MSS Module 2 Important Topics

> ***Table of contents***

Functional and non-functional requirements

Functional requirements

- Statements of services the system should provide
- How the system should react to particular inputs
- How the system should behave in particular situations.
- May state what the system should not do.

Non-functional requirements

- Non-functional requirements are like rules for the system that don't involve specific tasks.
- They are limits on what the system can do, like how fast it should work or following certain standards during development.
- Often apply to the system as a whole rather than individual features or services.

Requirements Engineering Process

- Requirements Engineering (RE) is a systematic process of gathering, documenting, analyzing, and managing the needs and constraints of a system or software product from various stakeholders.

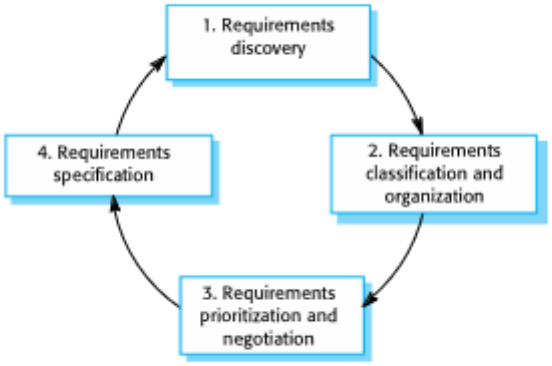
Key activities in Requirements Engineering include:

- Requirements elicitation;
- Requirements analysis;
- Requirements validation;
- Requirements management.

Requirements elicitation, Requirements validation, Requirements change

Requirements Elicitation

- Sometimes called requirements elicitation or requirements discovery.
- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system’s operational constraints.



Requirements elicitation process

Stages involved

- Requirements discovery
  - Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.
- Requirements classification and organization,
  - Groups related requirements and organises them into coherent clusters
- Requirements prioritization and negotiation,
  - Prioritising requirements and resolving requirements conflicts
- Requirements specification
  - Requirements are documented and input into the next round of the spiral.

Techniques used

1. Interviewing

- In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed.
- Requirements are derived from the answers to these questions
- There are 2 types of interviews
  - Closed Interviews
    - Here the stakeholder answers a predefined set of questions
  - Open Interviews
    - There is no predefined agenda.
    - The requirements engineering team explores a range of issues with system stakeholders

2. Ethnography

- Ethnography is an observational technique that can be used to understand operational processes and help derive requirements for software to support these processes
    - **Ethnography** is like being a detective to understand how things work.
    - Instead of just asking people or reading about it, you watch them closely.
    - This is useful for figuring out what a computer program needs to do to support how people really work, not just what the company says they should do.
    - Ethnography helps find hidden requirements.

Requirements Validation

- Requirements validation is the process of checking that requirements define the system that the customer really wants.
- Requirements validation is critically important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service

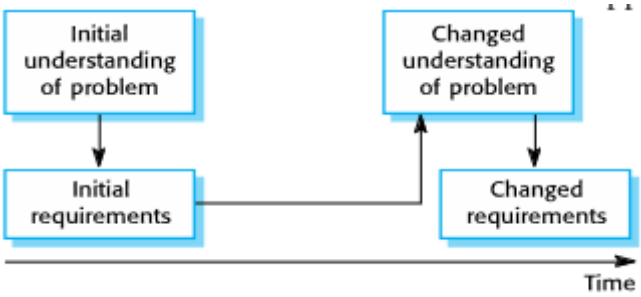
Checks to be done

- Validity Checks
    - These check that the requirements reflect the real needs of system user
- Consistency checks
    - Requirements in the document should not conflict
- Completeness checks
    - The requirements document should include requirements that define all functions and the constraints intended by the system user.
- Realism checks
    - Ensure that they can be implemented within the proposed budget for the system.
- Verifiability
    - Write a set of tests that can demonstrate that the delivered system meets each specified requirement

Requirements Change

- The business and technical environment of the system always changes after installation.
    - New hardware may be introduced,
    - Business priorities may change
    - New legislation and regulations may be introduced
- The people who pay for a system and the users of that system are rarely the same people
- Large systems usually have a diverse user community

- With many users having different requirements and priorities that may be conflicting or contradictory



Requirements change Management

- Problem analysis and change specification
  - During this stage, the problem or the change proposal is analyzed to check that it is valid
- Change analysis and costing
  - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements
- Change implementation
  - The requirements document and, where necessary, the system design and implementation, are modified.

