1 Introduction

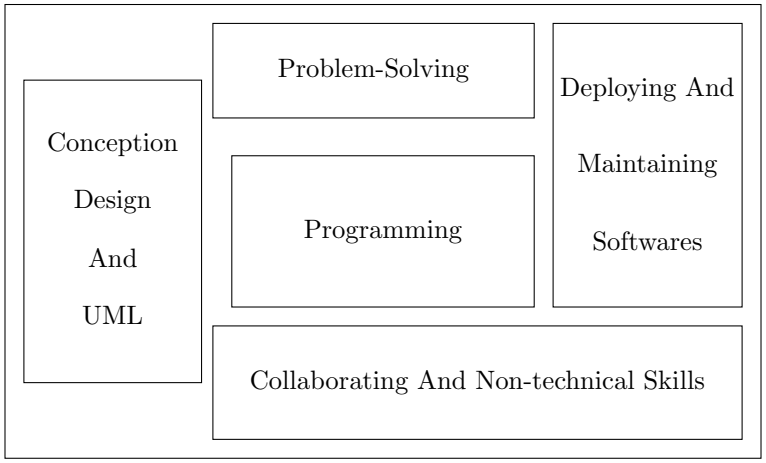
1.1 What's Software Engineering?

Software Engineering is a branch in the field of Computer Science that revolves around designing, testing, coding, and maintaining high-quality software while being able to understand the needs of the client.

1.2 What's the Difference Between a Programmer and a Software Engineer?

People often confuse a programmer with a software engineer. While it is true that both write code for software, a software engineer is a step ahead of the programmer. Not only does the engineer program, but they also possess non-technical skills that allow them to understand and guide the client according to their needs, create the architecture of the software, and apply various techniques and methodologies for well-structured software. In other words, a programmer $\subset$ software engineer.

Software Enginner's Skill



Difference Between UML & SE

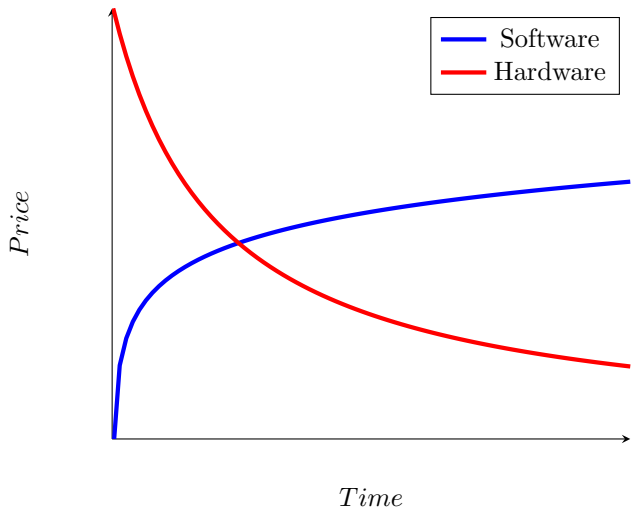
SE $\not\Leftrightarrow$ UML

Many people tend to confuse UML, which is merely a tool used for visualizing client needs, software architectures, and other elements in easy-to-understand diagrams, with software engineering, which involves the entire project creation process . UML is simply one of many skills within the field of software engineering.

2 The Birth of Software Engineering

2.1 Development of Hardware/Software and Their Relationship

- **1940-1950: Vacuum Tubes**: Vacuum tubes were electronic components that controlled the flow of electricity in a vacuum. Used in radios, televisions, and early computers, they were large, expensive, and consumed a lot of power, but were essential for electronics at the time.
- **1950-1960: Transistors**: Transistors are semiconductor devices used to amplify or switch electronic signals. They replaced vacuum tubes due to their smaller size, greater efficiency, and lower cost. Transistors were used in radios, early computers, and have since become fundamental in all modern electronics.
- **1960-1970: Integrated Circuits**: An integrated circuit (IC) is a small chip that contains a set of electronic circuits, including transistors, resistors, and capacitors. ICs made electronics more compact, affordable, and powerful, revolutionizing industries during the 1960s. ICs were found in computers, calculators, and a wide range of other electronic devices.
- **1970-Present: Microprocessors**: A microprocessor is a single-chip CPU (Central Processing Unit) that performs the functions of a computer's central processor. Microprocessors became more affordable in the 1970s, paving the way for the rise of personal computers, smartphones, and embedded systems in countless devices today.

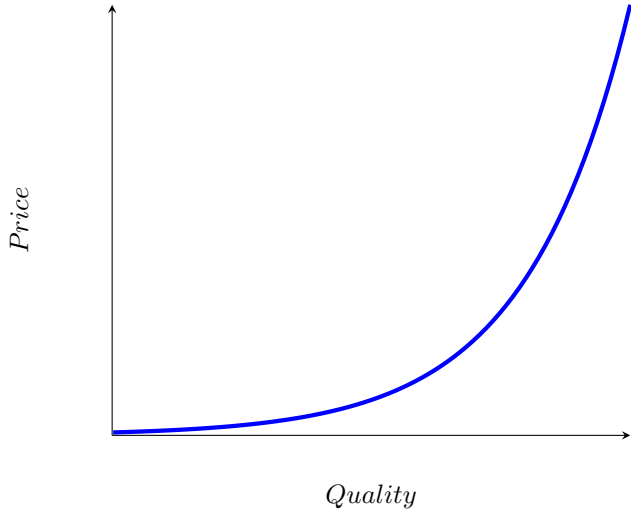


As shown in the graph above, there is an inverse relationship between hardware and software prices. As hardware becomes more affordable and advanced, software prices tend to increase, a phenomenon often referred to as the "Software Crisis." This was primarily caused by a lack of development methodology and reflection on part of the developers. Software projects frequently went over budget, were late to release, and were often filled with bugs.
To address this issue, developers needed to establish efficient methodologies and practices for building better software.

2.2 Qualities of Good Software

A well-designed software product must meet several key criteria to be considered high-quality. Here are some essential qualities:

- **Maintainability**: The ease with which software can be modified to correct faults, improve performance, or adapt to a changed environment. Code should be well-documented, modular, and easy to update without causing regressions in other parts of the software.
- **Affordability**: High-quality software should provide value for money. The cost of development, licensing, maintenance, and support should be reasonable and aligned with the customer's budget and expectations. Affordability doesn't mean the cheapest option, but one that balances cost with features and reliability.
- **Functionality**: The software must meet the specific needs of its users and perform the tasks it was designed for effectively. It should deliver all required features and capabilities, ensuring that it fulfills its purpose.
- **Reliability**: Software should function consistently and accurately over time, without crashing or producing incorrect results. It should handle errors gracefully and work well under various conditions, including edge cases and unexpected inputs.
- **Performance**: Good software should operate efficiently, with fast response times and minimal use of system resources. Performance includes aspects like load times, memory usage, and the ability to handle multiple tasks or large amounts of data simultaneously.
- **Scalability**: The software should be able to grow and accommodate increasing amounts of work or users without a significant drop in performance. This is especially important for systems that expect long-term growth or fluctuating demand.
- **Usability**: The user interface should be intuitive, making it easy for users to learn and navigate. Well-designed software provides a positive user experience, with thoughtful layouts, clear instructions, and accessible features for all users, including those with disabilities.
- **Security**: Security is critical in today's environment. High-quality software should protect user data and prevent unauthorized access, ensuring compliance with security standards. It should include features like encryption, authentication, and protection against common vulnerabilities such as SQL injection or cross-site scripting.
- **Portability**: Software should work across different platforms and environments with minimal modifications. This is especially important for applications intended to run on multiple operating systems, browsers, or devices. A portable software solution can save significant development effort.
- **Interoperability**: High-quality software should be able to integrate and work seamlessly with other systems and tools. This includes support for standard formats, APIs, and protocols that allow it to exchange data and collaborate with other software.
- **Minimization of Bugs**: A good software product should have a minimal number of defects. This involves rigorous testing during the development process to catch and fix bugs early, as well as ongoing maintenance and updates to address issues as they arise.
- **Extensibility**: Software should be designed in a way that allows new features and functionality to be added in the future without significant changes to the core architecture. Extensible software can adapt to changing user needs and technology advances.
- **Compliance with Standards**: High-quality software adheres to industry standards and regulations, ensuring that it meets legal, technical, and ethical guidelines. This is particularly important in industries like healthcare, finance, and government, where compliance is mandatory.



This graph illustrates the positive correlation between software quality and price. As the quality of the software improves, its price tends to increase exponentially.

3 Life Cycle Of Software

3.1 Whats's Life Cycle Of Software?

As the name suggest it's the life cycle of a software from start to finish , life cycle also known as model is a methodology that serves to help developers in building their product , each life cycle has steps and a chronology to follow

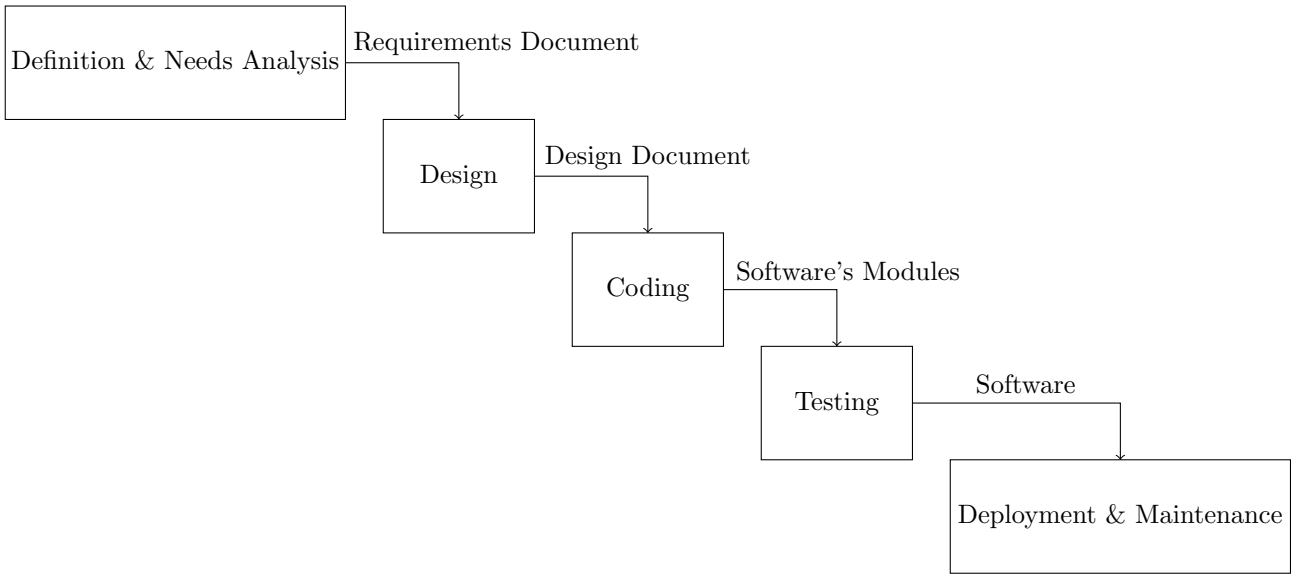
3.2 Waterfall Life Cycle

The Waterfall Model is considered one of the first software development methodologies for large-scale projects. As the name suggests, it follows a series of sequential steps, one after the other, similar to a waterfall. This model formed the basis of early software life cycles, solving many issues but also introducing some new challenges. Over time, other life cycle models were developed and improved upon.

3.2.1 First Version

The first version of the WaterFall model includes 5 steps

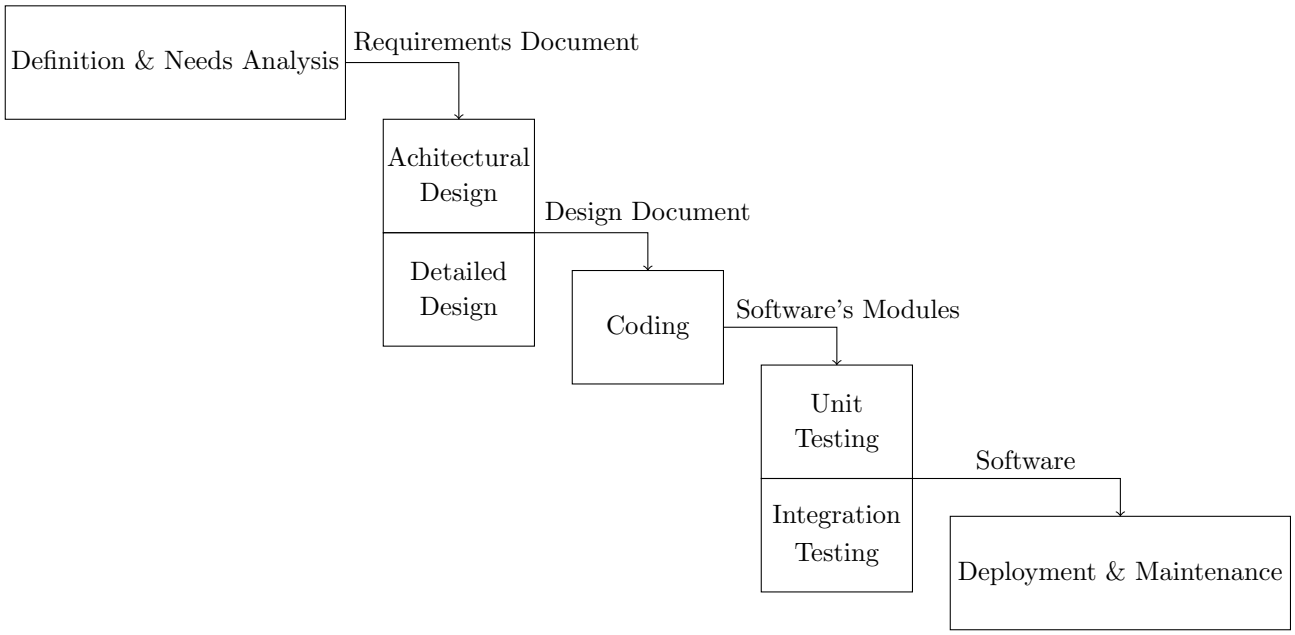
- **Definition & Needs Analysis**: The first step involves interacting with the client to understand their requirements and guide them. This step is crucial because any misunderstanding between the developers and the client could cause the entire project to fail, requiring a restart. The output of this step is a document that contains all the necessary information, called the "Requirements Document."
- **Design**: In this step, developers create the software's architecture, defining the modules and how they interact with each other. The output of this step is the "Design Document."
- **Coding**: The implementation of the modules into a programming language takes place in this step.
- **Testing**: This step involves testing the software and fixing bugs.
- **Deployment & Maintenance**: The final step is delivering the software to the client and maintaining it by adding new features and fixing undetected and future bugs.



