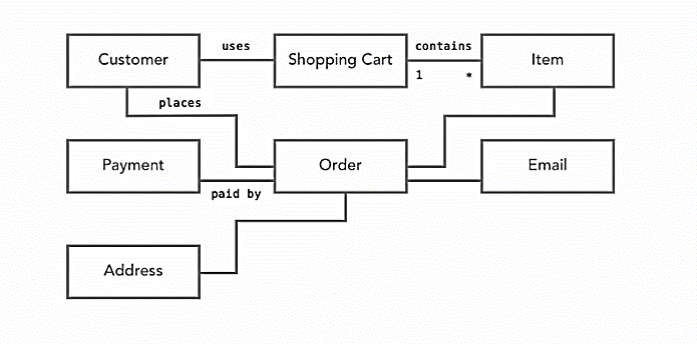
Conceptual Model

- 3. Drawing Objects
  - Just draw the conceptual model by box all objects.



Conceptual Model

- 3. Identifying Object Relationships
  - Drawing a line between objects, and writing the relationship verbs is enough to denote there is a relationship.



23

Conceptual Model

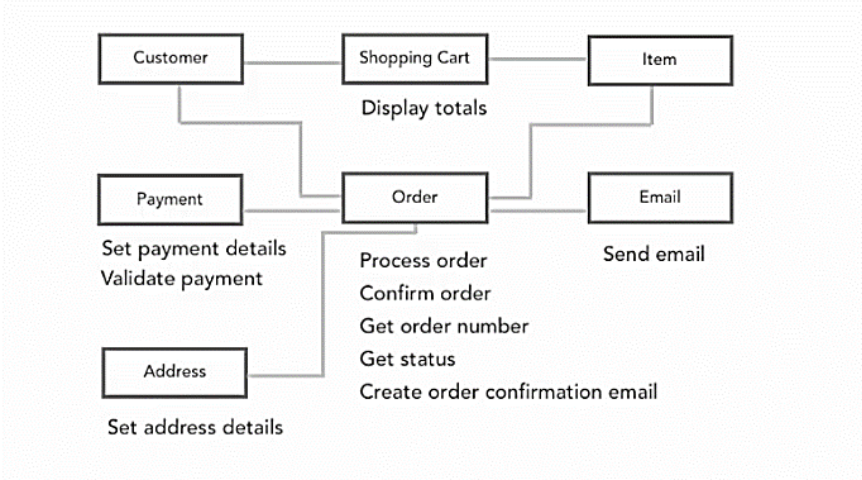
- 4. Identifying Object Behaviours
  - Behaviors are the things (verbs) the object can do, or, in other words, the **responsibilities** of an object, that will become the **methods** in our object class.
  - Pick responsibilities using the use case or a user story by looking for verbs and verb phrases.

Use Case Scenario: Customer verifies items in shopping cart. Customer provides payment and address to process sale. System validates payment and responds by confirming order, and provides order number that Customer can use to check on order status. System will send Customer a copy of order details by email.

Verify items	Confirm order
Provide payment and address	Provide order number
Process sale	Check order status
Validate payment	Send order details email

24

Conceptual Model



Conceptual Model

Remember:

- Whose Responsibility Is This?
  - What isn't always obvious is where these responsibilities belong, particularly if they affect different objects. It's because the use case describes what initiates a behavior, not necessarily who's responsible to perform that behavior.
  - So, even though it's the customer who wants to know the status of the order, it's the responsibility of the order to check it's status. The customer should ask the order object to report it's own status.
  - *When you ask whose responsibility is this? Always remember that an object should be responsible for itself.*

Conceptual Model

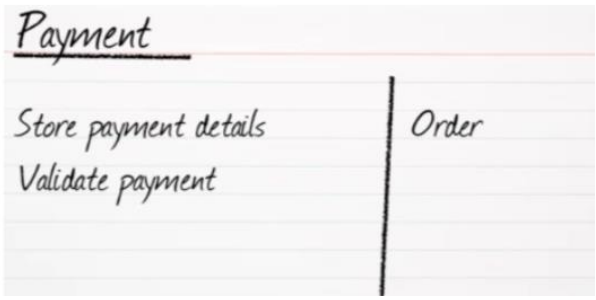
Caution:

