Object Oriented Modelling and Design

BCS1430

Dr. Ashish Sai

Week 2 Lecture 1 & 2

BCS1430.ashish.nl

EPD150 MSM Conference Hall

Table of contents

- Object-Oriented Analysis and Design
- What is good software?
- How to design good (OO) software?
- Gathering Requirements
- Analysis

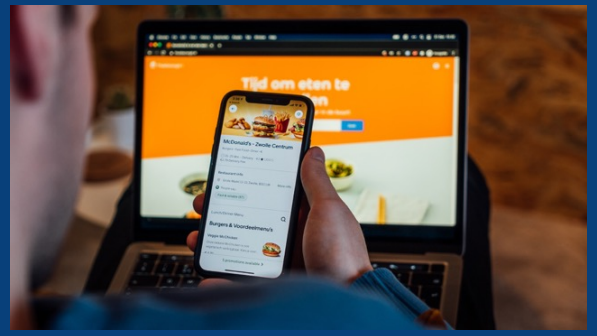
Lecture 1

> Understanding what needs to be designed

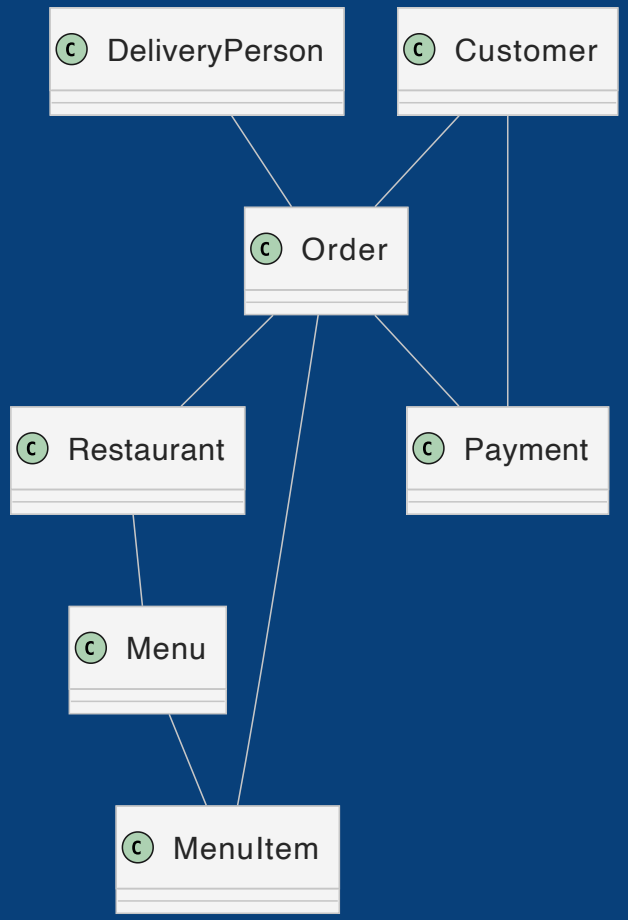
Object-Oriented Analysis and Design

What is Object-Oriented Analysis and Design?

