

UML

System Analysis & Design
Istanbul Arel University
Spring 2023

Motivation

- Realistic, real life systems are usually **large** and **complex**.
- They require **many** people working together.
 - Developers, testers, managers, clients, users, etc.
- We apply an object-oriented approach to system analysis and design.
 - This approach views a system as a collection of self-contained objects, including both data and processes.

- Object-oriented systems focus on capturing the structure and behavior of information systems in little modules that encompass both data and processes.
- These little modules are known as *objects*.
- A *class* is the general template used to define and create specific *instances*, or objects.
 - Every object is associated with a class.
 - An object is an instantiation of a class.

Objects

- Objects directly relate to real-world entities
- Every object has an identity, state and behavior.
 - Identity: The property which distinguishes it from other objects
 - State: describes the data stored in the object
 - Behavior: Describes the methods in the object's interface by which the object can be used.
- The state of an object is represented by the values of its properties (attributes).

Objects

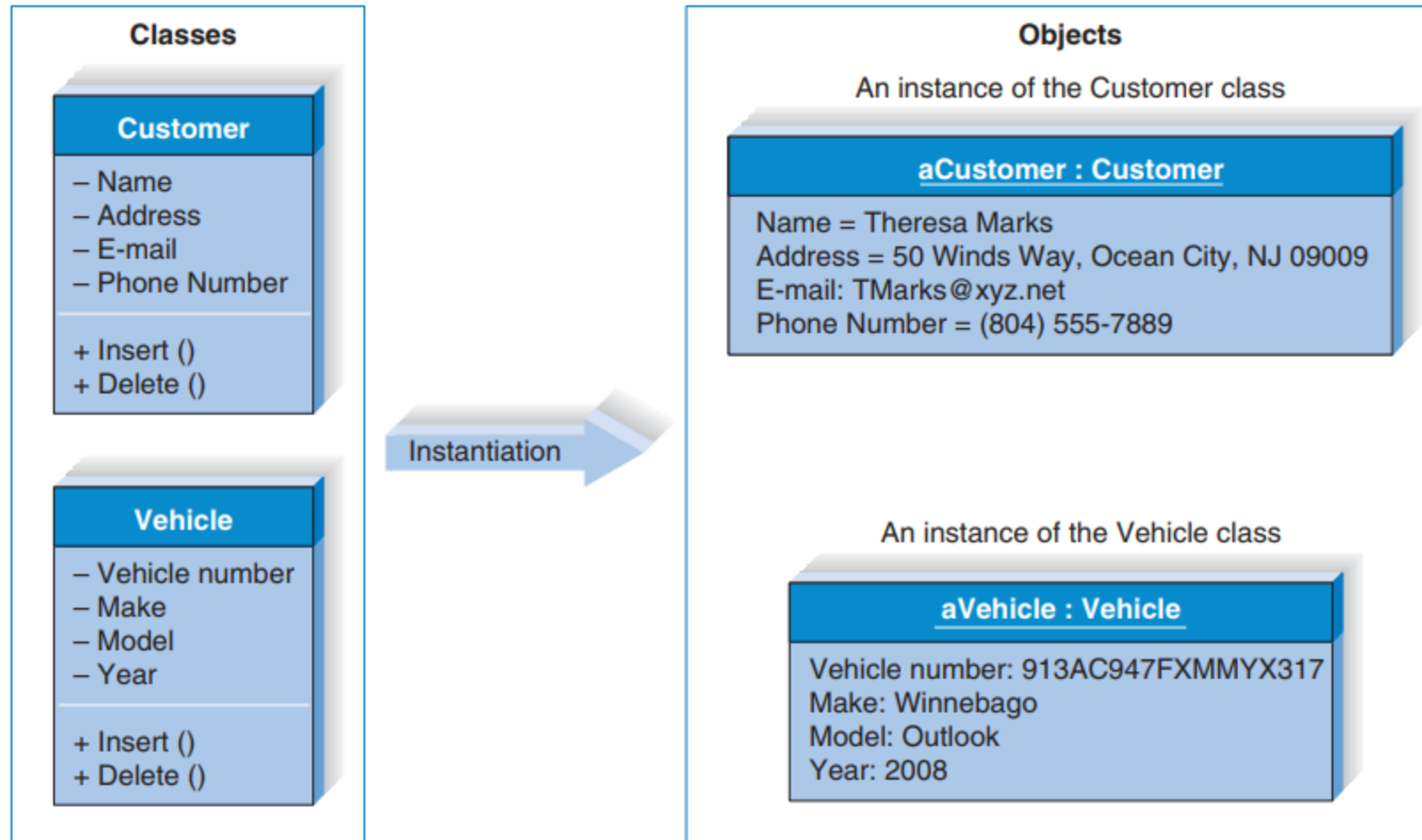
- **Identity:**
 - Dog
 - Bicycle
- **State:**
 - Dogs states (name, color, breed, hungry?)
 - Bicycles (current gear, current speed)
- **Behaviour:**
 - Dogs (barking, fetching, wagging tail)
 - Bicycles (changing gear, applying brakes)