3.2.2 Improved Version

The Waterfall model was slightly improved by dividing some steps into two:

- **Design**:
  - **Architectural Design**: Design how the software's modules will interact with each other.
  - **Detailed Design**: Design each module's components in detail.
- **Testing**:
  - **Unit Testing**: Test all the modules independently.
  - **Integration Testing**: Test the interaction and communication between modules.



Note

Why Architectural Design before Detailed Design?

We first define the high-level structure of the modules and their interactions to provide an overall system architecture. This gives a clear overview of the software before delving into the detailed characteristics of each module.

Why Unit Testing before Integration Testing?

It is important to test each module individually to ensure that they function correctly in isolation. This helps to simplify debugging, as any issues can be pinpointed within a module, rather than confusion over whether the problem lies in the module itself or in the interaction between modules.

3.2.3 Pros

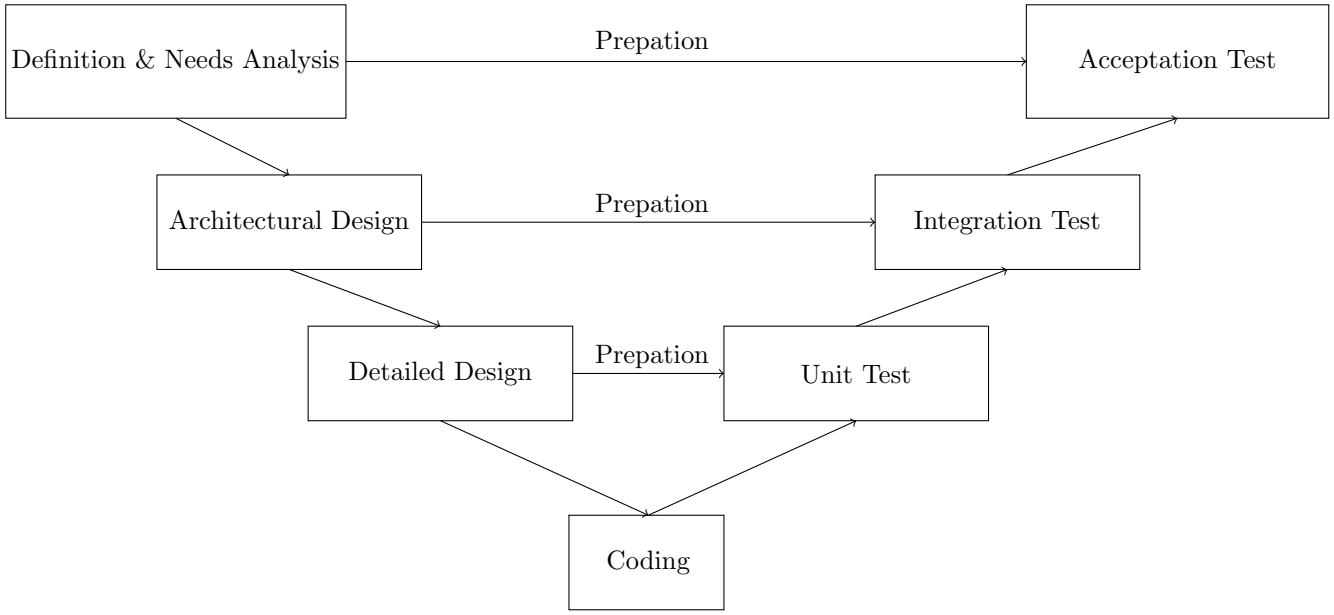
- Helps organize the project by providing a clear structure and well-defined stages.
- Simple to understand and execute, especially for small to medium-sized projects.
- Easy to manage due to its linear progression, making it suitable for projects with stable requirements.
- Documentation is thorough, ensuring that each phase has a clear output.

3.2.4 Cons

- Lack of parallelism, as tasks cannot be done concurrently, leading to longer project timelines.
- High dependency between steps, meaning each phase must be completed fully before the next can begin.
- Inflexibility to handle changes during development, as going back to previous phases can be costly and time-consuming.
- Risk of significant rework if the needs analysis phase is not done properly, potentially requiring a restart from the beginning.
- Limited feedback during development, as testing and validation occur only at later stages.

3.3 V Life Cycle

The V Model is similar to the Waterfall Model, as both follow a sequential process with the same key steps. However, the V Model introduces a key difference: it integrates preparation for the corresponding testing phases alongside each development step, forming a "V" shape in the process diagram. For example, after completing the **Architectural Design**, preparation for **Integration Testing** begins. Similarly, after the **Detailed Design**, preparation for **Unit Testing** takes place. This parallel preparation ensures better alignment between development phases and their corresponding testing phases, saving time and helping to catch potential issues earlier.



Note

It's important to note that while there is parallelism in the preparation for testing, the development steps themselves are still executed sequentially, with no overlap.

3.3.1 Pros

- Improves time efficiency by preparing testing phases in parallel with development phases, allowing for faster transitions between steps.
- Allows early detection of potential issues, as test preparation begins immediately after design, enabling a quicker response to design flaws or misunderstandings.

3.3.2 Cons

- Lacks parallelism in the execution of development steps, as each phase must still be completed sequentially.
- Carries the risk of significant rework if the needs analysis phase is not thoroughly completed, potentially requiring a restart from the beginning.
- Maintains strong dependencies between steps, meaning that each phase relies heavily on the correct execution of the previous one. This can create delays or issues if earlier phases were not executed properly.

3.4 Prototyping Life Cycle

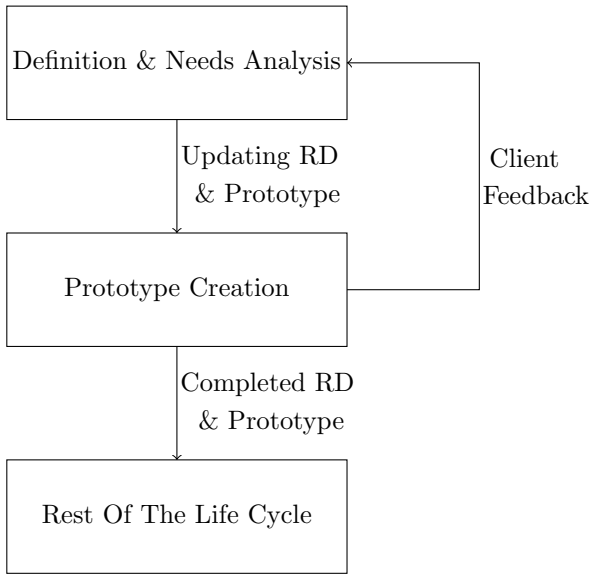
The Prototyping Life Cycle was developed to reduce the risk of restarting the project due to incomplete, contradictory, or ambiguous requirements in the initial requirement document. In this approach, the development team first meets with the client to conduct a needs analysis and create the initial requirement document. Then, a small prototype is developed and presented to the client for feedback. This process helps clarify the client's needs, improve understanding, and refine the requirements document. The cycle of developing and reviewing prototypes continues until the requirements document is fully defined and clear. Once finalized, the project proceeds through the remaining steps of the chosen life cycle model.

