1 Introduction

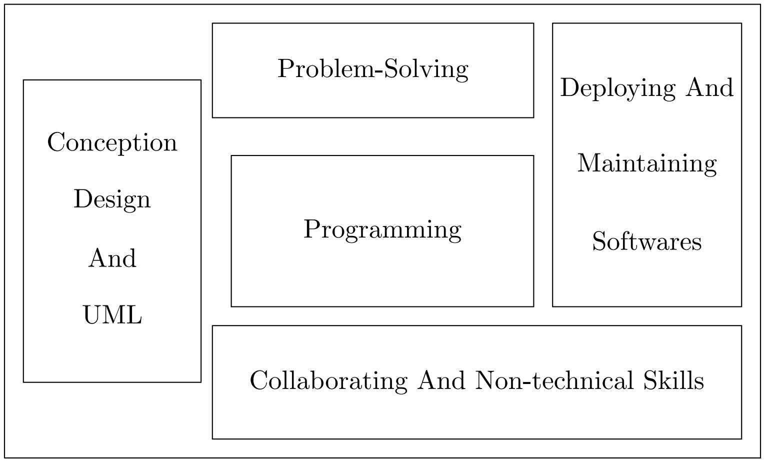
1.1 What's Software Engineering?

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


Conception Design And UML
Problem-Solving
Programming
Deploying And Maintaining Softwares
Collaborating And Non-technical Skills

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

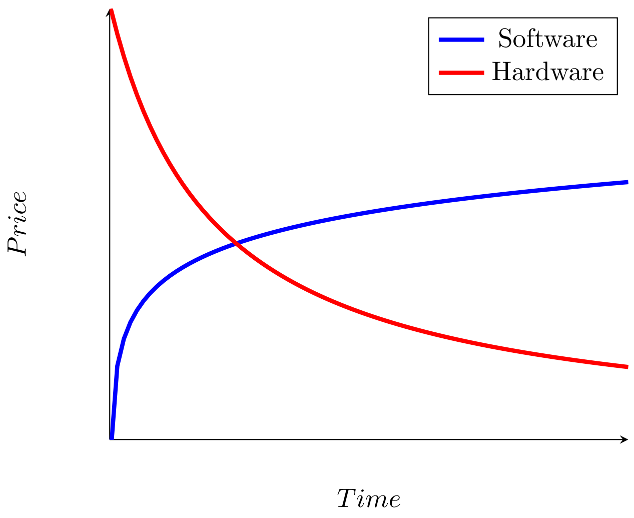
2.1 Development of Hardware/Software and Their Relationship

- **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.



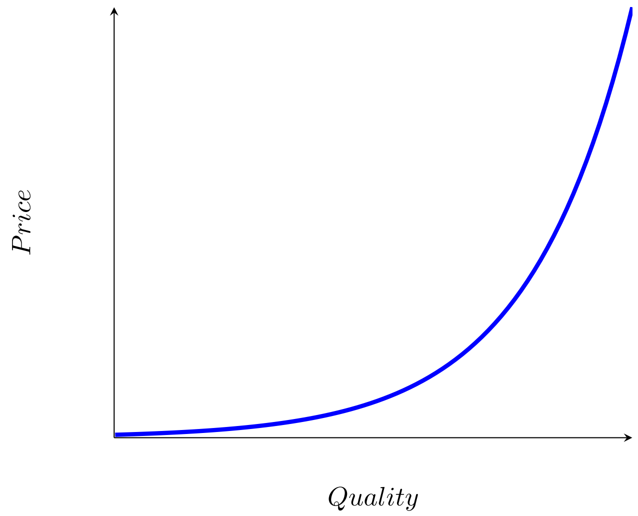
As shown in the graph above, there is an inverse relationship between hardware and software prices. As hardware becomes more affordable and advanced, software prices tend to increase, a phenomenon often referred to as the "Software Crisis." This was primarily caused by a lack of development methodology 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

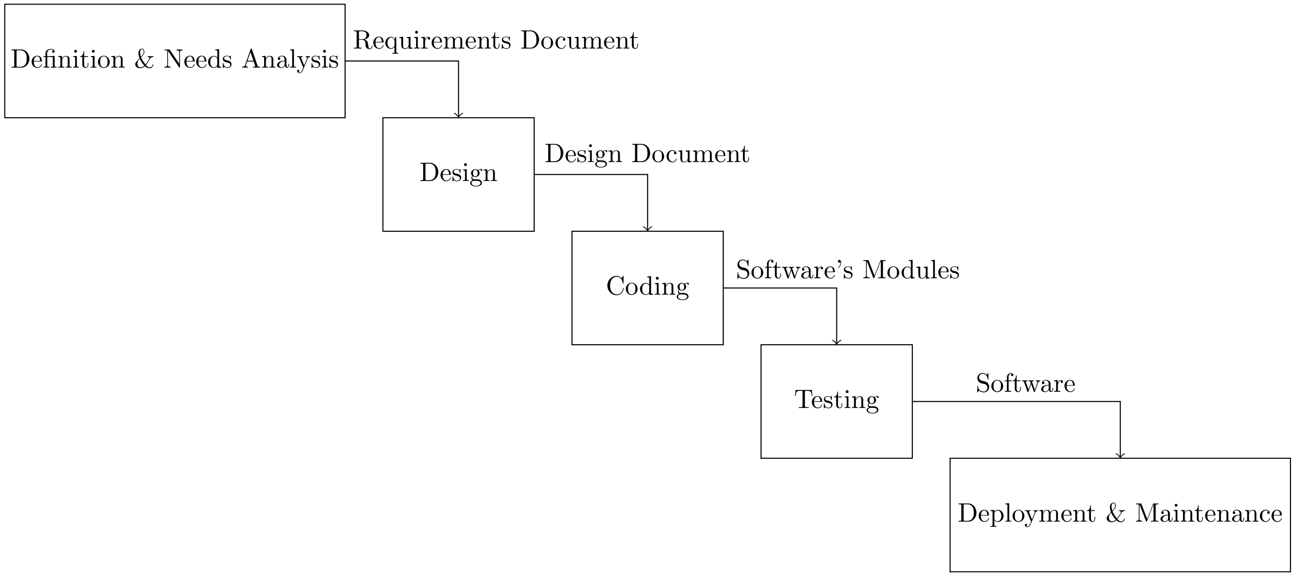
3.2 Waterfall Life Cycle

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

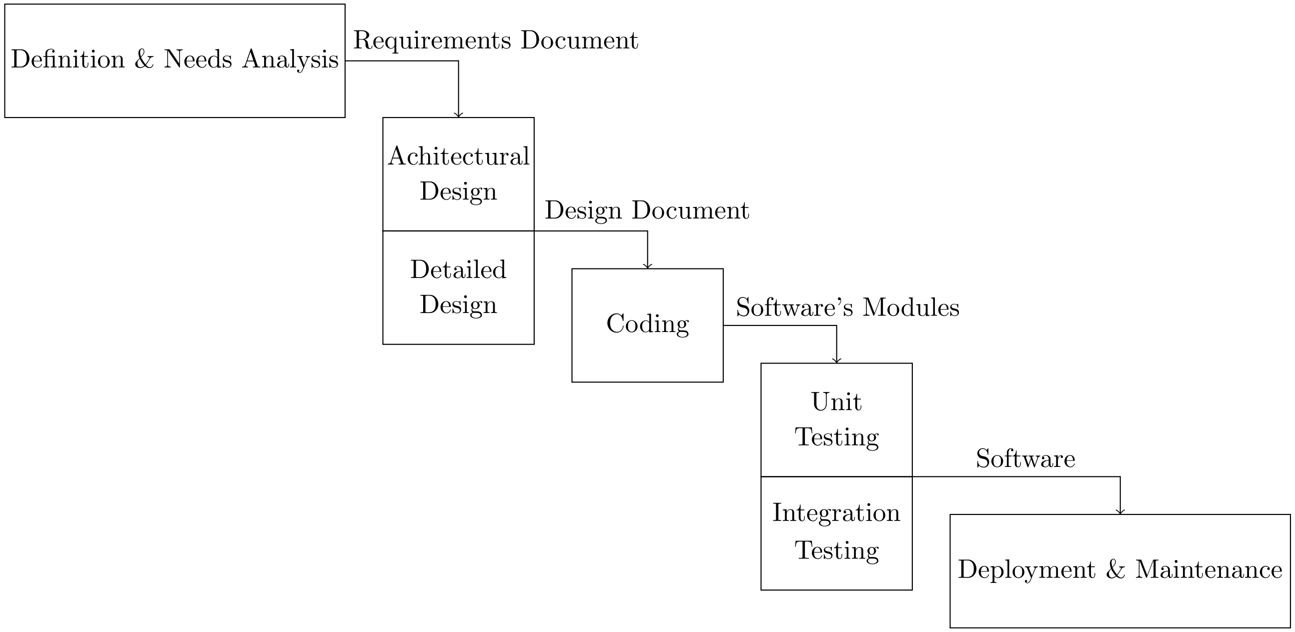
- **Definition & Needs Analysis**: The first step involves interacting with the client to understand their requirements and guide them. This step is crucial because any misunderstanding between the developers and the client could cause the entire project to fail, requiring a restart. The output of this 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