Traceability Matrix

- Traceability is a software engineering term that refers to documented links between software engineering work products
- A traceability matrix allows a requirements engineer to represent the relationship between requirements and other software engineering work products.
- They can provide continuity for developers as a project moves from one project phase to another, regardless of the process model being used

---

Software Requirements Specification

- The process of writing down the user and system requirements in a requirements document
- User requirements have to be understandable by end-users and customers who do not have a technical background

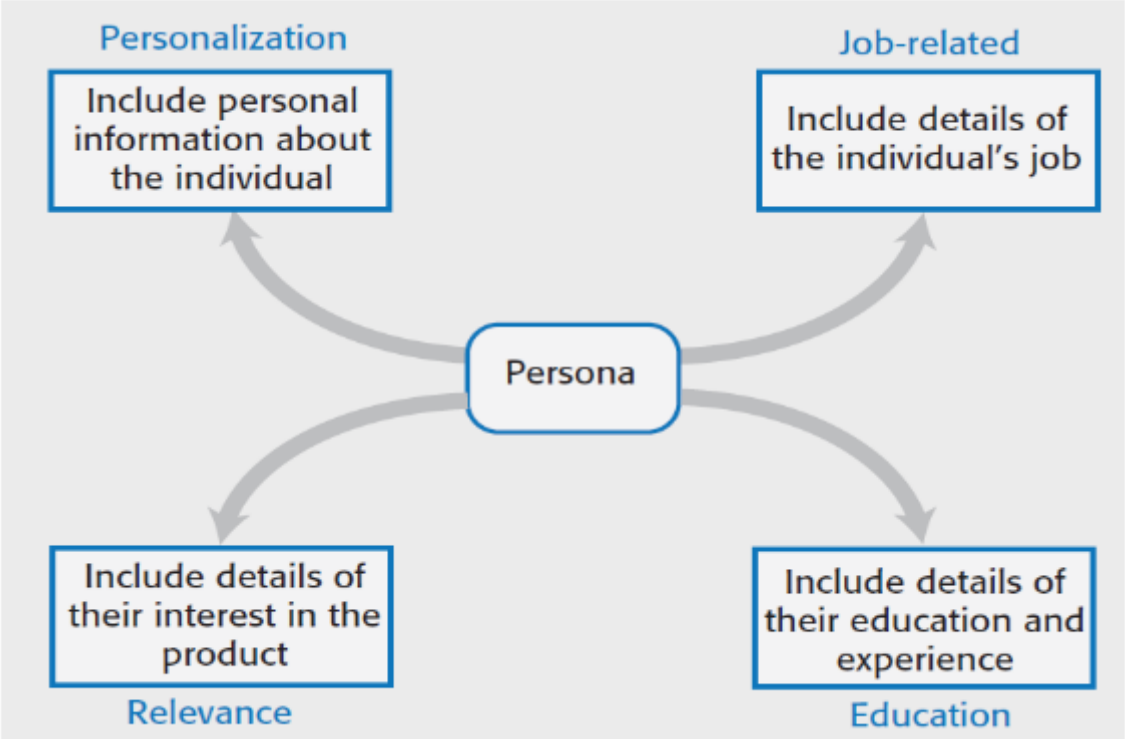
- System requirements are more detailed requirements and may include more technical information

Ways of writing specifications

- Natural language
  - The requirements are written using numbered sentences in natural language.
- Structured natural language
  - The requirements are written in natural language on a standard form or templat
- Design description languages
  - This approach uses a language like a programming language, but with more abstract features to specify the requirements
- Graphical notations
  - Graphical models, supplemented by text annotations, are used to define the functional requirements for the system
  - UML use case and sequence diagrams are commonly used.
- Mathematical specifications
  - These notations are based on mathematical concepts