Note

Should We Discard the Prototype or Build Upon It?

The decision depends on several factors. If the prototype was quickly assembled and lacks scalability or if adding new features proves to be complex due to a poorly thought-out design, it might be more efficient to discard the prototype and start fresh.

However, if the prototype was designed with a solid foundation and can easily accommodate additional features, continuing to build on it could save time and resources.



3.4.1 Pros

- Reduces the risk of restarting the project due to continuous client feedback, as a complete requirements document is established upfront.

3.4.2 Cons

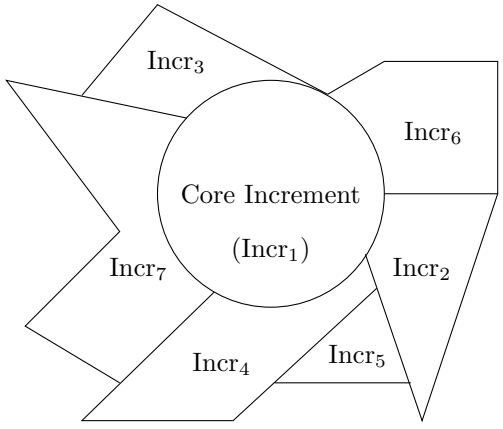
- Creates uncertainty about whether to discard the prototype or continue building on it, which can be time-consuming and resource-intensive.
- Carries the cons of the used life cycle

3.5 Incremental Life Cycle

The Incremental Life Cycle is an iterative model designed to address the uncertainty of whether to discard the prototype or continue building on it. In this model, development starts with the implementation of a core set of essential features, known as the core increment or the first increment. This initial increment is delivered to the user for feedback. Subsequent increments, each containing additional features, are built on top of the core increment and are also sent to the client for review. This process continues iteratively, with feedback shaping the development, until the project is fully completed.

Note

It is crucial to carefully select the first set of features when implementing the core increment. If this initial set is not scalable, it will be difficult for developers to add new features and continue building upon it effectively.



3.5.1 Pros

- Dividing the project into smaller sets of features makes it easier for developers to manage and focus on one increment at a time.
- Constant client feedback throughout the development process ensures that adjustments can be made as new features are added.
- Minimizes the risk of wasted effort, as nothing is discarded—each increment builds upon the previous one.

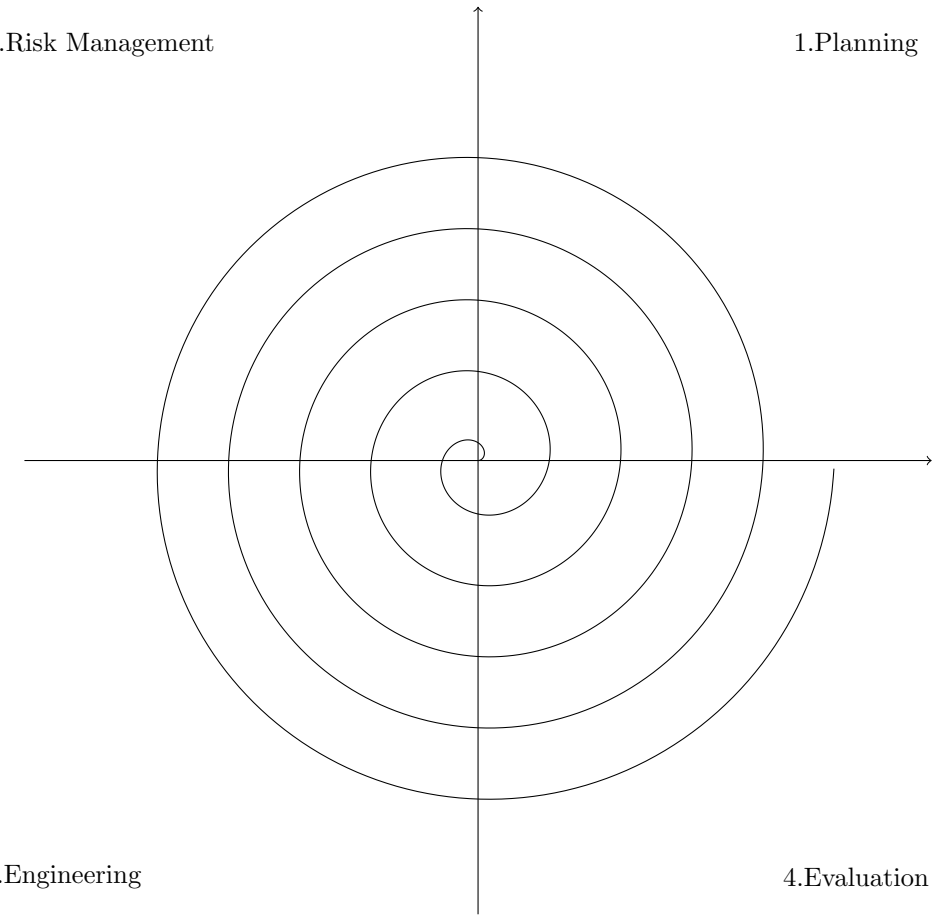
3.5.2 Cons

- There is a risk of restarting the project if the developers do not select the right initial set of features for the core increment, as future increments depend on this foundation.
- Each increment inherits the flaws or limitations of the chosen life cycle model.

3.6 Spiral Life Cycle

The Spiral Life Cycle is an iterative model designed for large-scale projects, with a strong focus on risk assessment and mitigation throughout the development process. It is called "spiral" because it involves multiple phases arranged in a spiral pattern, with each loop in the spiral representing a phase of the software development process (an iteration).

- **Planning(Objective)**: The project's objectives are identified, and alternatives and constraints are defined. High-level requirements gathering (Needs Analysis) is also performed.
- **Risk Analysis(Risk Management)**: This phase focuses on evaluating risks, uncertainties, and potential problems of the current iteration, ensuring that risks and strategies to mitigate them are identified early on.
- **Engineering (Development & Testing)**: The actual designing, development, and testing of the product occur in this phase, which includes activities such as unit testing and integration testing.
- **Evaluation(Planning next itteration)**: After each iteration (loop) of the spiral, there's an evaluation of the product by the customer or stakeholders. Feedback is collected, and necessary changes are made before the next loop begins, allowing for continuous improvement throughout the project.