- **The Generic Verbs**
  - Change the generic verb **provide** to set and get instead to make it clear.
- **The "System" Object**
  - It's common to see phrases like "system" validates payment or "system "will send the customer an email at use cases, and that can lead to people creating a system object and putting a huge amount of responsibilities in it.
  - Figure out which part of the system should be responsible for that behavior.

27

Conceptual Model

- Class Responsibility Collaborator (CRC) Cards
  - Class Responsibility Collaborator (CRC) is another technique for organizing the objects.
- You write every object on a piece of paper, with it's associating behaviors on a side, and the other side has the other objects (collaborators) that has a relationship with that object.

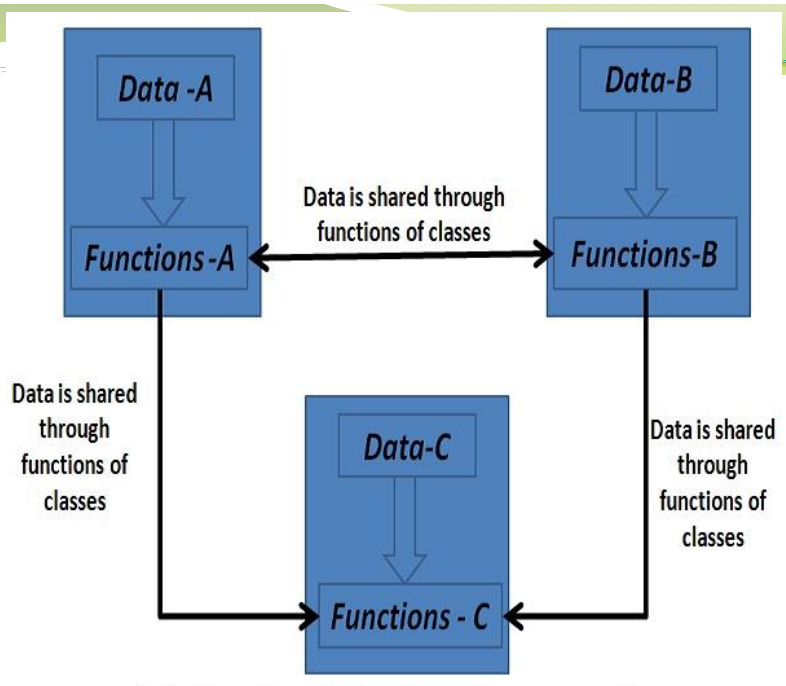


28

Programming Paradigms

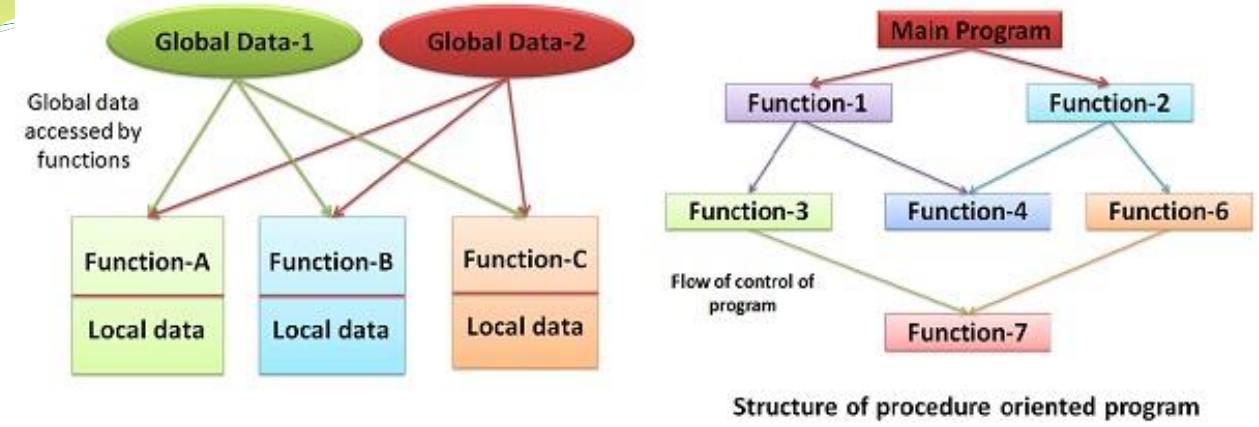
- Object-oriented Programming(Data-oriented)
- Procedural-oriented Programming (process-oriented)