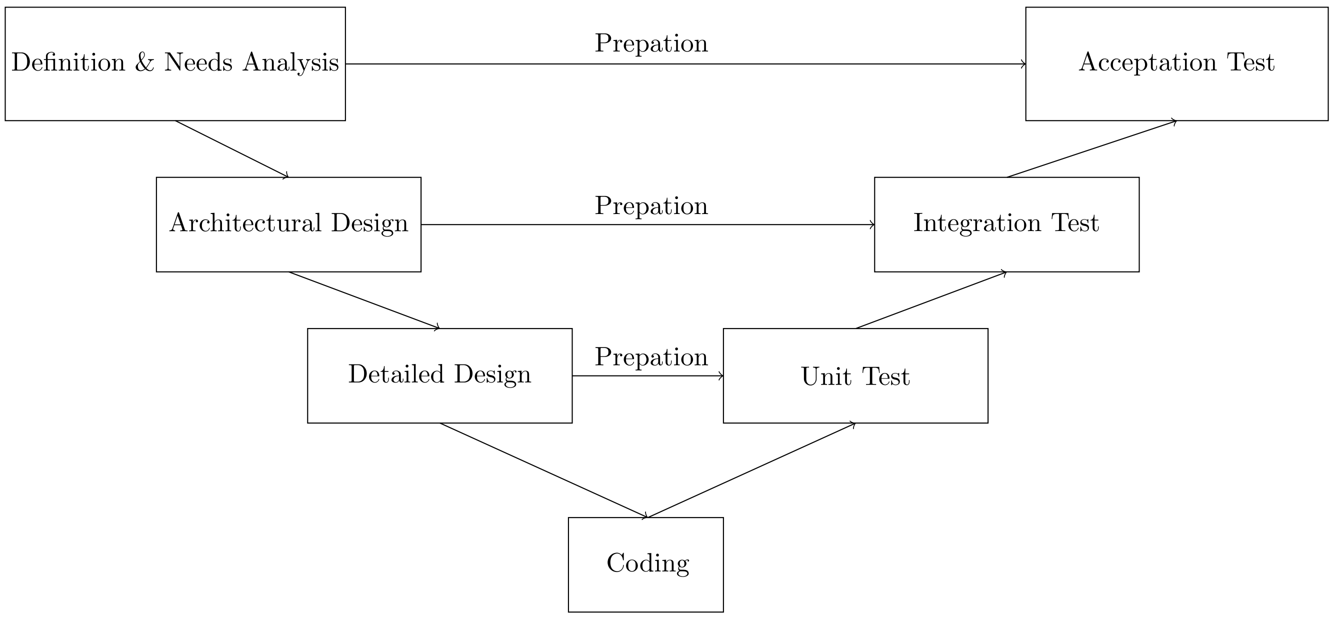
- Helps organize the project by providing a clear structure and well-defined stages.
- Simple to understand and execute, especially for small to medium-sized projects.
- Easy to manage due to its linear progression, making it suitable for projects with stable requirements.
- Documentation is thorough,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

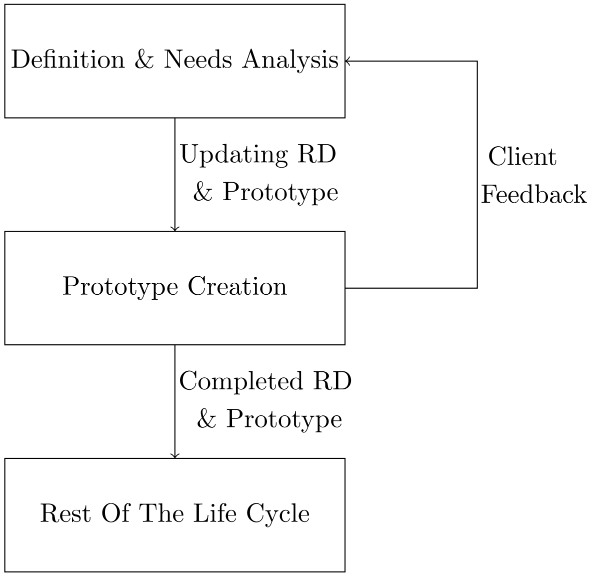
3.4 Prototyping Life Cycle

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.

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

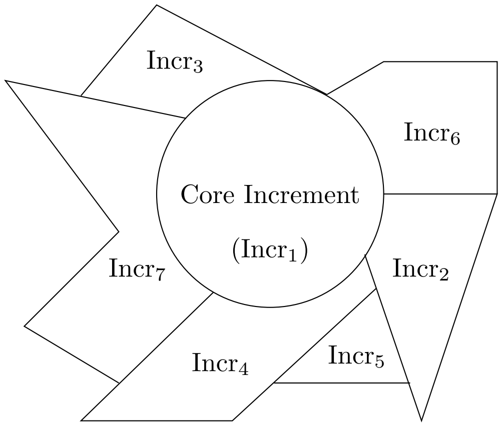
- Creates uncertainty about whether to discard the prototype or continue building on it, which can be time-consuming and resource-intensive.
- Carries the cons of the used life cycle

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.

Importance Of Selecting The Core Features

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.