3.6.1 Pros

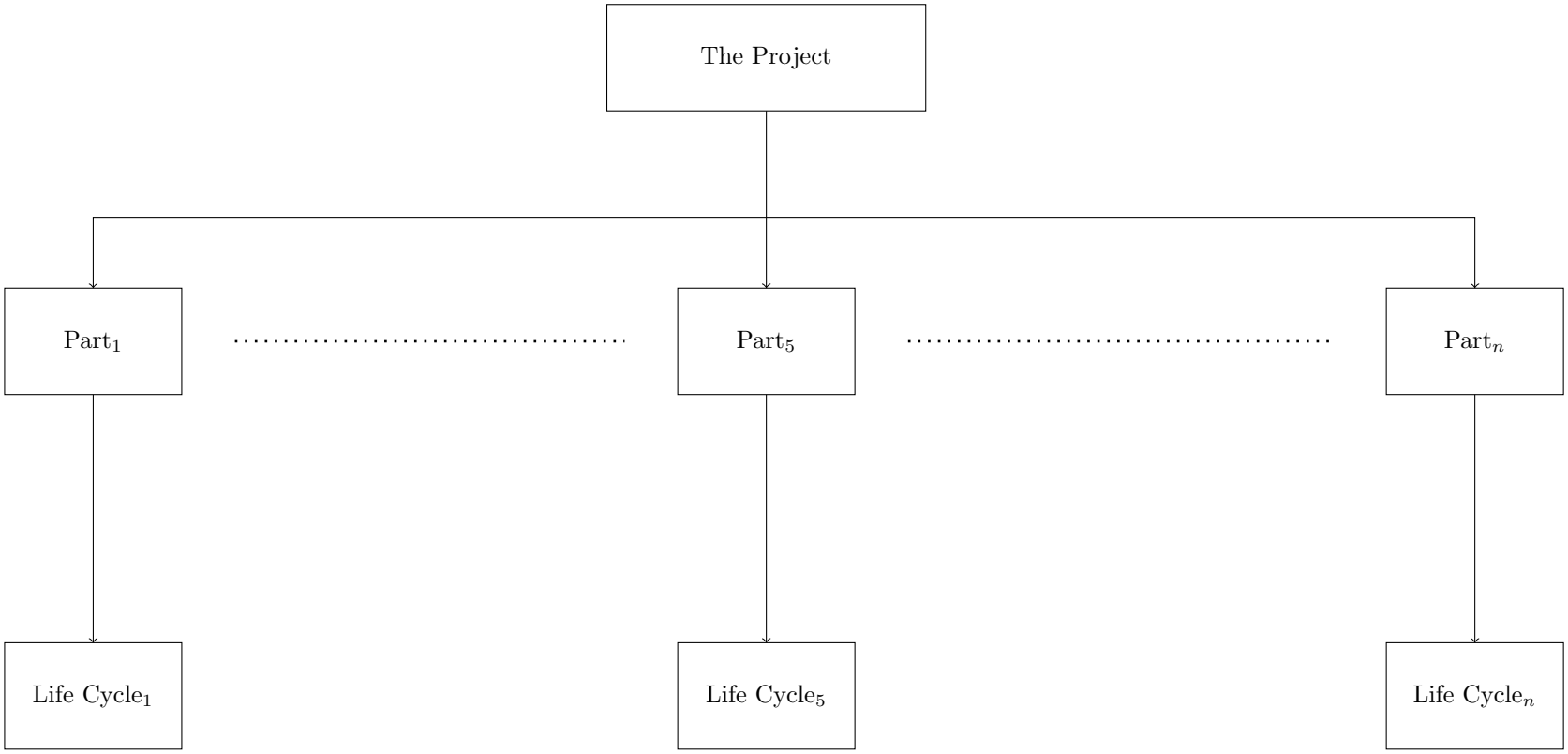
- Continuous client feedback facilitates timely adjustments and improvements to the project.
- Regular identification and assessment of risks, along with strategies for mitigation, help ensure a smoother development process.

3.6.2 Cons

- The model can be complex and challenging to implement effectively, requiring careful planning and coordination.
- It can be resource-intensive, potentially demanding significant time and effort from the development team.

3.7 Hybrid Life Cycle

The Hybrid Life Cycle involves dividing the project into n parts, where each part_i follows its corresponding lifeCycle_i .This model provides the greatest flexibility among all the methodologies discussed so far.



Note

Difference Between Incremental & Spiral & Hybrid ?

- Hybrid: This model divides the project into n parts, each with its own life cycle. One of these life cycles can be Incremental or Spiral.
- Incremental: This approach breaks the project into a set of features called increments, building each subsequent increment on the core increment.
- Spiral: The Spiral model follows an iterative loop structure, where each loop consists of four steps: Planning, Risk Analysis, Development & Testing, and Evaluation.

3.7.1 Pros

- Offers significant flexibility since each part can adopt a different life cycle model tailored to its specific needs and requirements, enabling teams to capitalize on the strengths of various methodologies.

3.7.2 Cons

- Can be complex to manage, as coordinating different life cycles for various parts of the project may require additional resources and effort.
- Potential for misalignment between parts if communication and integration aren't carefully managed, which could lead to inconsistencies in the overall project.

Note

Difference Between The Life Cycles ?
All the life cycles we've reviewed so far include similar steps (activities), the differences arise in the logical and chronological sequencing of these activities.

4 Needs Analysis

A thorough needs analysis is critical in software development. Any ambiguity, contradiction, or missing information found in the requirements document can lead to significant setbacks, potentially requiring a restart of the project. In this section, we will take a detailed look at how the requirements document is structured.

Note

Difference Between Goals & Needs ?

It is crucial to understand that the requirements document holds the client's needs and not their goals . A goal is subjective and open to interpretation, while a need is objective—measurable or verifiable.

Example :

The client might request a "pleasant user interface" , This can be interpreted in many ways:

- a visually appealing UI with lots of colors and animations
- an easy-to-use interface
- a minimalistic design

and so on. To turn this into a clear need, we must clarify what the client means. For instance, they might want the UI to be organized with all features accessible through a dropdown menu.