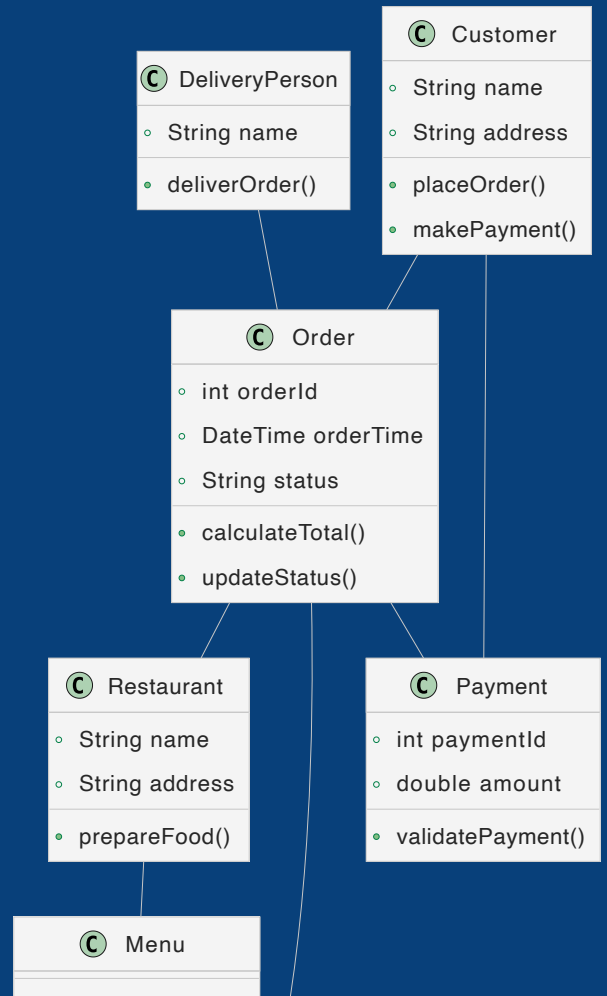
Analyze the system



Model the system



Design the software

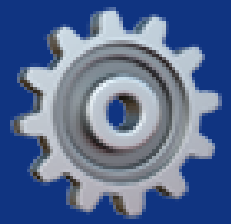


What is Object-Oriented Analysis and Design?

- Analysis: what are we trying to do?
- Design: how are we going to do it?

What is Object-Oriented Analysis and Design?

- **Object-Oriented Analysis (OOA)**:
  - This is the process of looking at a ***problem***, ***system***, or ***situation***, and identifying the *objects* and *interactions* between those objects.



What is Object-Oriented Analysis and Design?

- Design is a conceptual solution that meets the requirements — how can we solve the problem
- **Object-Oriented Design (OOD)**:
  - We take the analysis results and mold them into a design that can be implemented in a specific programming language such as Java.

Why does it matter?

- In the real world, complex systems are often best understood through the interactions of simpler parts. OOA&D leverages this principle to create ***modular, reusable*** and ***flexible*** software.

Benefits of OOA&D

Modularity: Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components. (ISO/IEC 2011)

Maastricht University

UM & 2FA

Benefits of OOA&D

- **Reusability:** Objects and classes created for one project can be used in another, reducing development time and increasing productivity.
- Degree to which an asset can be used in more than one system, or in building other assets. (ISO 25010)

Payment processing for different services (e-commerce, in-app purchases)

Benefits of OOA&D

- **Flexibility:** Systems designed with OOA&D can be easily adapted to meet changing requirements.

OO Analaysis in Software Development

- OOA is the investigative phase where you dive deep into the problem domain.
- Ask the right questions and identify the objects and interactions.
- Goal: Create a model of the real world that can be translated into a software system.

It’s easier to change a design than to change a built system.

OO Design in Software Development

- Turn the conceptual model (from OOA) into a blueprint for building the system.
- Think about the system architecture, the choice of system components etc.

What is good software?

Understanding Software Quality

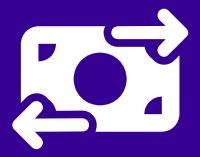
ISO 25010 [1] defines software quality with a comprehensive set of attributes:

- **Functionality**
- **Reliability**
- **Usability**
- **Efficiency**
- **Maintainability**
- *Portability*
- *Compatibility*
- *Security*

1. https://www.iso.org/standard/35733.html

Quality Attribute: Functionality

- **Definition:** The degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions



Completeness, Correctness & Appropriateness

Quality Attribute: Reliability

- **Definition:** Reliability is about the software's capacity to maintain its performance level under stated conditions for a specified period.

Maturity, Fault Tolerance and Recoverability

Quality Attribute: Usability

- **Definition:** Usability refers to how well a product or system can be used to achieve specified goals effectively, efficiently, and satisfactorily.