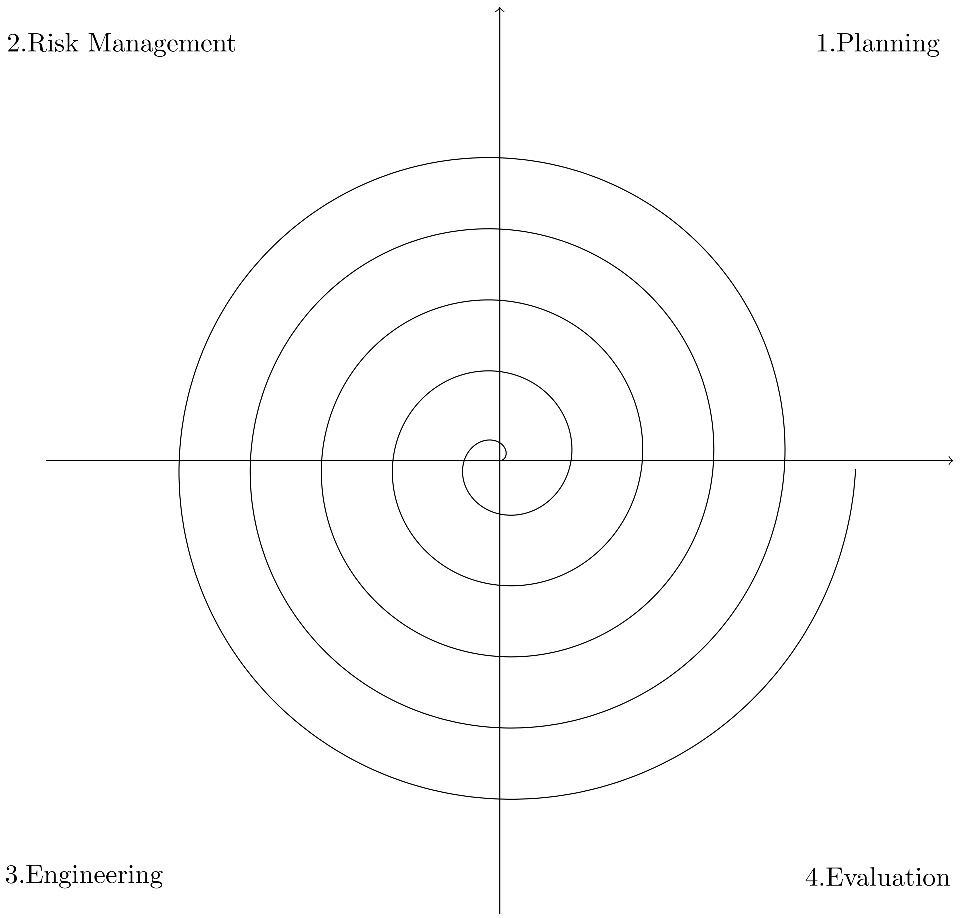
3.5.2 Cons

- 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

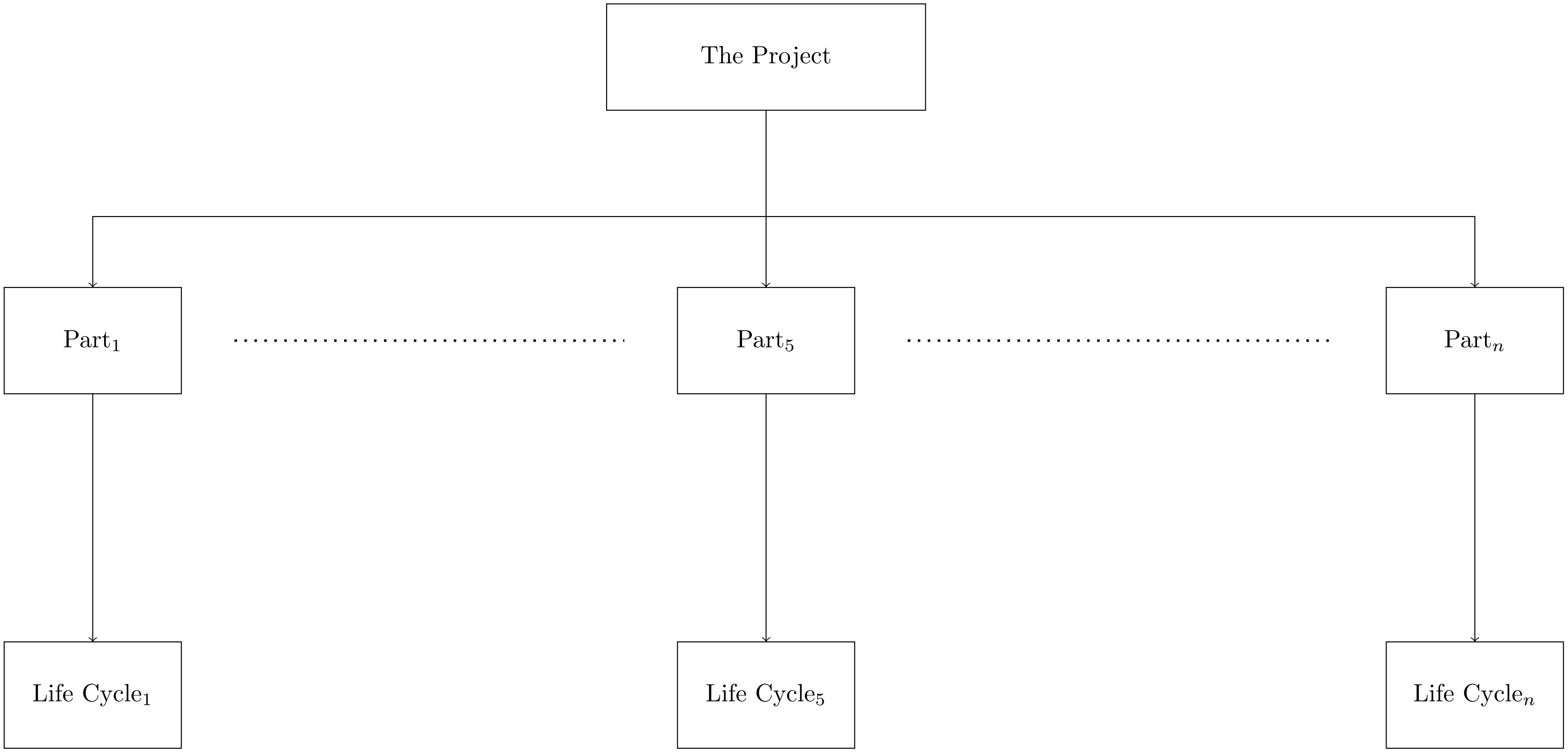
- Continuous client feedback facilitates timely adjustments and improvements to the project.
- Regular identification and assessment of risks, along with strategies for mitigation, help ensure a smoother development process.

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.



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.

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.

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.

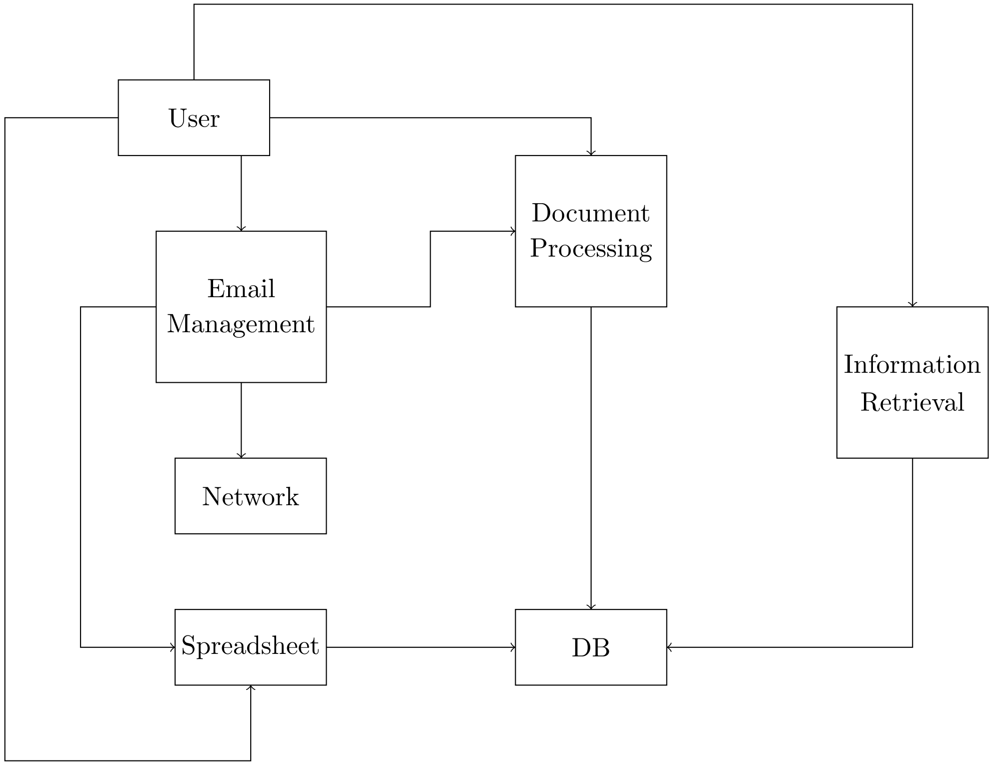
4.1 Requirments Document Structure

- **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.

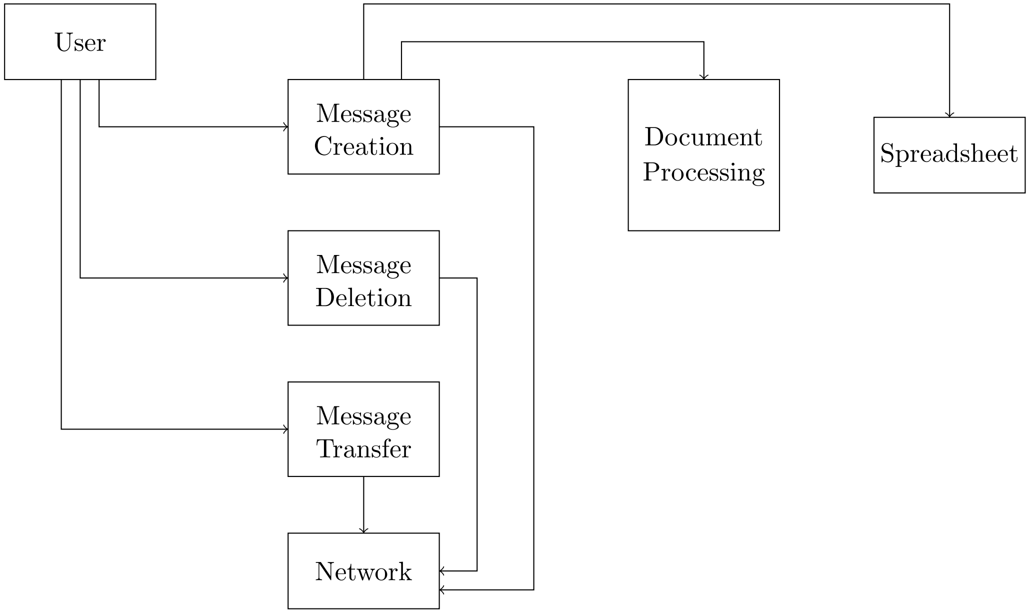
- **Hardware :** This section states whether the software requires any special hardware components like : GPU, sensors, or a camera ...etc . It also outlines the minimum hardware requirements needed to run the software, as well as the optimal hardware configuration for the best and smoothest experience.
- **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.