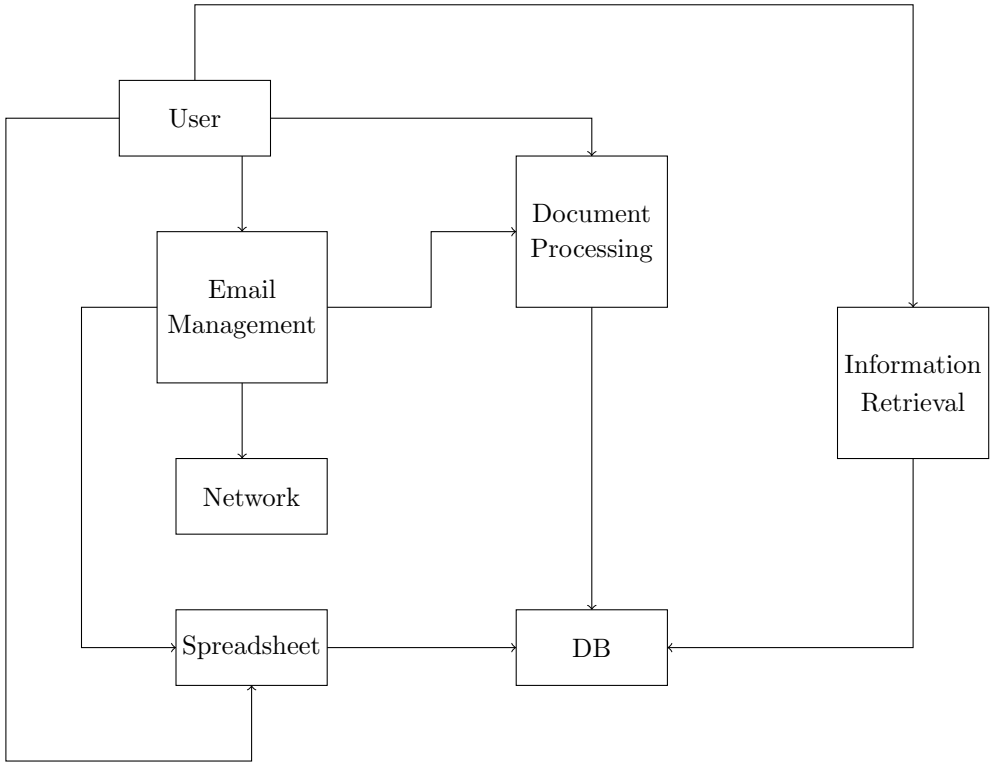
4.1 Requirments Document Structure

- **Introduction :** This section provides an overview of the document by outlining its key components :
  - Purpose: The reason for creating this document, which is primarily to align the development team and facilitate communication with the client.
  - Scope: A high-level summary of the software's main functionality, without going into too much detail.
  - Context: The reason for creating the software, which could be to sell it to a client or company, to develop an open-source project, or other similar motivations.

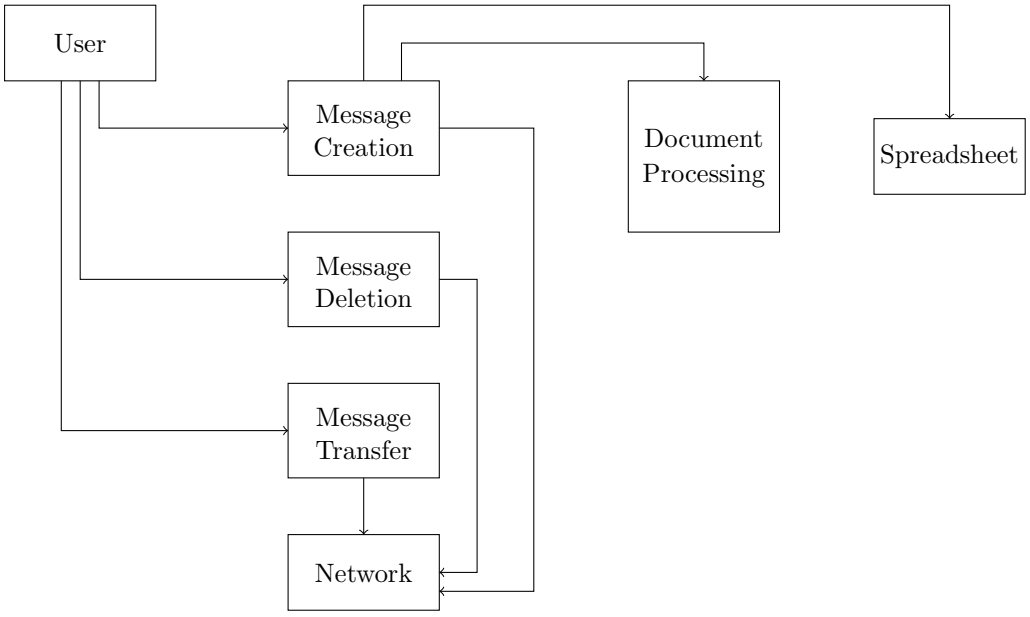
- **Hardware :** This section states whether the software requires any special hardware components like : GPU, sensors, or a camera ...etc . It also outlines the minimum hardware requirements needed to run the software, as well as the optimal hardware configuration for the best and smoothest experience.
- **Conceptual Model :** This section describes the overall software architecture through a high-level graphical representation, highlighting key components and their relationships. It provides an overview of how the system is structured and operates, making it easier for stakeholders to understand.

  Example :

  A desktop system composed of an email service, a spreadsheet, a document processing service, and an information retrieval service.



The next step consist into creating a new conceptual model for each complexe fonction



- **Functional Requirements:** These define what the system should do, focusing on the specific features and functions the software must deliver to meet user or business needs. They describe the expected behavior of the system in various situations. Functional requirements are concrete and measurable. They can be expressed using natural, semi-formal, or formal language, or a mix of these.
  - **Natural Language:** Easy to implement and understand, but lacks structure and precision, which can lead to ambiguity. It makes automating analysis of the document harder, relying heavily on the writer's linguistic experience.
  - **Structured (Semi-Formal) Language:** Limited use of natural language, more structured and precise than natural language , often accompanied by graphical notations.
  - **Formal Language:** Hard to master and time-consuming to implement. It is difficult for clients to understand but is based on mathematical theory, making it the most precise language and easier to automate verification.
- **Non-Functional Requirements:** Define the restrictions and constraints related to both hardware and software within the context of the ongoing project. Non-functional requirements are particularly influenced by changes in technologies (both hardware and software) and are crucial for complex software systems. As the project develops, changes in hardware may occur. These changes can be anticipated by projecting the expected performance levels that will be required by the end of the project.
- **Maintenance Information:** Anticipates possible actions after the software's initial release, such as adding new features, improving performance, or addressing potential issues.
- **Glossary:** Provides definitions of the terms and concepts used in the document to help readers. This ensures that the terminology is clear, as the requirements document is shared and read by the design team, developers, and stakeholders, without assuming prior knowledge of these terms.
- **Index:** Helps the reader find specific topics and sections of the document more efficiently by providing a detailed list of references to relevant sections, parts, and page numbers.

Note

Functional and non-functional requirements are inherently connected, and their interplay can sometimes lead to conflicts. By understanding the potential for these conflicts and establishing a process for managing them, development teams can better balance user needs and system performance, ultimately leading to a more successful product.

4.2 Requirements Validation

The requirements need to be coherent, realizable, and complete. Anticipation of hardware needs to be considered. It is crucial for the requirements document to be validated in order to initiate the next steps of the software life cycle.

- review Technique is an efficient way to monitor and update the requirements .
- There are various analysis tools available that can facilitate the validation process of the requirements document, helping to ensure accuracy and completeness.