Identifying & Using objects' states

- We need to ask some questions to understand.
 - What possible states can this object be in?
 - What possible behavior can this object perform?
- By attributing state and providing methods to *change* that state, the object remains in control of how the *outside world* is allowed to use it.
 - For example, if a bicycle has only 6 gears, a method to change that should reject <1 and >6 .

- Many individual objects can be of the same kind.
 - Each bicycle was built from the same set of blueprints and will contain same components.
- A class is the *blueprint* from which individual objects are created.

Classes and Objects



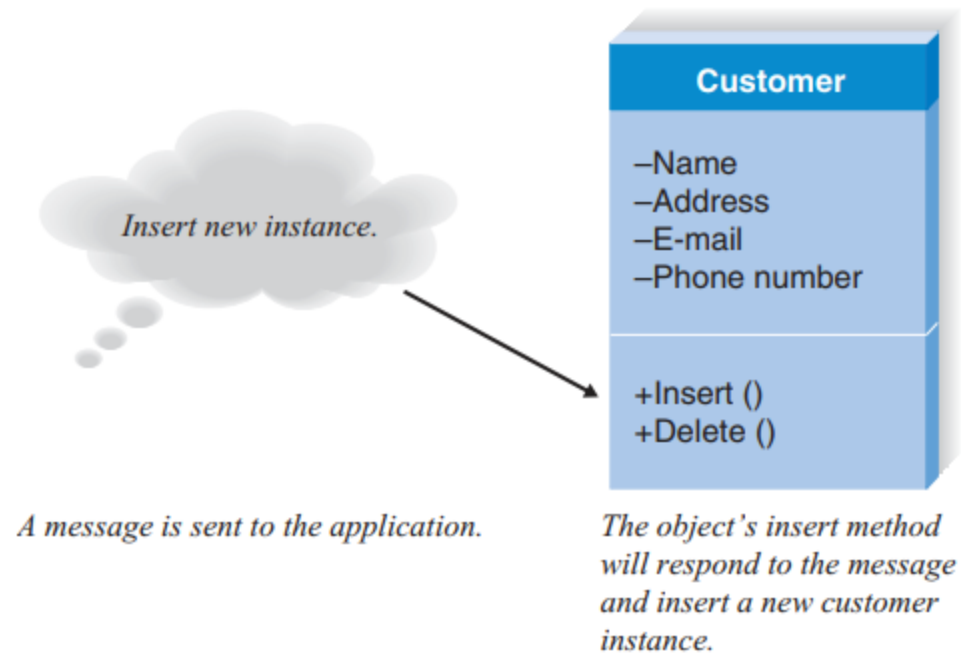
- Each object have *attributes* (name, address, phone number etc.)