Relation Between Functional & Non-Functional Requirements

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

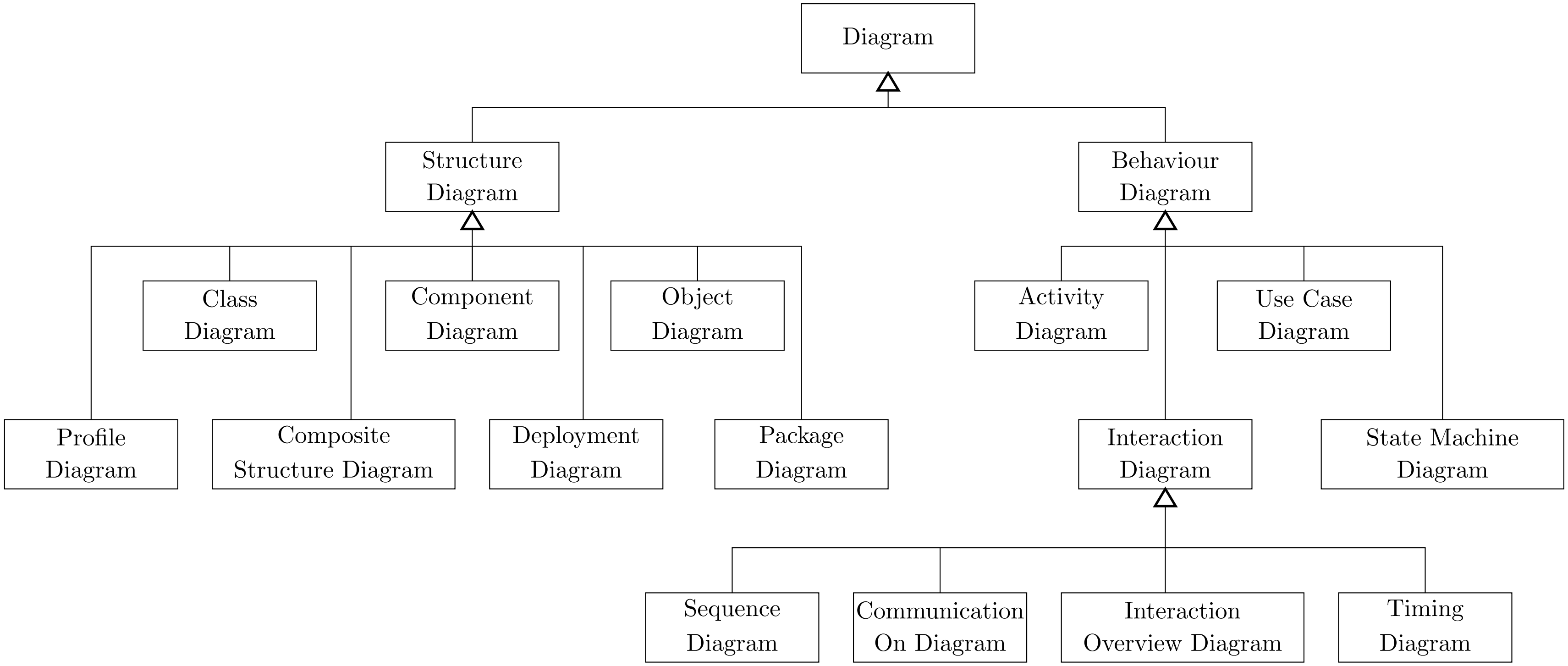
As seen in the previous sections, the requirements document is very important, as any mistake, ambiguity, or misunderstanding of the client's needs can lead to a project restart. We've reviewed the high-level structure of a requirements document; one of the most important parts is the functional requirements, 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

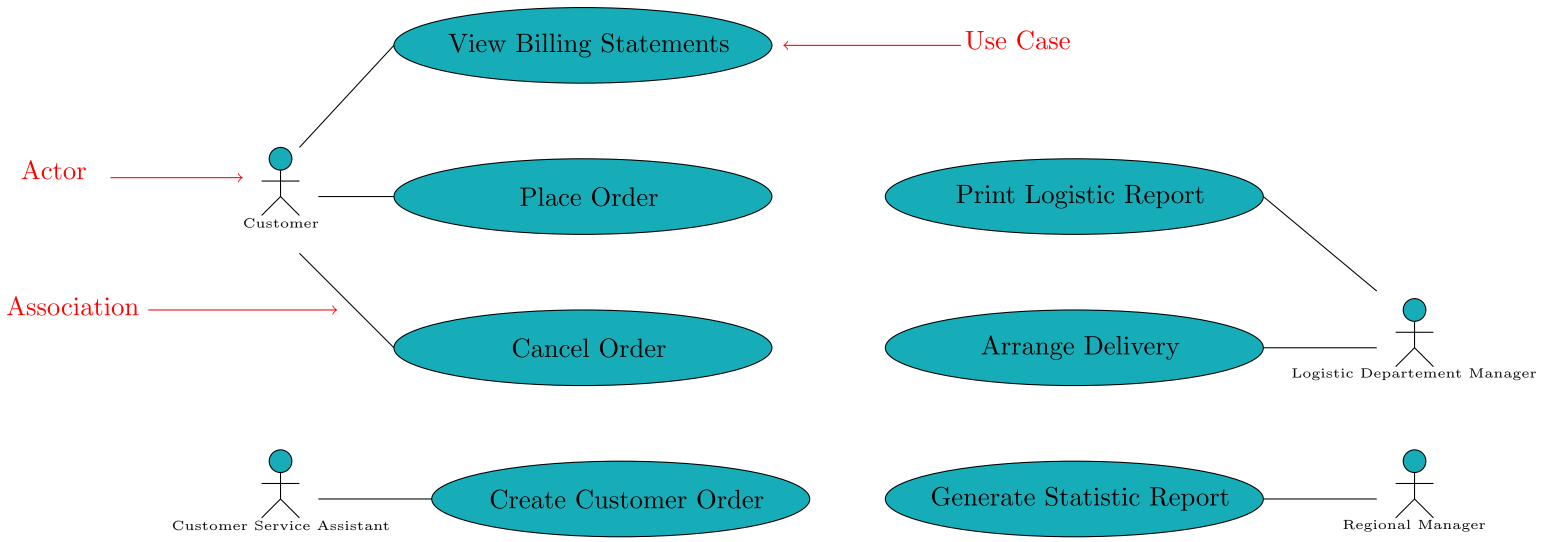


5.3 Use Case Diagram

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:



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

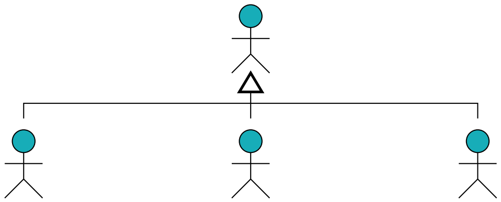
The Ideal Level Of Decomposition Of Use Cases

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.

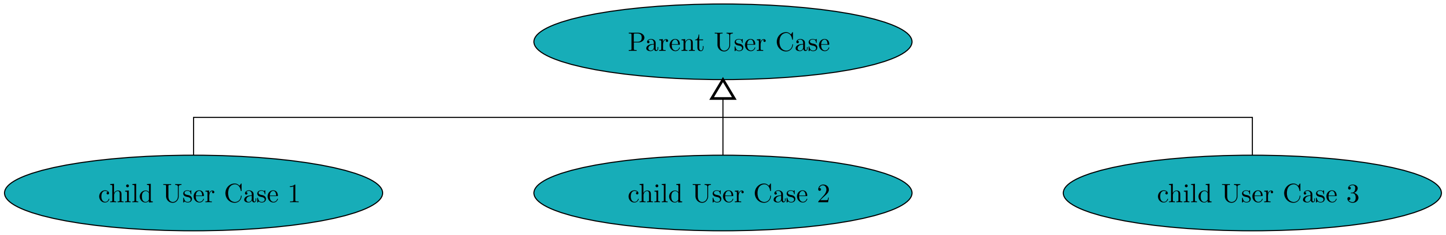
It's acceptable to leave some use cases at a more generic level without decomposing them, especially since the diagram is accompanied by a document that provides detailed explanations. This document defines and elaborates on all sub-use cases as needed.

5.3.1 Relations In Use Case Diagram

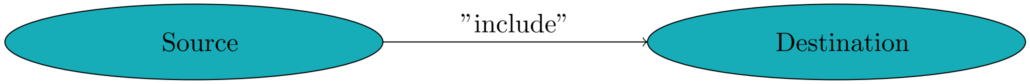
- **Generalization**: Generalization establishes a hierarchical relationship that helps organize actors and use cases in UML diagrams.
  - **Between Actors**: When a child actor is linked to a parent actor, it inherits all the roles and responsibilities of the parent actor,in addition to its own specific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



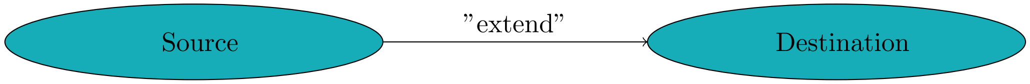
  - **Between Use Cases**: Allows a feature to be divided into multiple, more specific versions. These child use cases represent the same general feature but can differ in their implementation or method. This promotes reusability and clarity in how features are handled in different scenarios.