29



Data flow in object-oriented programming

30



Structure of procedure oriented program

31

Assignment

- **Write a comprehensive report on the following topic (due date next week)**
  - **Architectural Style,**
  - **Architectural Patterns**
  - **and Design Patterns.**

32

The Object-Oriented Paradigm

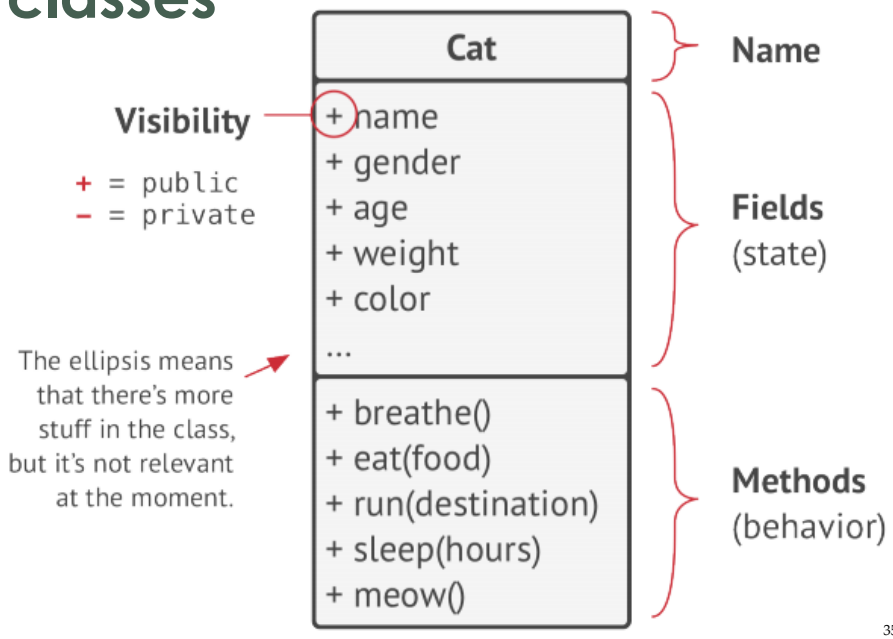
33

Basics of OOP

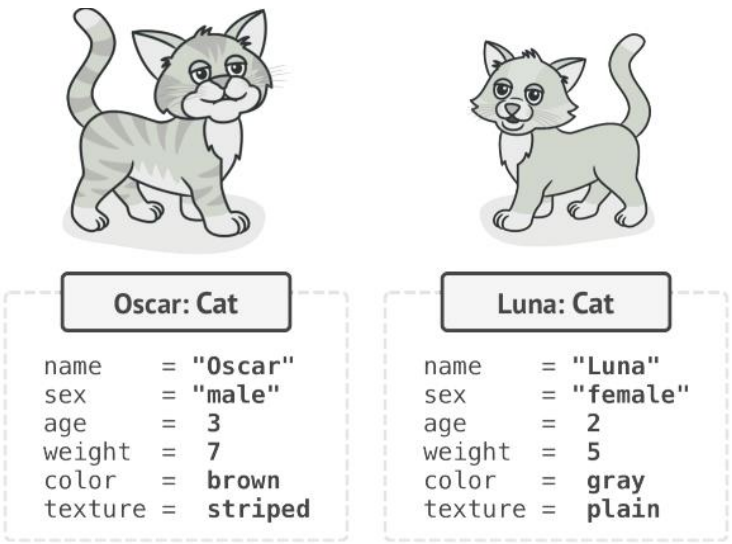
- **Object-oriented programming is a paradigm based on the concept of wrapping pieces of data, and behavior related to that data, into special bundles called objects, which are constructed from a set of "blueprints" , defined by a programmer , called classes.**