Understandability, Learnability and Operability

Quality Attribute: Efficiency

- **Definition:** Efficiency relates to the relationship between the performance level of the software and the amount of resources used, under stated conditions.

Time Behavior, Resource Utilization and Capacity

Quality Attribute: Maintainability

- **Definition:** Maintainability is the ease with which the software can be modified to correct faults, improve performance, adapt to a changed environment, or enhance the product.

Modularity, Reusability and Analyzability

How to design good (OO) software?

Developing good (OO) Software

1. Understand the requirements
2. Analyze the requirements in an OO context (OO Analaysis)
3. Design the software using OO design principles (more on this in the later half of the lecture)

Gathering Requirements

Introduction to Software Requirements

Requirements are descriptions of the system's features, and constraints.

- Essential for guiding development and testing.
- Help in setting customer expectations.
- Basis for project planning and design decisions.

Types of Software Requirements

Software Requirements:

- Functional Requirements
- Non-functional Requirements
- Domain Requirements

Functional Requirements (FR)

Functional Requirements describe:

- The services the system must provide.
- How the system should react to particular inputs.
- How it should behave in certain situations.

Functional Requirements (FR)

Functional Requirements Characteristics:

- Clearly state what the system should do.
- Include calculations, technical details, data manipulation and processing, and other specific functionality.

Functional Requirements (FR)

Example:

> “The system shall allow users to enter their credentials to log in.”

> “Upon receiving a user’s input, the system shall calculate and display the results within 2 seconds.”

Non-Functional Requirements

Non-functional requirements set criteria to judge the operation of a system, rather than specific behaviors, including constraints on the system and standards.

Non-Functional Requirements

- **Performance:** Speed, responsiveness, consumption.
- **Reliability:** Frequency of failure, recoverability.
- **Usability:** Understandability, learnability, operability.
- **Security:** Access control, integrity, confidentiality.

Non-Functional Requirements

Example:

> “The system should load the homepage in under 3 seconds when accessed from a standard broadband connection.”

> “User data should be encrypted using industry-standard encryption algorithms.”

Domain Requirements

Domain requirements reflect domain-specific knowledge, standards, or regulations that the software must comply with.

Key Aspects:

- Reflect application domain behavior.
- Influence system functionality and performance.
- Include legal and regulatory requirements.

Domain Requirements

Example:

> “The software must store medical data in compliance with healthcare regulations.”

> “The system must perform currency conversion according to international financial standards.”

The Art of Gathering Requirements

- Requirements gathering is the process of collecting the needs and specifications from stakeholders to design a system that meets their expectations.
- Understand stakeholder needs, and translate those into technical specifications.

Stakeholders

Stakeholders are:

- Individuals or groups with an interest in the success of the project.

Example:

- Clients, end-users, project managers, developers or your examiner.

Techniques for Gathering Requirements

- Techniques:
  - Interviews,
  - Surveys/Questionnaires,
  - Observations,
  - Workshops, and
  - Brainstorming sessions.

Introduction to Use Cases

- A use case is a detailed description of how users perform tasks, outlining a system's behavior from a user's perspective.
  - Starts with a user's goal and ends when that goal is fulfilled, providing a sequence of steps and interactions.

Elements of a Use Case

- **Actor:** The user performing the behavior.
- **Stakeholder:** Individuals with interests in the system.
- **Preconditions:** States that must be true before the use case runs.
- **Triggers:** Events that initiate the use case.
- **Main Success Scenarios (Basic Flow):** The expected successful path.
- **Alternative Paths (Alternative Flows):** Variations when deviations occur.

Writing a Use Case

1. **Identify users:** Determine who will be using the system.
2. **Define actions:** Each action becomes a use case.
3. **Describe normal and alternate courses:** Outline the basic and alternative flows.
4. **Note commonalities:** Identify similar patterns across use cases.
5. **Repeat for all users:** Ensure comprehensive coverage.