- **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.

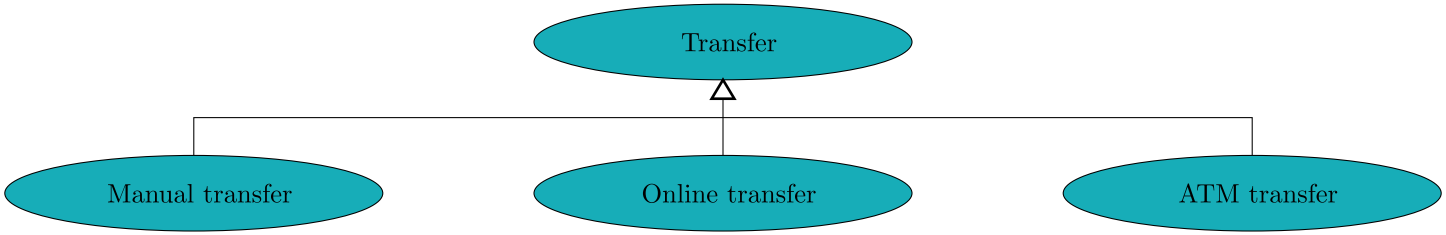


Example:

Let's Take the example of a bank application

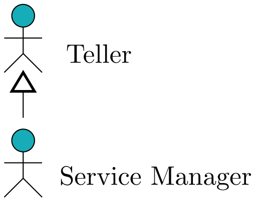
Generalisation Between Use Cases:

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)



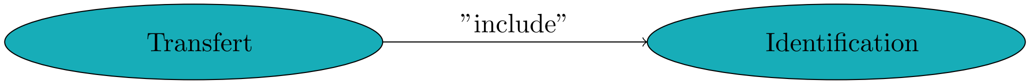
Generalisation Between Actors:

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



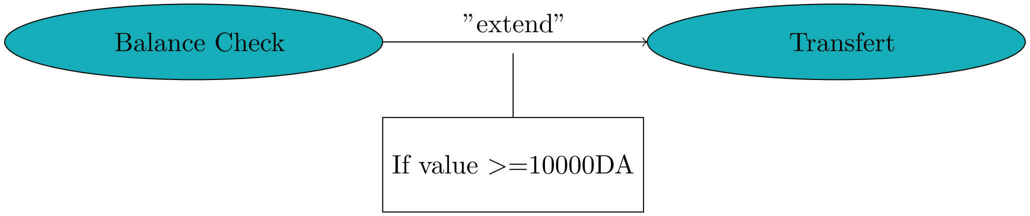
inclusion Between Use Cases:

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



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

5.4 Class Diagram

What's a Class Diagram

A Class Diagram is a graphical representation of a structure diagram that illustrates classes, their attributes, and the relationships between them. It consists of three main components:

- **UML Elements**: Classes , enumeration , interfaces.
- **Relations**: Connections that define how classes interact with each other.
- **Cardinalities**: Numerical indicators of the relationships between classes.

Each component will be explained in detail.

5.4.1 UML Elements

Class

Each class is represented as a rectangle containing the following components:

- **Class Name**: Displayed at the top of the rectangle. It may have an access modifier and other modifiers:
  - `+`: Public
  - `-`: Private
  - `#`: Protected
  - (no symbol): Default (Package-private)
  - `static`: Inner class (indicating that the class can exist without an instance of the outer class)
  - `final`: The class cannot be subclassed (i.e., it is a non-extendable class)
  - `abstract`: The class cannot be instantiated directly (i.e., it is intended to be subclassed)
- **Attributes**: Attributes include both methods and variables of the class. Each method or variable has can have an access modifier and other modifiers:
  - **Variables**:
    - `+`: Public
    - `-`: Private
    - `#`: Protected
    - (no symbol): Default (Package-private)
    - `static`: The variable belongs to the class, not to instances. It is shared by all instances and has the same value for every instance.
    - `final`: The variable can only be assigned a value once and cannot be modified afterward.
  - **Methods**:
    - `+`: Public
    - `-`: Private
    - `#`: Protected
    - (no symbol): Default (Package-private)
    - `static`: The method can be accessed without creating an instance of the class.
    - `final`: The method cannot be overridden by subclasses.
    - `abstract`: The method has no implementation in the class and must be implemented by subclasses.

Enumeration

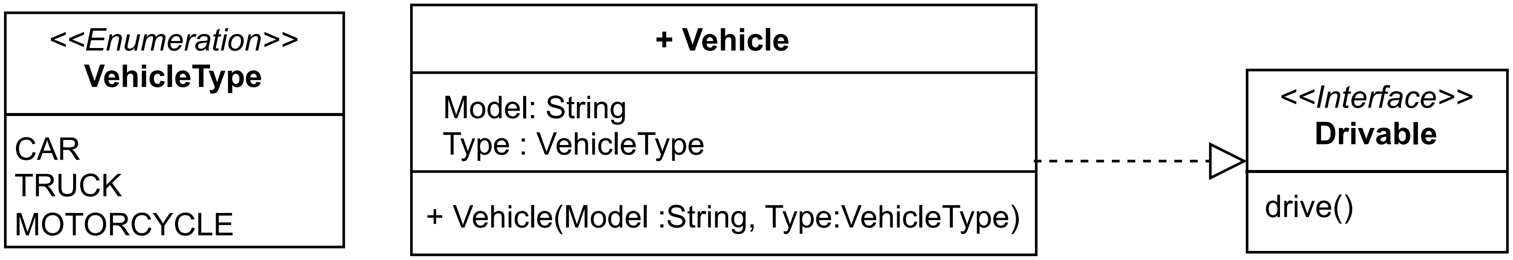
Enumerations are represented in a rectangle as well. The top displays ≪ Enumeration ≫ enumName . Below that, the rectangle holds the possible values of the enum. it is placed next to the class that uses the enum

Interface

Interfaces are represented in a rectangle as well. The top displays ≪ Interface ≫ interfaceName. Below that there are the constant and method prototypes of the interface it is linked to classes that implements it using the realization relation

Example :

UML :



Note

When class implements an interface there is no need to list the implemented methods

Java Code :

Vehicle Class :

```java
public class Vehicle implements Drivable {

String Model;
VehicleType Type;

public Vehicle(String Model,VehicleType Type) {
  this.Model = Model;
  this.Type = Type;
}

@Override
public void drive() {
System.out.println("The " + this.Type.toString().toLowerCase() + " is driving.");
}

}
```

VehicleType Enumeration :

```
public enum VehicleType {
CAR,TRUCK,MOTORCYCLE;
}
```

Drivable Interface :

```
public interface Drivable {
void drive();
}
```

5.4.2 Cardinalities

Cardinality

Cardinality in UML defines the number of instances of one class that can or must be associated with an instance of another class. It helps indicate the number of objects involved in aggregation , association, compostion relation.

How to Read Cardinality:

Cardinality is represented by numbers near a class, The format is typically ‘min..max‘ , so if there is association between student and teacher , cardinality near to teacher would be how many instance of teacher for 1 student , and cardinality near student class would be how many instance of student for 1 teacher

Examples:

- ‘1‘ means **exactly one instance**.
- ‘0..1‘ means **zero or one instance**.
- ‘0..*‘ means **zero or more instances** (unlimited).
- ‘1..*‘ means **one or more instances** (unlimited).

5.4.3 Relations Between Class

Simple Relation(Association)

Association

Association is represented as a solid line between classes. It indicates that one class references another. There are several types of associations:

- **Reflexive Association**: A class references itself.
- **Unidirectional Association**: One class references another class, but the reverse is not true.
- **Bidirectional Association**: Two classes reference each other.
- **N-ary Association**: An association involving more than two classes.
- **Association Class**: The association between classes is created through separate class that references the participating classes and may include additional attributes or behaviors , it is represented by dashed line connecting between association class and solid line association.