5 UML

5.1 Introduction

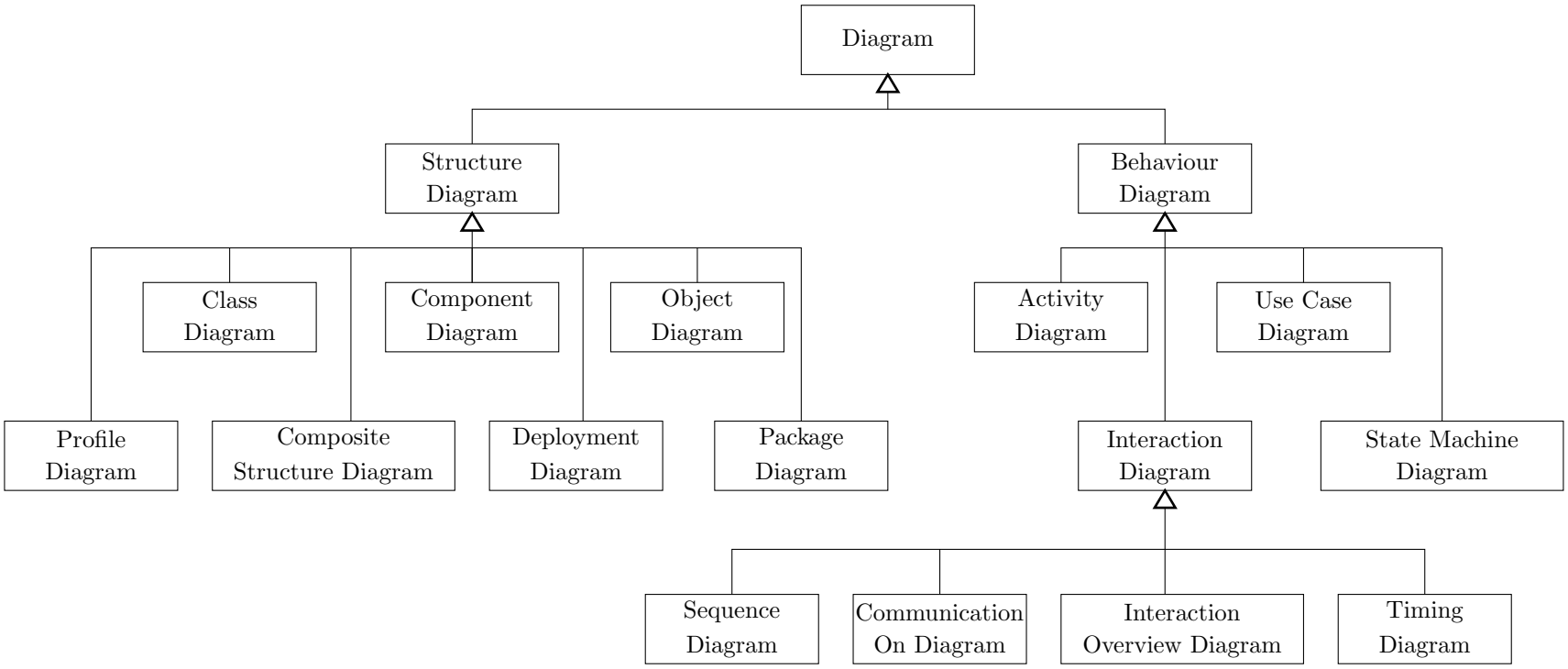
As seen in the previous sections, the requirements document is very important, as any mistake, ambiguity, or misunderstanding of the client's needs can lead to a project restart. We've reviewed the high-level structure of a requirements document; one of the most important parts is the functional requirements, which can be expressed in natural language. This approach is easy to implement and understand but isn't very precise, doesn't automate verification, and relies solely on the linguistic experience of the writer. There is also a formal approach based on mathematical theory, which automates verification, is very precise, but is hard to master and time-consuming. Finally, there is a semi-formal approach that benefits from the advantages of both natural and formal language; an example of this is UML (Unified Modeling Language).

5.2 UML(Unified Modeling Language)

UML is a semi-formal language that is easy to understand and widely used by developers. It divides into two main categories of diagrams:

- Structure Diagram : Represent the static aspects of a system, showing its classes, objects, and relationships.
- Behaviour Diagram : Describes the dynamic aspects of a system, illustrating how actors interact with the system and how the system changes over time.

In this section we will focus on Use case diagram

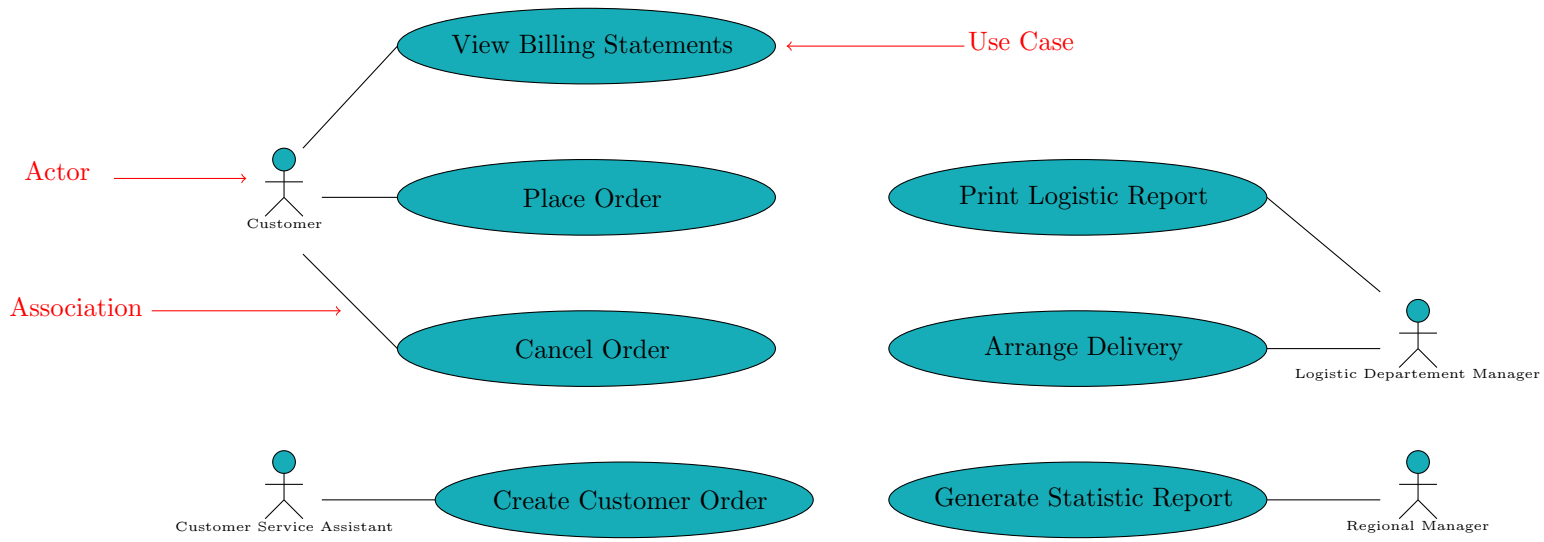


5.3 Use Case Diagram

A use case diagram is a graphical representation of a behavior diagram, composed of three main elements:

- Actors: Actors represent roles that interact with the system. Each actor performs a set of actions called use cases, which represent features of the software they interact with. Actors are depicted as stick figures and can be either human or non-human.
  - Human actors: Individuals using the system.
  - Non-human actors: Entities such as hardware (e.g., sensors, cameras) or software (e.g., APIs, dependencies).
- Use Cases (Features): These represent features or functions of the software and are shown as ellipses , they have to be programmable.
- Associations: Associations are the connections between actors and use cases, depicted as lines. These lines indicate what action (use case) each actor can perform.