Simple Laundry Use Case 

- **Actor:** Housekeeper
- **Basic Flow:** Sorting, washing, drying, folding, ironing, and discarding items.
- **Preconditions:** It’s Wednesday, and there is laundry.
- **Trigger:** Dirty laundry is present.
- **Post Conditions:** All laundry is clean and either folded or hung up.

Alternative Flows in Laundry Use Case 🧺

- **Alternative Flow 1:** If items are wrinkled, they are ironed and hung.
- **Alternative Flow 2:** If items are still dirty, they are rewashed.
- **Alternative Flow 3:** If items have shrunk, they are discarded.

Example Use Case: Enter Order 🏦

Description: This use case describes how a customer can enter a new order into the system.

Actors:

- Customer
- System
- Market

Preconditions: Customer is logged on to the website.

Postconditions:

- The trade has been processed.
- The Customer has received the confirmation from the system.

Basic Flow of Enter Order 🏦

Step ID	Actor	Action	Notes and References
BF-1	Customer	Customer navigates to the order entry page.	
BF-2	System	System displays the order entry page.	
BF-3	Customer	Customer enters the following order details: Buy/Sell, Quantity, Stock Symbol	
BF-4	Customer	Customer submits the order.	
BF-5	System	System validates the information entered by the customer.	AF-1: Customer enters invalid information
BF-6	System	System submits the order to the marketplace.	
BF-7	Market	Market executes the order.	
BF-8	Market	Market sends the execution report to the system.	
BF-9	System	System displays the execution report to the customer.	

Alternate Flows - Invalid Information

AF-1: Customer enters invalid information

Step ID	Actor	Action	Notes and References
AF-1-1	System	If the customer has entered invalid information the system will display an error message: “You have entered invalid information.”	
AF-1-2	Customer	Customer corrects the information and resubmits.	
AF-1-3		[Go to BF-5]	

Exception Flows - System Issues

EF-1: System is down or unavailable

Step ID	Actor	Action	Notes and References
EF-1-1	System	If the system is down/unavailable the system will display a message to the customer: “The system is currently unavailable.”	
EF-1-2		[Use case ends]	

EF-2: System cannot connect to the market

Step ID	Actor	Action	Notes and References
EF-1-1	System	If the system cannot connect to the market the system will display a message to the customer: “The system cannot connect to the market. Please try again later.”	
EF-1-2		[Use case ends]	

Supplemental Requirements

ID	Name	Description
SR-1	Tabbing	The system will enable the user to tab from field to field on the order entry page.

Use Cases: Key Takeaways

- **User-Centric Design:** Focus on user actions and system responses.
- **Detailing Flows:** Clear definition of basic, alternate, and exception flows.
- **Supplemental Requirements:** Address additional user experience enhancements.
- **Adaptability:** The use case's structure allows for easy updates and modifications.

Case Study: Starbucks Mobile Order & Pay ☕

Starbucks introduced Mobile Order & Pay to allows customers to place orders and pay in advance to reduce waiting times.

Objectives:

- Improve customer satisfaction by reducing in-store waiting times.
- Increase operational efficiency and order accuracy.
- Enhance the customer’s personalization and convenience.

Use Case: Placing an Order through Mobile Order & Pay ☕

Actor: Customer

Preconditions:

- The customer has the Starbucks mobile app installed.
- The customer has an account with payment information.

Basic Flow:

1. The customer opens the app and selects the ‘Order’ option.
2. The customer browses the menu and selects items to order.
3. The customer reviews the order and makes any modifications.
4. The customer confirms the order and selects a pickup location.
5. The customer pays for the order through the app.
6. The system sends the order to the selected store.
7. The customer receives a notification when the order is ready for pickup.

Postconditions:

- The customer picks up the order and is satisfied with the service.