Attribute

- An attribute is an element of data which helps to describe an object.
- Attribute types may be restricted by a programming language.

Methods and Messages

- Methods implement an object's behavior. Methods are actions that an object can perform.
 - Functions in procedural languages.
- *Messages* are information sent to objects to *trigger* methods.
 - It is a function call.



Encapsulation

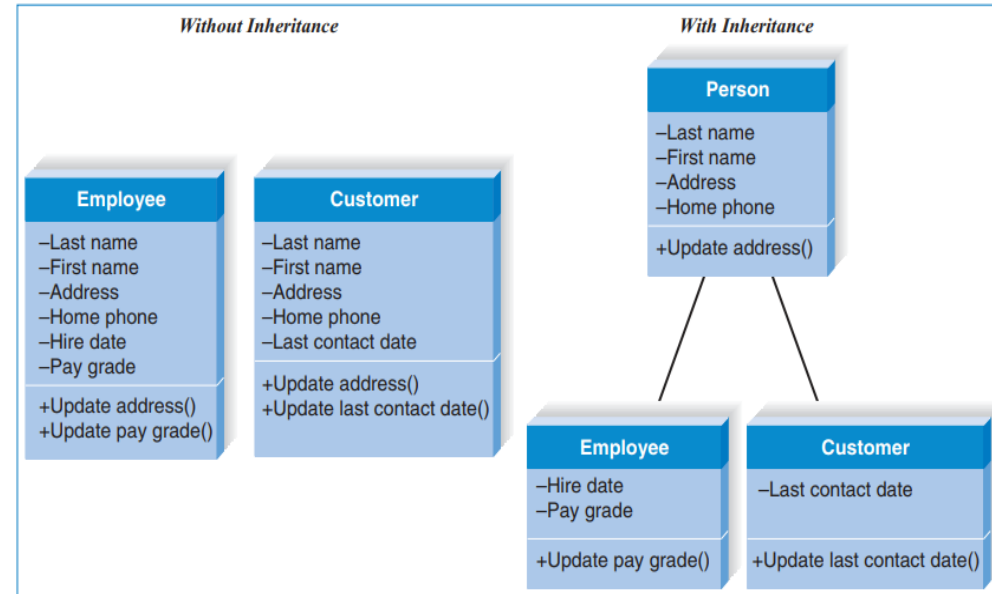
- It is *information hiding*.
 - Public vs private
- Only the information required to use a software module be published to the *user* of the module.
 - Objects are treated like *black boxes*.
- Protects the object's internal state from being corrupted by other objects.
- They are also protected from changes in the object implementation.
- Communication is achieved through an *interface*.

Interface

- Interface is a group of related methods with empty bodies.
- Implementing an interface allows a class to become more formal about the behavior.
 - A remote is an interface between us and the television.
- Interface is an *abstract class* that is used to group related methods with empty bodies.
 - It is a way to achieve *abstraction*.
- We use them to achieve security. We hide certain details and show only the important details of an object.

Inheritance

- Inheritance makes it simpler to define new classes.



- Here we can see that there are several attributes which are common to both classes.
 - We can combine them into a new class (Person) and we then our new classes can *inherit* that parent class.
 - Employee and Customer becomes *subclass*.


Inheritance

- A class called ink-jet printers inherits all the behavior and attributes of the class *computer printer*.
- Inheritance enables *polymorphism*.
- Each subclass is a specialized version of its superclass.

Inheritance

- When a class have *instances*, it is called a **concrete class**.
 - If John and Jane were instances of Customer class, Customer class would be a concrete class.
- There are also classes which do not have any *instances*, they are just there to be templates for other classes. These are called **abstract classes**.
 - Person is an abstract class. Instead of creating instances of Person, we create subclasses which inherit the Person and we create instances of those subclasses.