34

Objects, classes



35



- **Objects are instances of classes**
- **Data stored inside the object' s fields is often referenced as state, and all the object' s methods define its behavior.**

36

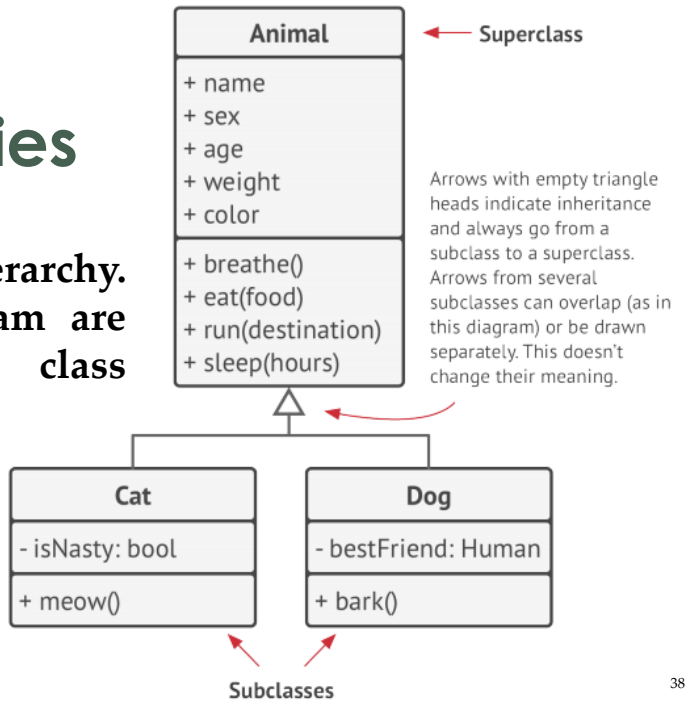
Class

- Class is like a blueprint that defines the structure for objects, which are concrete instances of that class.

37

Class hierarchies

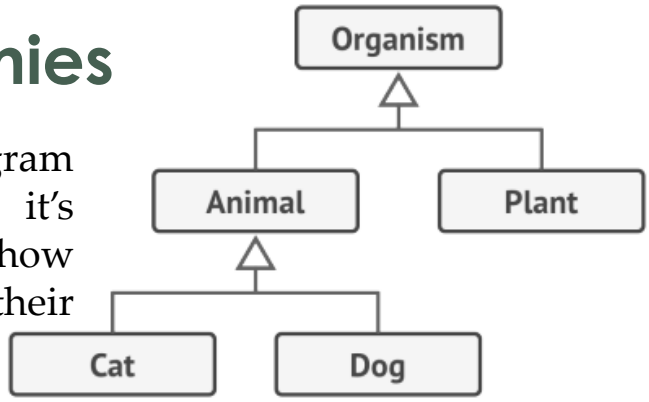
UML diagram of a class hierarchy. All classes in this diagram are part of the Animal class hierarchy.