Derived Requirements from Use Case

Functional Requirements:

- FR1: The app must allow users to browse the menu and select items.
- FR2: The app must provide an option for order modification before confirmation.
- FR3: The app must securely process payment information.
- FR4: The system must send the order to the selected store and confirm readiness through a notification.

Non-Functional Requirements:

- NFR1: The app should load the menu within 2 seconds (Performance).
- NFR2: The payment system must comply with PCI DSS standards (Security).
- NFR3: The app should be accessible and user-friendly (Usability).
- NFR4: The system should handle 1000 orders simultaneously without performance degradation (Scalability).

Requirements Prioritization

- Not all requirements are of equal importance.
- Evaluate the urgency, value, and feasibility of each requirement. (Prioritization)
- Technique(s): MoSCoW (Must have, Should have, Could have, Won’t have this time)

Requirements Prioritization: MoSCoW Technique

MoSCoW technique is a prioritization tool used to reach a common understanding on the importance of various requirements.

Components:

- **Must have (M):** Essential features, required for successful and functional project.
- **Should have (S):** Important but not vital features.
- **Could have (C):** Desirable features that are beneficial but not crucial.
- **Won't have this time (W):** Features that, may be useful, are not a priority for this iteration.

Starbucks ☕- Derived Requirements from Use Case

Functional Requirements:

- FR1: The app must allow users to browse the menu and select items.
- FR2: The app must provide an option for order modification before confirmation.
- FR3: The app must securely process payment information.
- FR4: The system must send the order to the selected store and confirm readiness through a notification.

Non-Functional Requirements:

- NFR1: The app should load the menu within 2 seconds (Performance).
- NFR2: The payment system must comply with PCI DSS standards (Security).
- NFR3: The app should be accessible and user-friendly (Usability).
- NFR4: The system should handle 1000 orders simultaneously without performance degradation (Scalability).

MoSCoW Prioritization - Functional Requirements

Must have (M):

- FR1: Browse menu and select items - critical for basic functionality.
- FR3: Securely process payment - essential for trust and legal compliance.

Should have (S):

- FR2: Order modification - important for user satisfaction but not critical.

Could have (C):

- Additional personalized recommendations based on user preferences.

Won't have this time (W): - Advanced custom ordering options that require new technology.

MoSCoW Prioritization - Non-Functional Requirements

Must have (M):

- NFR2: Compliance with PCI DSS for payment security - non-negotiable.
- NFR3: Basic app usability - essential for user adoption.

Should have (S):

- NFR1: Fast menu loading - important for a good user experience.

Could have (C):

- NFR4: Scalability to handle more than 1000 orders - desirable for future growth.

Won't have this time (W):

- Integration with third-party loyalty systems - considered for future phases.

Analysis

Object-Oriented Analysis

What is OOA?

Methodical approach to understanding a system by viewing it through the lens of the ‘objects’ (real-world entities) it involves.

Object-Oriented Analysis

Key Aspects:

- **Understanding the Domain:** Examining the problem or system from an object-centric perspective.
- **Capturing Requirements:** Identifying what the system must do from the viewpoint of the objects.
- **Defining the System:** Creating a model that effectively addresses and fulfills the identified needs.

Identifying Objects and Classes

Core Concepts:

- **Objects:** Instances from the problem domain represented in the system.
- **Classes:** Blueprints defining attributes and behaviors of objects.

Identifying Objects and Classes

Process:

1. **Examine the Domain:** Analyze the real-world scenario to model.
2. **Spot Key Entities:** Identify significant entities that play vital roles.
3. **Define Nature and Functionality:** Determine attributes and operations for each entity.

Identifying Objects and Classes

Example: - *Library System*: Key objects like books, members, and loans with specific attributes and behaviors.

Identify Relationships and Interactions

Objects do not exist in isolation!

Types of Relationships:

- **Associations:** Simple connections between objects.
- **Aggregations:** Whole-part relationships indicating a collective.
- **Compositions:** Strong whole-part relationships with dependency.