Polymorphism

- Polymorphism means having many forms.
- Same method can invoke many different kinds of behavior.
 - To talk about polymorphism, we need an inheritance hierarchy.
 - Each class in that hierarchy is dependent upon other classes.
-  for example can be used as *addition* for numbers and *concatenation* for strings.

Polymorphism

```
class Animal {  
    public void animalSound() {  
        System.out.println("The animal makes a sound");  
    }  
}  
  
class Pig extends Animal {  
    public void animalSound() {  
        System.out.println("The pig says: wee wee");  
    }  
}  
  
class Dog extends Animal {  
    public void animalSound() {  
        System.out.println("The dog says: bow wow");  
    }  
}
```

- UML: Unified Modelling Language
- It is mainly used for object-oriented-analysis and design.
 - In larger projects we use OOP.
- UML diagrams are divided into two: **structural** and **behavioral**.

Structural Diagrams

Structure Diagrams

Class	Illustrate the relationships between classes modeled in the system.
Object	Illustrate the relationships between objects modeled in the system. Function when actual instances of the classes will better communicate the model.
Package	Group other UML elements together to form higher level constructs.
Deployment	Show the physical architecture of the system. Can also be used to show software components being deployed onto the physical architecture.
Component	Illustrate the physical relationships among the software components.

Behavioral Diagrams

- These provide the analyst with a way to depict the dynamic relationships among the instances or objects that represent the business information system.
 - Also allow the modeling of dynamic behavior of individual objects throughout their lifetime.
 - They support the analyst in modeling the functional requirements of an evolving information system.

Behavioral diagrams

Behavioral Diagrams

Activity	Illustrate business work flows independent of classes, the flow of activities in a use case, or detailed design of a method.
Sequence	Model the behavior of objects within a use case. Focuses on the time-based ordering of an activity.
Communication	Model the behavior of objects within a use case. Focuses on the communication among a set of collaborating objects of an activity.
Interaction Overview	Illustrate an overview of the flow of control of a process.
Timing	Illustrate the interaction that takes place among a set of objects and the state changes that they go through along a time axis.
Behavioral State Machine	Examine the behavior of one class.
Protocol State Machine	Illustrate the dependencies among the different interfaces of a class.
Use Case	Capture business requirements for the system and to illustrate the interaction between the system and its environment.

Diagrams

- Different diagrams are used in different development processes and they play an important role.
- As the system is developed, the diagrams will have more detail which are going to be crucial for *development*.
 - In other words, diagrams move from *documenting the requirements* to laying out the *design*.
- Four UML diagramming techniques dominates object-oriented projects:
 - **Use-case diagrams**
 - **Class diagrams**
 - **Sequence Diagrams**
 - **Behavioral state machine diagrams**

Diagrams

- Use case diagram is used to *summarize* the set of use cases for the system.
 - After that, the other three diagrams are used to further define the evolving systems.
 - Use case diagram is always created **first**.
 - However the order of creating the other diagrams can change.
 - Usually the order is
1. Use case Diagram
 2. Class Diagram
 3. Sequence and behavioral state machine diagrams.