Example:



Note

Internal & External Actors :

In a use case diagram all actors have to be internal to the software and all use cases must be features from the software that are programmable

- Internal Actor : must interact with the software
- External Actor : don't interact with the software

Note

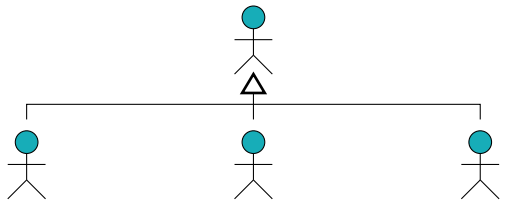
The Ideal Level Of Decomposition Of Use Cases :

In a use case diagram, the level of decomposition for use cases should strike a balance. It shouldn't be too specific, as this can make the diagram cluttered and difficult to read, nor too generic, which can make it hard to understand. A moderate level of detail is ideal.

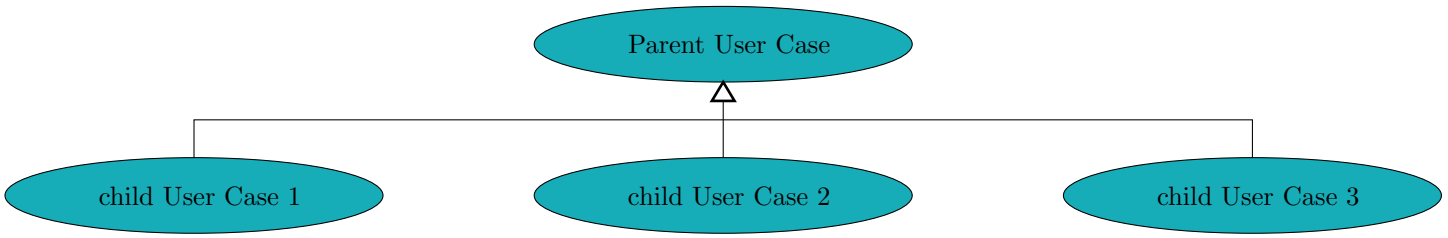
It's acceptable to leave some use cases at a more generic level without decomposing them, especially since the diagram is accompanied by a document that provides detailed explanations. This document defines and elaborates on all sub-use cases as needed.

5.3.1 Relations In Use Case Diagram

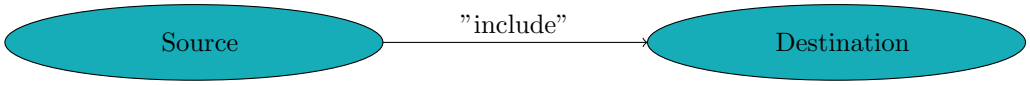
- **Generalization**: Generalization establishes a hierarchical relationship that helps organize actors and use cases in UML diagrams.
  - **Between Actors**: When a child actor is linked to a parent actor, it inherits all the roles and responsibilities of the parent actor,in addition to its own specific roles. This helps clarify who can perform what actions in the system.



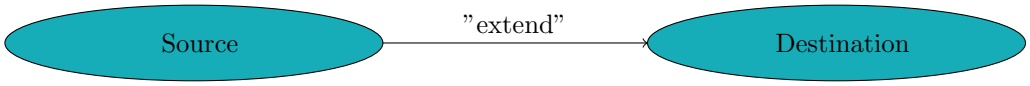
  - **Between Use Cases**: Allows a feature to be divided into multiple, more specific versions. These child use cases represent the same general feature but can differ in their implementation or method. This promotes reusability and clarity in how features are handled in different scenarios.



- **Inclusion**: This relationship applies only between use cases. The destination use case must execute before the source use case whenever the source use case is invoked.



- **Extension**: This relationship applies only between use cases. Before executing the destination use case, a specific condition is checked. If the condition is met, the source use case will execute before the destination use case.

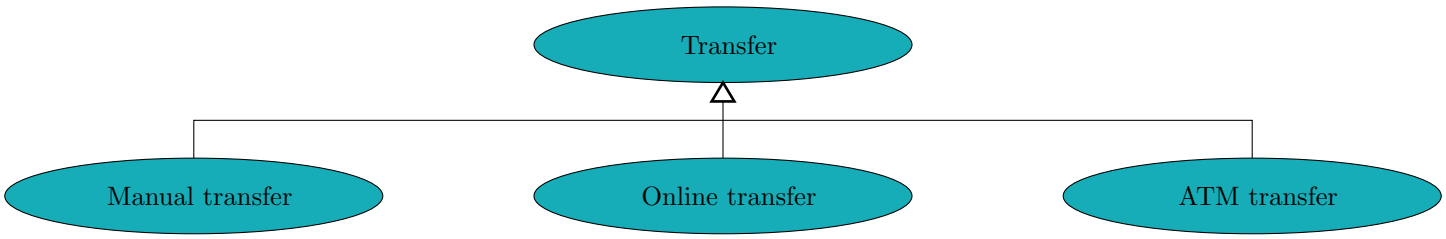


Example:

Let's Take the example of a bank application

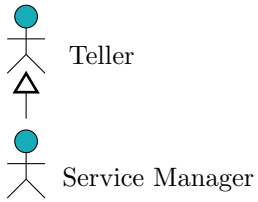
Generalisation Between Use Cases:

the diagram below shows a generalisation relationship between use case where the main feature is transfert that is divided into specific method and way of achieving the transfert (ATM,Manual,Online)



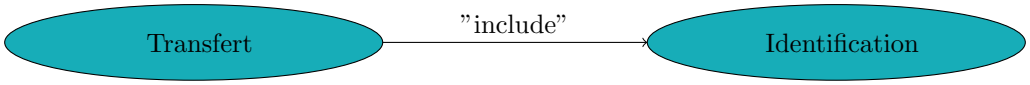
Generalisation Between Actors:

the diagram below shows a generalisation relationship between Actors where the child actor Service Manager is inheriting all roles of the parent actor teller + his own roles



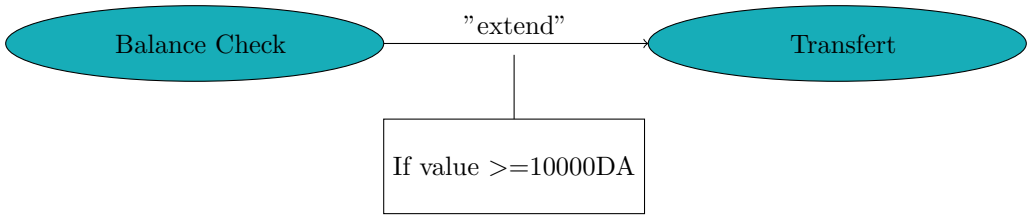
inclusion Between Use Cases:

the diagram below shows an inclusion relationship between use cases where each time the transfert case is invoked the identification case must execute first



Extension Between Use Cases:

the diagram below shows an extension relationship between use cases where each time the transfert case is invoked the balance check case will be executed if the value is >= 10000 DA



Note

Difference Between Inclusion & Extension :

In The inclusion relation the Destination case will always execute before source case in all cases , whereas in the extension relation the source case will execute before destination source only if certain conditions are met

Extends With No Conditions :

If an extend relation doesn't have an explicit condition it just means that it's an option and the actor can decide wether to trigger the extended use case or not (option)