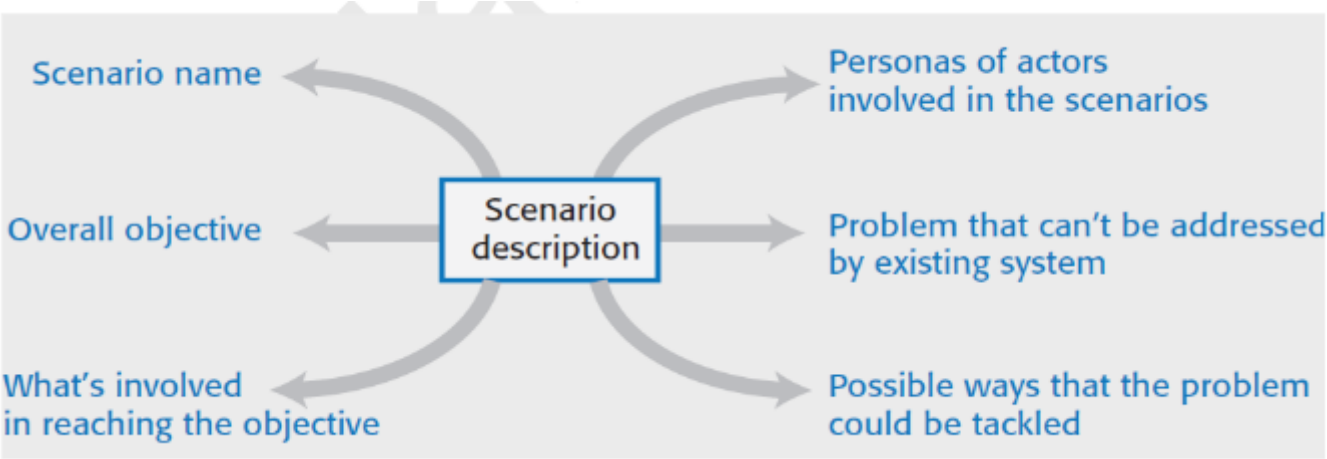
Personas, Scenarios, User stories, Feature identification

Personas



- Personas are 'imagined users' where you create a character portrait of a type of user that you think might use your product.
  - For example, if your product is aimed at managing appointments for dentists, you might create a dentist persona, a receptionist persona and a patient persona
- A persona should 'paint a picture' of a type of product user. They should be relatively short and easy-to read.
- Aspects of persona description
  - Personalization
  - Job Related
  - Education
  - Relevance

Scenarios



- A scenario is a narrative that describes how a user, or a group of users, might use your system
- There is no need to include everything in a scenario
- It is simply a description of a situation where a user is using your product's features to do something that they want to do

User Stories

User Stories are like short stories that describe something a user wants from a software system

- **Detailed Descriptions:**
  - User stories tell exactly what a user wants in a clear and structured way.
- **Focus on Small Bits**
  - User stories are best when they focus on one thing or a small part of a feature. This way, it can be completed in a single work cycle, which is usually called a sprint.
- **Complex Stuff:**
  - If a story is about something more complicated that might take a few work cycles, it's called an "epic." Epics are like big chapters in the user storybook.

In simple terms, user stories are like little chapters that users write to explain what they want from a software, and it helps the team plan and build the software step by step