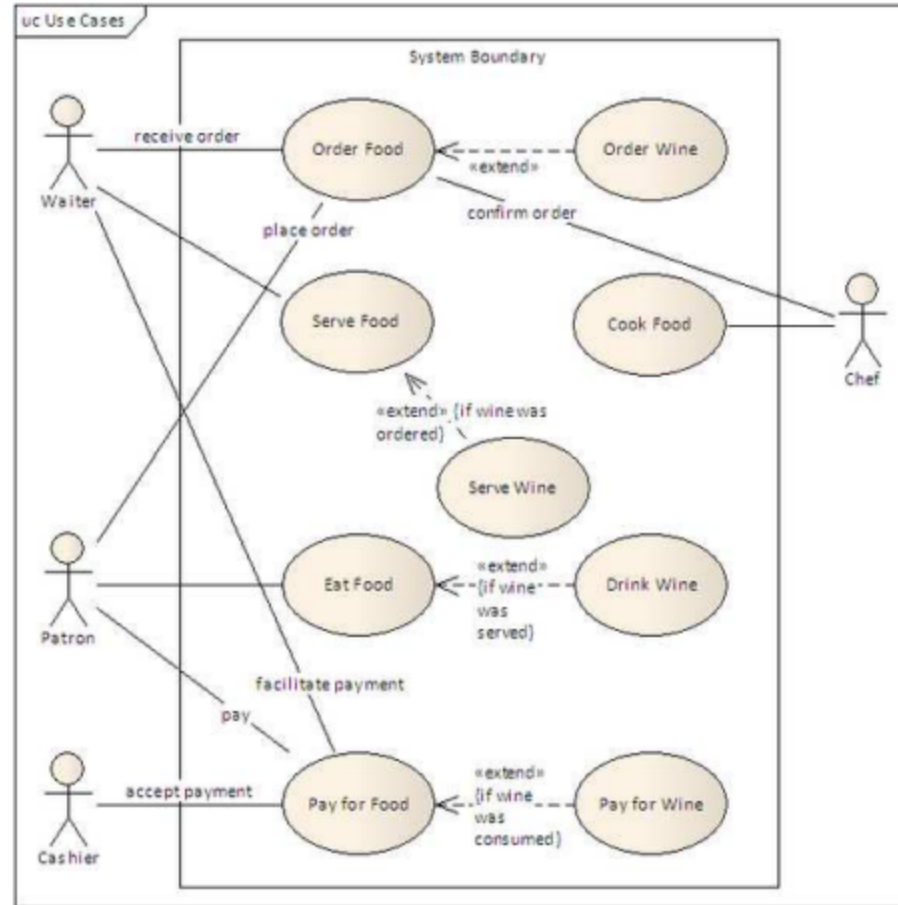
Diagrams

- Class diagrams show what objects contain and how they are related.
- Sequence diagrams show how objects dynamically interact
 - Developing sequence diagrams often leads to changes in the class diagrams and vice versa. So analysts often move back and forth between the two.
- Behavioral state diagrams are developed **after** the class diagrams have been refined.

Diagrams

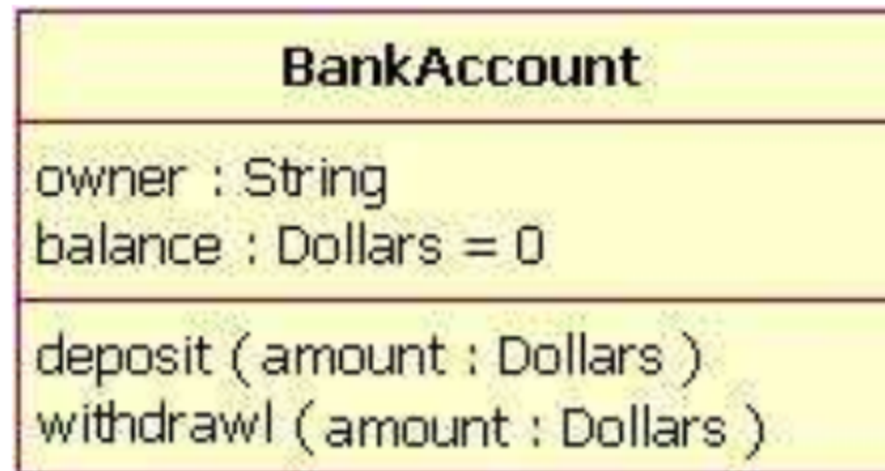
Use Case Diagram

Show use cases, actors and their interrelationships.