Note

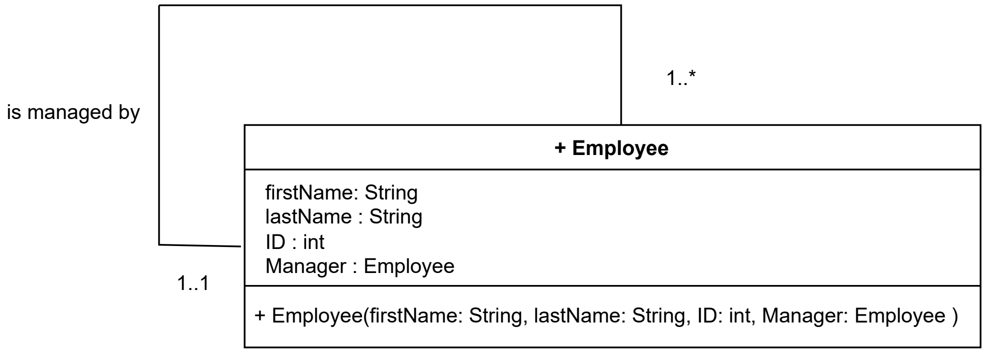
There is no need to explicitly list references of the other classes in the class association because it is implicitly implied

Example :

Reflexive Association

Employee class that references itself to have a supervisor or manager , so an employee is managed by one manager , and a manager manages many employees

UML

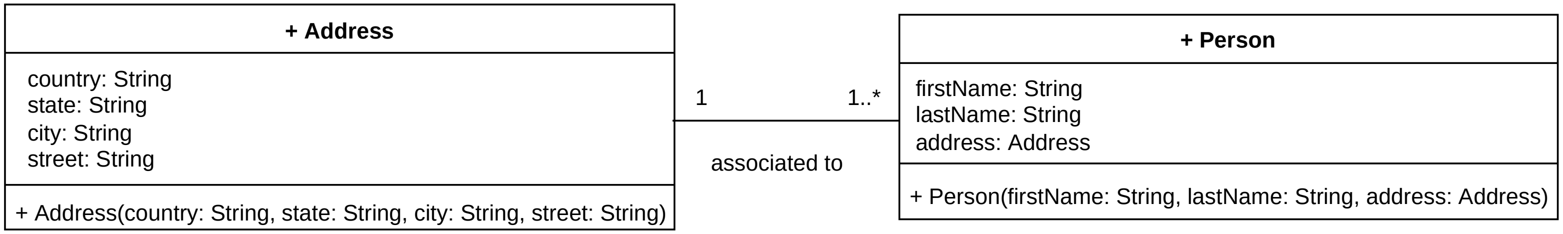


Employee Class

```java
public class Employee {

String firstName;
String lastName;
int ID;
Employee Manager;

public Employee(String firstName , String lastName , int ID , Employee Manager) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.ID = ID;
  this.Manager = Manager;
}

}
```

Uni-direction Association

UML



Java Code

Person Class

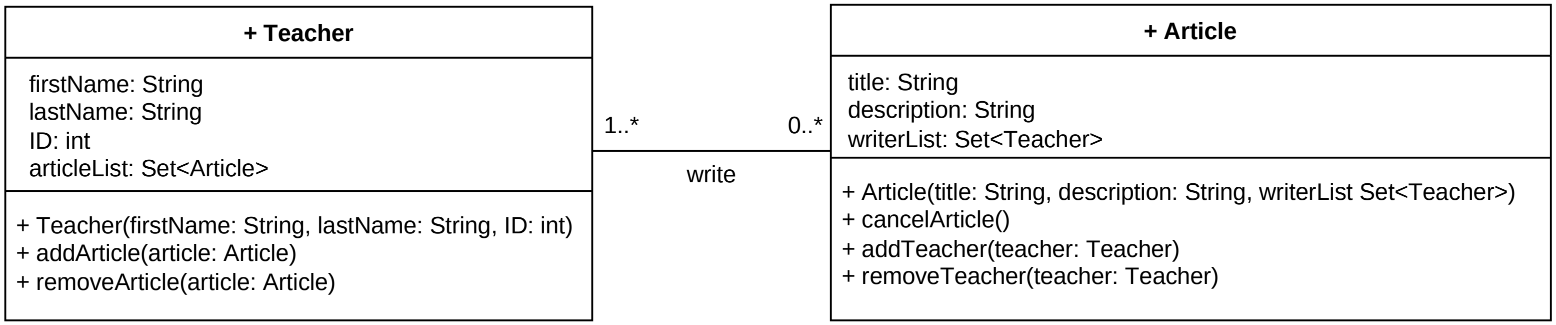
```java
public class Person {

String firstName;
String lastName;
Address address;

public Person (String firstName,String lastName,Address address) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.address = address;
}

}
```

Address Class

```java
public class Address {
String country;
String state;
String city;
String street;

public Address (String country,String state,String city,String street) {
  this.country = country;
  this.state = state;
  this.city = city;
  this.street = street;
}

}
```

Bi-direction Association

UML


+ Teacher
firstName: String
lastName: String
ID: int
articleList: Set<Article>
+ Teacher(firstName: String, lastName: String, ID: int)
+ addArticle(article: Article)
+ removeArticle(article: Article)
1..*
0..*
write
+ Article
title: String
description: String
writerList: Set<Teacher>
+ Article(title: String, description: String, writerList Set<Teacher>)
+ cancelArticle()
+ addTeacher(teacher: Teacher)
+ removeTeacher(teacher: Teacher)

Java Code

Teacher Class

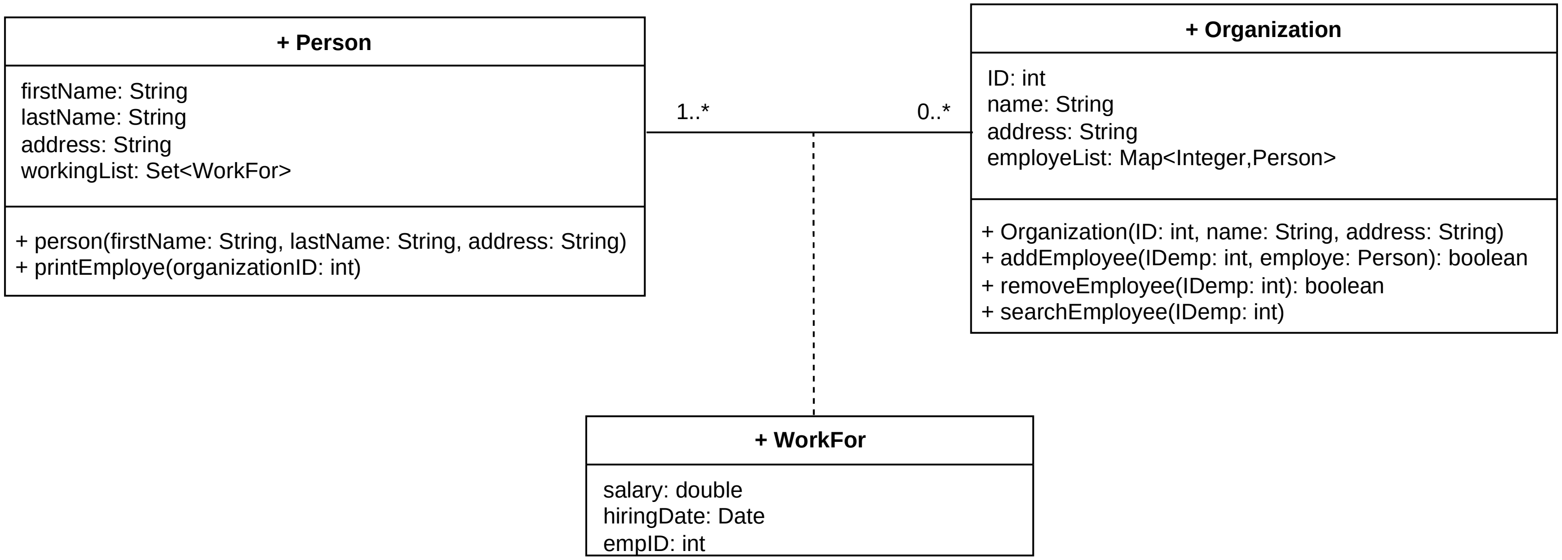
```java
import java.util.LinkedHashSet;
import java.util.Set;

public class Teacher {
String firstName;
String lastName;
int ID;
Set<Article> articleList = new LinkedHashSet<>();

public Teacher(String firstName , String lastName , int ID) {
 this.firstName = firstName;
 this.lastName = lastName;
 this.ID = ID;
}

public void addArticle(Article article) {this.articleList.add(article);}
public void removeArticle(Article article) {this.articleList.remove(article);}
}
```

Article Class

```java
import java.util.LinkedHashSet;
import java.util.Set;

public class Article {
String title;
String description;
Set<Teacher> writerList = new LinkedHashSet<>();

public Article(String title, String description, Set<Teacher>writerList) {
  if (writerList == null || writerList.isEmpty()) {
    throw new IllegalArgumentException("Error Article Must Have At Least One Writer");
  }
  this.title = title;
  this.description = description;

  for(Teacher teacher : writerList) {
    this.addTeacher(teacher);
  }

}

public void cancelArticle() {
  for(Teacher teacher : this.writerList) {
    this.removeTeacher(teacher);
  }
}

public void addTeacher(Teacher teacher) {
  if(this.writerList.add(teacher)) {
  teacher.addArticle(this);
  }
}

public void removeTeacher(Teacher teacher) {
  if(this.writerList.remove(teacher)) {
  teacher.removeArticle(this);
  }
}

}
```