Feature Identification

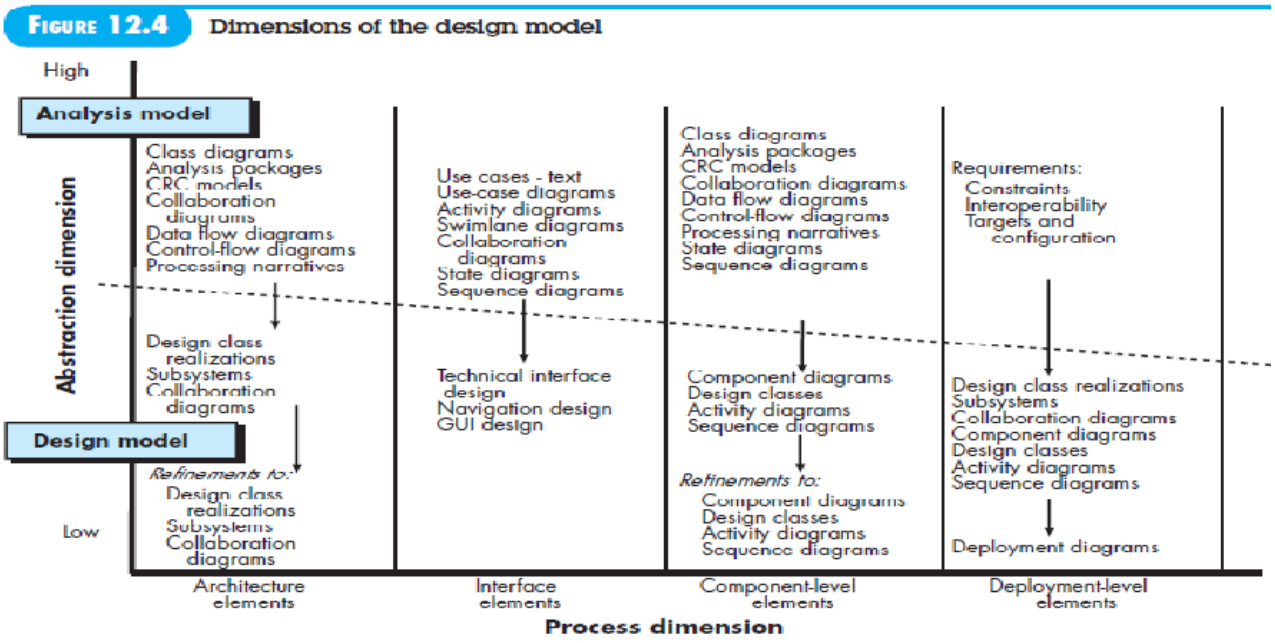
- Your aim in the initial stage of product design should be to create a list of features that define your product
- Features should be
  - Independent
    - Features should not depend on how other system features are implemented

- Coherent
  - Features should be linked to a single item of functionality.
- Relevant
  - Features should reflect the way that users normally carry out some task.

Architectural Design

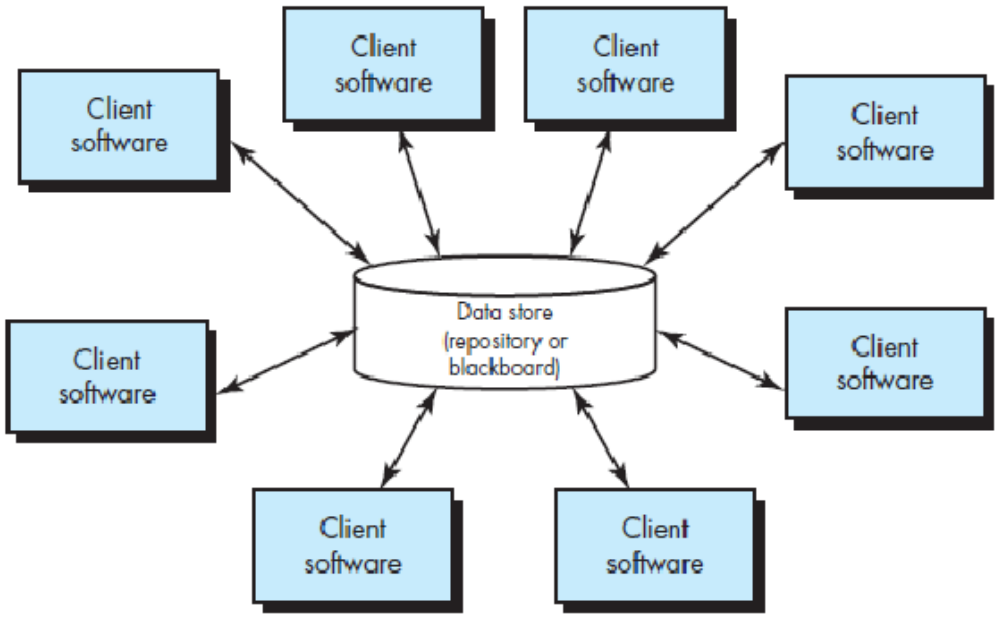
Design Model

- The design model can be viewed in two different dimensions
  - The process dimension indicates the evolution of the design model
  - The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively

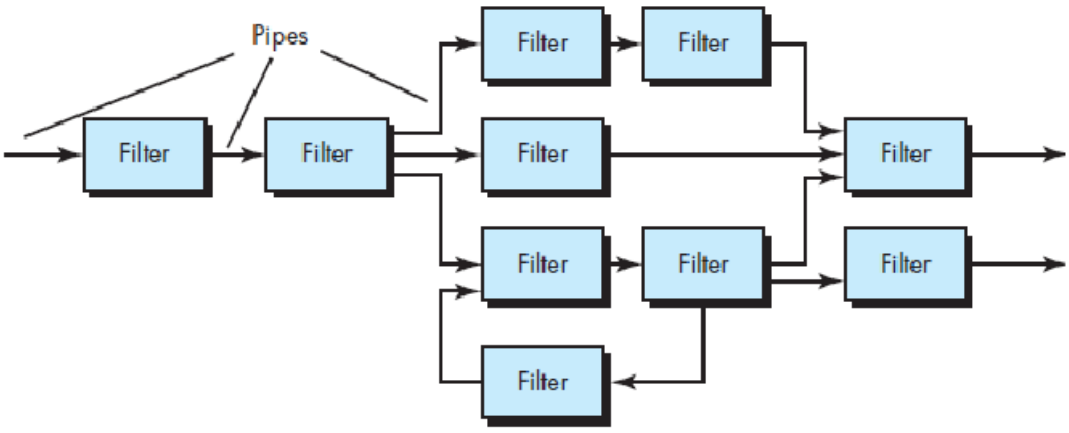


Architectural Styles

- An architectural style is a transformation that is imposed on the design of an entire system
- Data Centred Architecture
  - A data store resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store



- Data Flow Architecture
  - This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data
  - This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.



Pipes and filters

- Call and return architecture
  - This architectural style enables you to achieve a program structure that is relatively easy to modify and scale.

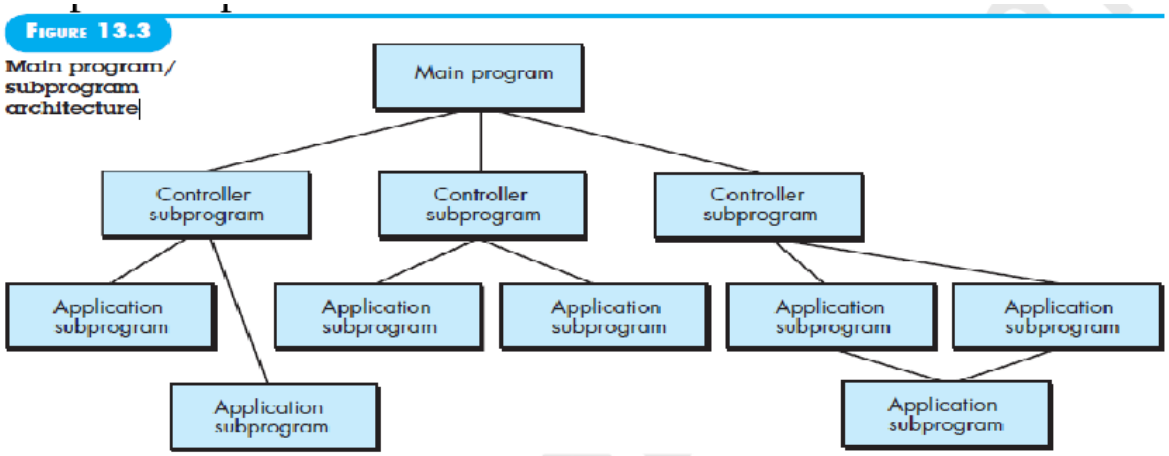
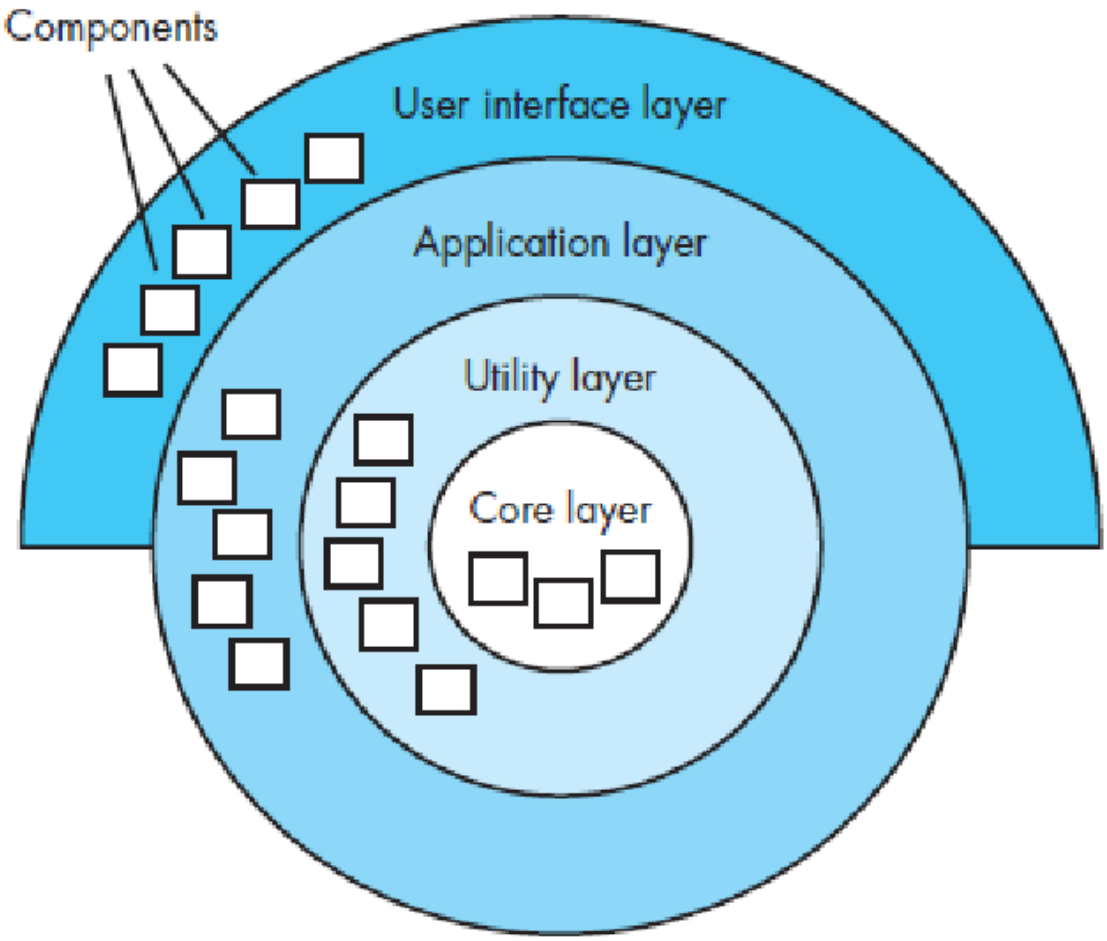