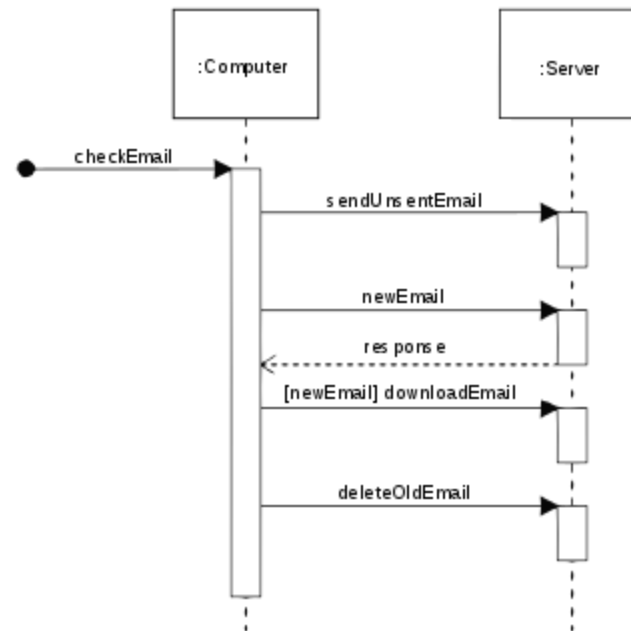
Class Diagram

Shows a collection of static model elements such as classes and types, their contents and their relationships.



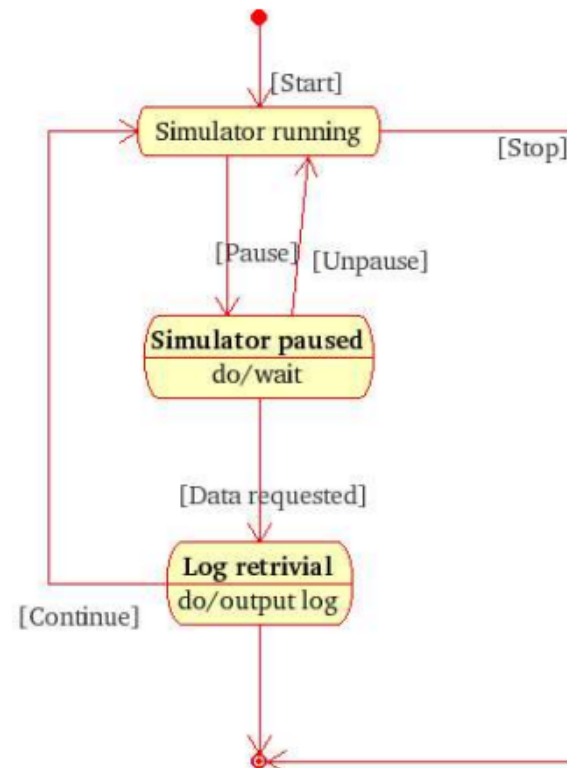
Sequence Diagram

Models the sequential logic, in effect the time ordering of messages between classifiers.



State machine diagram

Describes the states of an object or interaction may be in, as well as the transitions between states. Formerly referred to as a *state diagram*, *state chart diagram* or *state-transition diagram*.



Modeling

System development

- System development is *model building*.
- Large project is *complex*.
 - Large number of components and a lot of team work.
- It is hard to communicate via talking and it is sometimes not reliable or accurate
- Models are *standards* and they are more reliable and accurate.
- Modeling is a part of developing a model

Types of models

- There are different types of models for every phase in development.
 - Requirement model
 - Analysis model
 - Design model
 - Implementation model
 - Test model

Types of models

- Requirement model *describes* **users' requirements** and **functionality**.
- Analysis model *gives* **system specifications** and a **robust and changeable structure and structured components**.
- Design model presents a **refined structure to the current implementation environment**.
- Implementation model *documents* **details of how a design is implemented**.
- Test model *gives* **verification** and **validation**.

Requirements Model

- Users' requirements try to solve;
 - A client's needs and expectations
 - Requirements should be problem-based and not describe solutions
 - Because the solutions are not developed yet
 - Requirements are modeled using Use Case diagrams