Association Class

UML


+ Person
firstName: String
lastName: String
address: String
workingList: Set<WorkFor>
+ person(firstName: String, lastName: String, address: String)
+ printEmploye(organizationID: int)
1..*
0..*
+ Organization
ID: int
name: String
address: String
employeList: Map<Integer,Person>
+ Organization(ID: int, name: String, address: String)
+ addEmployee(IDemp: int, employe: Person): boolean
+ removeEmployee(IDemp: int): boolean
+ searchEmployee(IDemp: int)
+ WorkFor
salary: double
hiringDate: Date
empID: int

Person Class

```java
import java.util.LinkedHashSet;
import java.util.Set;

public class Person {
String firstName;
String lastName;
String address;
Set<WorkFor> workingList = new LinkedHashSet<>();

public Person (String firstName,String lastName,String address) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.address = address;
}

public void printEmploye(int organizationID) {
  for (WorkFor work : workingList) {
    if(work.organization.ID == organizationID) {
      System.out.println("First Name "+this.firstName);
      System.out.println("Last Name "+this.lastName);
      System.out.println("Address"+this.address);
      System.out.println("Salary : "+work.salary);
      System.out.println("Hiring Date "+work.hiringDate);
      break;
    }
  }}
}
```

Organization Class

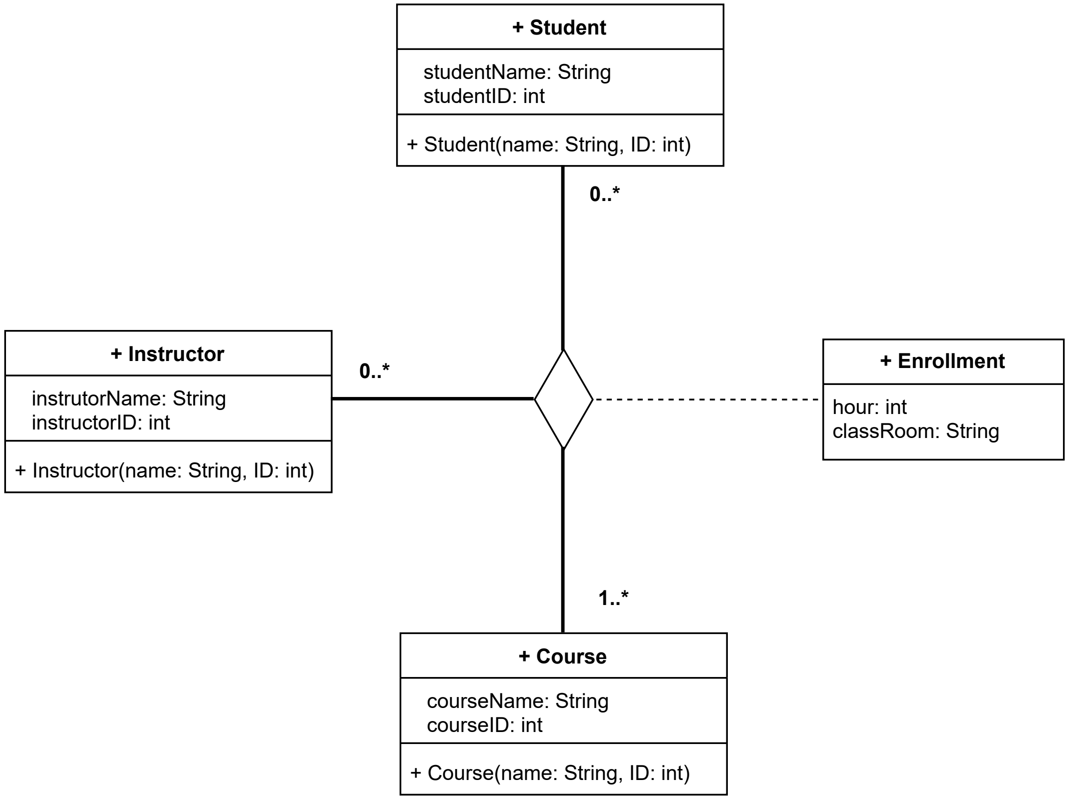
```java
import java.util.HashMap;
import java.util.Map;

public class Organization {
int ID;
String name;
String address;
Map<Integer,Person> employeList = new HashMap<>();

public Organization(int ID,String name , String address) {
  this.ID = ID;
  this.name = name;
  this.address = address;
}

public boolean addEmployee(int empID, Person employe) {
    if (!employeList.containsKey(empID)) {
        employeList.put(empID, employe);
        return true;
    } else {
        System.out.println("Employee with ID " + empID + " already exists.");
        return false;
    }
}

public boolean removeEmployee(int empID) {
    if (!employeList.containsKey(empID)) {
       System.out.println("Employee with ID " + empID + " doesn’t exists.");
         return false;
    } else {
        this.employeList.remove(empID);
        return true;
    }
}


public void searchEmploye(int empID) {
    Person employee = employeList.get(empID);

    if (employee != null) {
        employee.printEmploye(this.ID);
    } else {
        System.out.println("Employee Not Found");
    }
}



}
```

WorkFor Class

```java
import java.util.Date;

public class WorkFor {
Organization organization;
Person employe;
double salary;
Date hiringDate;
int empID;


public WorkFor(int empID,Organization organization,Person employe,float salary,Date hiringDate) {
  this.empID = empID;
  this.organization = organization;
  this.employe = employe;
  this.salary = salary;
  this.hiringDate = hiringDate;
  if(organization.addEmployee(empID, employe)) {
    employe.workingList.add(this);
  }
}

}
```

3-ary Association

UML



Java Code

Course Class

```java
public class Course {
  String courseName;
  int courseID;

  public Course(String name ,int ID ) {
    this.courseName = name;
    this.courseID = ID;
  }
}
```

Instructor Class

```java
public class Instructor {
  String instructorName;
  int instructorID;

  public Instructor(String name ,int ID ) {
    this.instructorName = name;
    this.instructorID = ID;
  }
}
```

Student Class

```java
public class Student {

String studentName;
int studentID;

public Student(String name ,int ID ) {
  this.studentName = name;
  this.studentID = ID;
}

}
```

Enrollment Class

```java
import java.util.LinkedHashSet;
import java.util.Set;

public class Enrollment {
Set<Student> studentList = new LinkedHashSet<>();
Set<Instructor> instructorList = new LinkedHashSet<>();
Course course;
int hour;
String classRoom;

public Enrollment(Set<Student>stdList,Set<Instructor> instList,Course course,int hour,String classRoom) {
  if(stdList == null || stdList.isEmpty()) {
    throw new IllegalArgumentException("Error Can't Course With 0 Students");
  }
  else if(instList == null || instList.isEmpty()) {
    throw new IllegalArgumentException("Error Can't Course With 0 Teachers");
  }
  this.studentList = stdList;
  this.instructorList = instList;
  this.course = course;
  this.hour = hour;
  this.classRoom = classRoom;
}

}
```

Dependency

Dependency

A dependency is represented by a dashed line with an arrow pointing to the **provider class**. It indicates a **temporary relationship** between two classes, where one class relies on the other to perform a specific function or operation. The types of dependency include:

- **Method Call:** One class calls a method of another class without retaining a permanent reference.
- **Object Usage:** One class temporarily interacts with an object of another class, either as:
  - **method parameter**.
  - **return type**.

Difference Between Association & Dependency

A dependency is a **weaker relationship** compared to an association. Unlike an association, which represents a **permanent structural relationship** (e.g., an instance variable), a dependency represents a **temporary interaction**.
In a dependency:

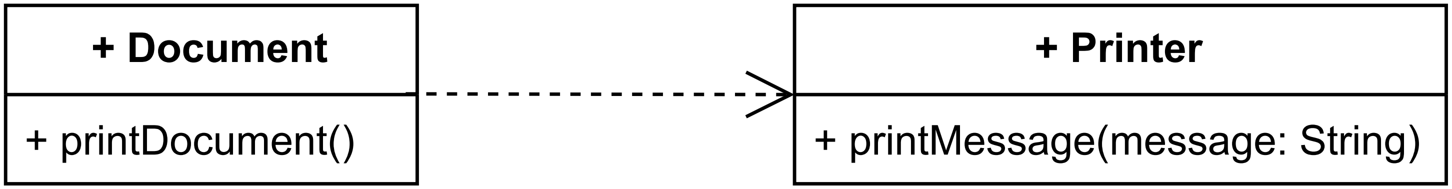
- The referenced object is used only temporarily and is eligible for garbage collection once the interaction ends.
- The classes are loosely coupled, meaning changes to one class have a lower impact on the other.