38

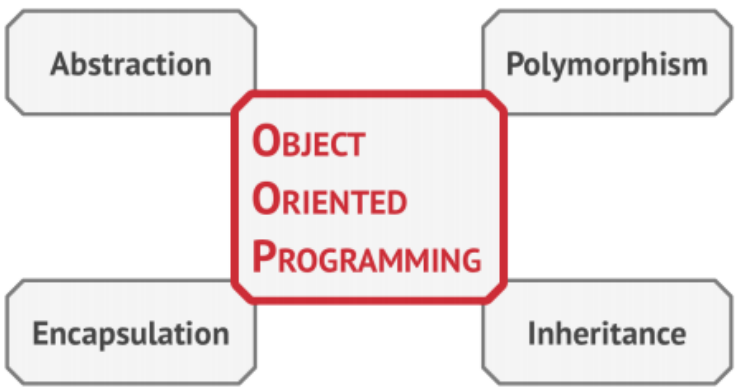
Class hierarchies

Classes in a UML diagram can be simplified if it's more important to show their relations than their contents.



39

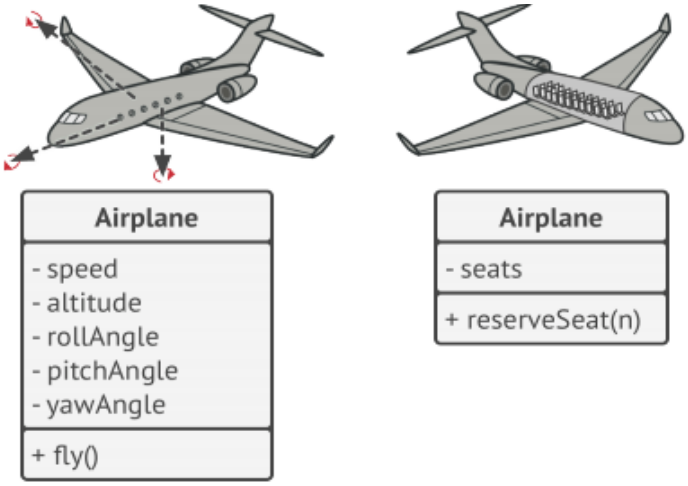
Pillars of OOP



Object-oriented programming is based on four pillars, concepts that differentiate it from other programming paradigms.

40

Abstraction



Different models of the same real-world object.

Abstraction is a model of a real-world object or phenomenon, limited to a specific context, which represents all details relevant to this context with high accuracy and omits all the rest.

41

Encapsulation

Definition:

- **Encapsulation** is the ability of an object to hide parts of its state and behaviors from other objects, exposing only a limited interface to the rest of the program.
- E.g. Car Engine

42

Encapsulation

To start a car engine, you only need to turn a key or press a button. You don't need to connect wires under the hood, rotate the crankshaft and cylinders, and initiate the power cycle of the engine. These details are hidden under the hood of the car . You have only a simple interface: a start switch, a steering wheel and some pedals. This illustrates how each object has an **interface** —a public part of an object, open to interactions with other objects.

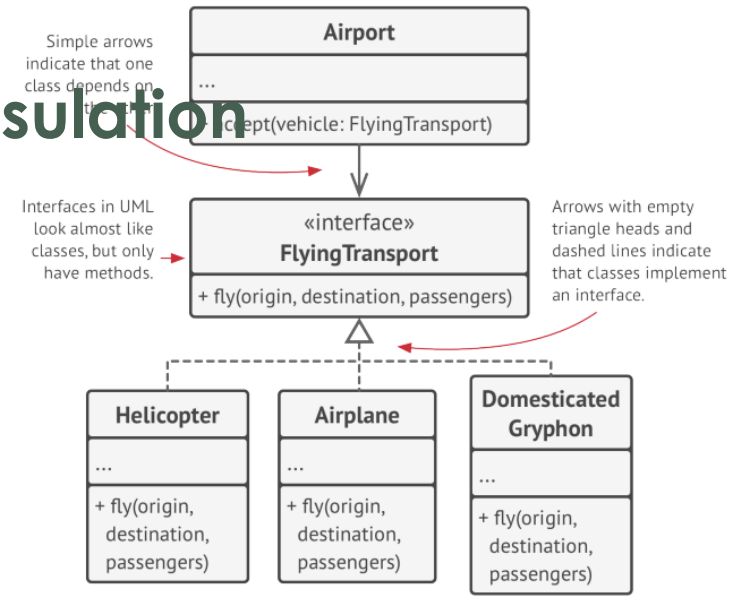
43

Encapsulation

- Interfaces and abstract classes/methods of most programming languages are based on the concepts of abstraction and encapsulation. In modern object-oriented programming languages, the interface mechanism (usually declared with the interface or protocol keyword) lets you define contracts of interaction between objects.