Identify Relationships and Interactions

Example: - *E-commerce Platform*: ‘Customer’ places an ‘Order’ containing ‘Products’.

Associations

Definition:
Represents a “use-a” or “has-a” relationship where one object uses or interacts with another.

- Can be one-to-one, one-to-many, many-to-one, or many-to-many.

Associations

```
1 class Customer {
2     private Order order;
3     void placeOrder() { /*...*/ }
4 }
5
6 class Order {
7     // Order related methods
8 }
```

Here, a Customer has an association with Order as it places an order. This is a one-to-one relationship.

Aggregations

Definition:

A specialized form of association representing a “whole-part” relationship, with objects having their own life cycle but forming a whole.

- Denotes a ‘has-a’ relationship.
- The part can exist independently of the whole.

Aggregations

```
1 class Team {
2     private List<Player> players;
3 }
4
5 class Player {
6     // Player specific methods
7 }
```

A Team consists of multiple Players. Players can exist without a team, illustrating the whole-part relationship.

Compositions

Definition:

Composition is a strong form of aggregation implying ownership. When the whole is destroyed, so are the parts.

- Represents a strong ‘contains-a’ relationship.
- The life cycle of the part is tied to the whole.

Compositions

```
1  class Engine {
2      // Engine specific methods
3  }
4
5  class Car {
6      private Engine engine = new Engine();
7      void start() { /* Starts the engine */ }
8  }
```

A Car has a Engine as part of its composition. If the car ceases to exist, so does the engine, indicating a strong dependency.

Object Interaction Analysis

- Object Interaction Analysis is a technique used to understand how objects in the system will communicate and collaborate.
- Interaction diagrams are used to visualize these interactions [1].

1. More on these diagrams in the next lecture.

Object Interaction Analysis

- For example, in an airline reservation system, understanding how a ‘Passenger’ object interacts with ‘Flights’, ‘Tickets’, and ‘Payments’ objects is crucial for designing a functional system.

Assigning Responsibilities with CRC Cards

Class-Responsibility-Collaborator (CRC) Cards: CRC cards is a simple tool used to define the behaviors and interactions of classes.

Components of a CRC Card:

- **Class Name:** The entity or concept being modeled.
- **Responsibilities:** What the class should know or do (its behavior).
- **Collaborators:** Other classes this class interacts with or uses.

CRC Card Example - ShoppingCart

Class Name: ShoppingCart

Responsibilities:

- Add Item: Include a new product in the cart.
- Calculate Total: Compute the total cost of items in the cart.
- Remove Item: Take out a product from the cart.
- Checkout: Initiate the purchasing process.

Collaborators:

- Product: Items that can be added to the shopping cart.
- User: The customer who owns the shopping cart.

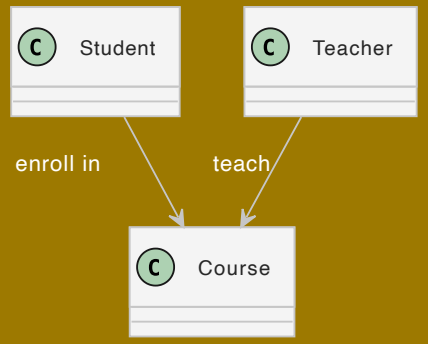
CRC Card Example - ShoppingCart

```
1  class ShoppingCart {
2      private List<Product> products;
3      private User owner;
4
5      void addItem(Product product) { /*...*/ }
6      double calculateTotal() { /*...*/ }
7      void removeItem(Product product) { /*...*/ }
8      void checkout() { /*...*/ }
9  }
10
11 class Product { /* Product details */ }
12 class User { /* User details */ }
```

The ShoppingCart class has clear responsibilities like managing items and calculating totals, and it collaborates with Product and User classes to fulfill these responsibilities.

Class Diagrams: The Static Blueprint

- Class diagrams show the classes, along with their attributes and operations, and the relationships between them.



See you tomorrow! 👋