FIGURE 13.3 Main program/subprogram architecture

- Object Oriented Architecture:
  - The components of a system encapsulate data and the operations that must be applied to manipulate the data
- Layered Architecture:
  - A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
  - At the outer layer, components service user interface operations.
  - At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.



Designing Class-Based Components

- Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied
- 1. Open Closed Principle (OCP)
  - A module should be open for extension but closed for modification
- 2. The Liskov Substitution Principle (LSP).
  - component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead.
- 3. Dependency Inversion Principle (DIP)
  - Depend on abstractions. Do not depend on concretions
- 4. The Interface Segregation Principle (ISP)
  - Many client-specific interfaces are better than one general purpose interface

Conducting Component level design

- **Step 1: Identify problem domain classes**
  - Look at the main issues or things we need to solve. Identify the classes (groups of things) related to those problems.
- **Step 2: Identify infrastructure domain classes**
  - Think about the basic structure or framework needed for our solution. Identify classes related to how everything will work together.
- **Step 3: Work on non-reusable classes**
  - Take a closer look at classes that can't be reused from somewhere else. Add more details to them, making them fit our specific solution.
- **Step 4: Deal with data sources**
  - Think about where we store information, like in databases or files. Identify the classes needed to handle and manage that data.
- **Step 5: Describe how classes behave**
  - Think about what each class or component should do. Describe their behavior and functions in more detail.
- **Step 6: Work on deployment diagrams**
  - Imagine how our solution will be implemented in the real world. Draw diagrams to show where each part will be used or deployed.
- **Step 7: Keep improving and considering alternatives**
  - Regularly check and improve the designs. If there's a better way to do something, consider it. Always be open to different options.