44

Encapsulation



UML diagram of several classes implementing an interface.

Inheritance

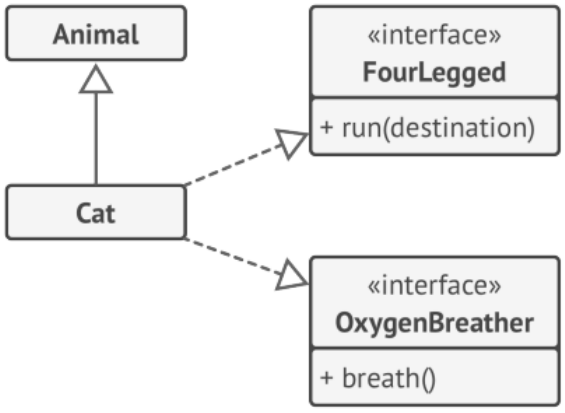
- Inheritance is the ability to build new classes on top of existing ones. The main benefit of inheritance is code reuse. If you want to create a class that' s slightly different from an existing one, there' s no need to duplicate code. Instead, you extend the existing class and put the extra functionality into a resulting subclass, which inherits fields and methods of the superclass.

Inheritance

- The consequence of using inheritance is that subclasses have the same interface as their parent class. You can't hide a method in a subclass if it was declared in the superclass. You must also implement all abstract methods, even if they don't make sense for your subclass.

47

Inheritance



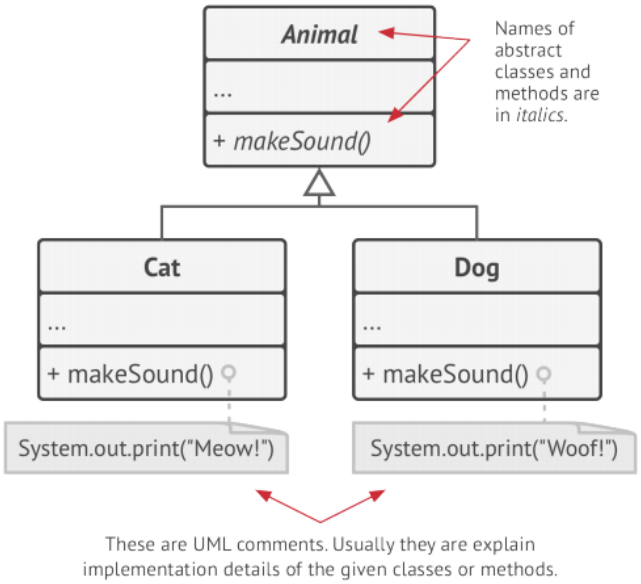
UML diagram of extending a single class versus implementing multiple interfaces at the same time.

48

Inheritance

- Subclass can extend only one superclass.
- Any class can implement several interfaces at the same time.
- If a superclass implements an interface, all of its subclasses must also implement it.

Polymorphism



Polymorphism

```
1  bag = [new Cat(), new Dog()];
2
3  foreach (Animal a : bag)
4    a.makeSound()
5
6  // Meow!
7  // Woof!
```

51

Polymorphism

- "Polymorphism is the ability of a program to detect the real class of an object and call its implementation even when its real type is unknown in the current context".

- You can also think of polymorphism as the ability of an object to " pretend" to be something else, usually a class it extends or an interface it implements. In our example, the dogs and cats in the bag were pretending to be generic animals.

52