In contrast, an **association** involves a permanent reference to another class that exists as long as the object itself exists.

Example

Method Call

UML



Printer Class

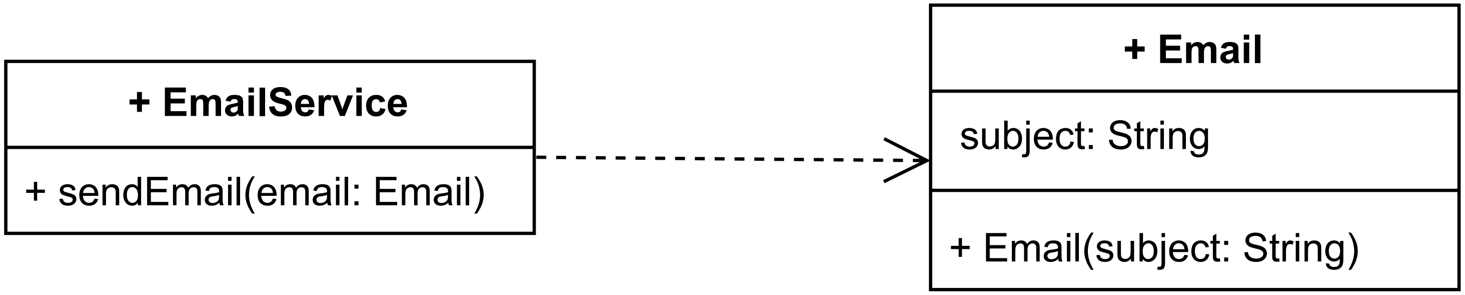
```
public class Printer {
   public void printMessage(String message) {
          System.out.println("Printer: " + message);
      }
}
```

Document Class

```
public class Document {
  public void printDocument() {
      Printer printer = new Printer();
      printer.printMessage("Printing the document...");
   }
}
```

Object Parameter In Method

UML



Email Class

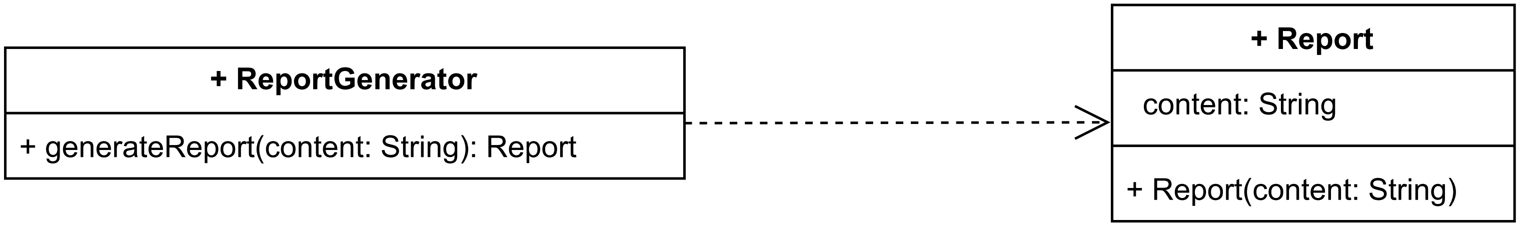
```
public class Email {

String subject;

 public Email(String subject) {
  this.subject = subject;
 }

}
```

EmailService Class

```
public class EmailService {
    public void sendEmail(Email email) {
        System.out.println("Sending email with subject: " + email.subject);
    }
}
```

Return Type Of A Method

UML


+ ReportGenerator
+ generateReport(content: String): Report
+ Report
content: String
+ Report(content: String)

Report Class

```java
public class Report {

    String content;

    public Report(String content) {
        this.content = content;
    }

}
```

ReportGenerator Class

```java
public class ReportGenerator {
    public Report generateReport(String content) {
        return new Report(content);
    }
}
```

Inheritance

Inheritance

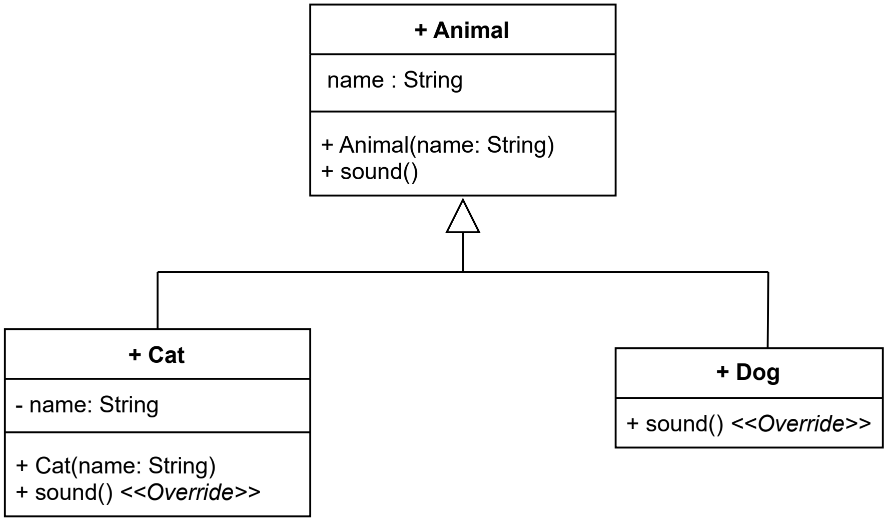
Inheritance occurs when a class **extends** (inherits from) another class. It is represented in UML by a **solid line with a hollow triangle** at the end, pointing to the **parent class**.

Note

Since the child class inherits all attributes and methods from the parent class, **there's no need to explicitly list them unless a method is overridden or the visibility of an attribute is changed**. The constructor is mentioned in the subclass only **if it takes more parameters than the parent class constructor or if its body contains more than just a call to the parent constructor**.

Example

UML



Animal Class

```java
public class Animal {
  String name;

  public Animal(String name){
    this.name = name;
  }

  public void sound() {
    System.out.println(this.name+" making animal sound");
  }
}
```

Cat Class

```java
public class Cat extends Animal {
private String name;

 public Cat(String name) {
    super(name);
    System.out.println("Cat Was Created");
 }

  @Override
 public void sound() {  System.out.println(this.name+"is meowing"); }
}
```

Dog Class

```java
public class Dog extends Animal{

  public Dog(String name) {
    super(name);
  }
  @Override
  public void sound() {
    System.out.println(this.name+"is barking");
  }

}
```

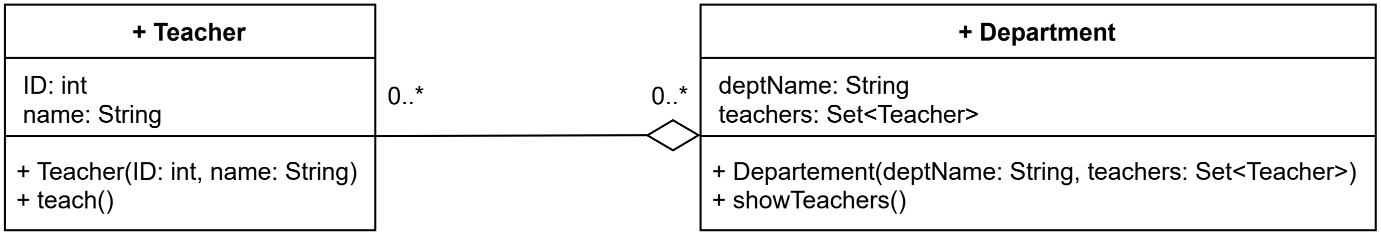
Aggregation

Aggregation

Aggregation is a stronger relationship than association. It involves two classes: a whole class and a part class. The whole class references the part class, but the part class exists independently of the whole class. The part class is unaware of the whole class. Even if the whole class is destroyed by the garbage collector, the part class continues to exist.

Example

UML



Teacher Class

```java
public class Teacher {

  String name;
    int ID;

    public Teacher(String name,int ID) {
        this.name = name;
        this.ID = ID;
    }

    public void teach() {
        System.out.println(name + " is teaching.");
    }
}
```

Department Class

```java
import java.util.LinkedHashSet;
import java.util.Set;

public class Department {
    String deptName;
    Set<Teacher> teachers = new LinkedHashSet<>();

    public Department(String deptName, Set<Teacher> teachers) {
        this.deptName = deptName;
        this.teachers = teachers;
    }

    public void showTeachers() {
        System.out.println("Teachers in " + deptName + " department:");
        for (Teacher teacher : teachers) {
            System.out.println("- "+teacher.ID+ " " + teacher.name);
        }
    }
}
```

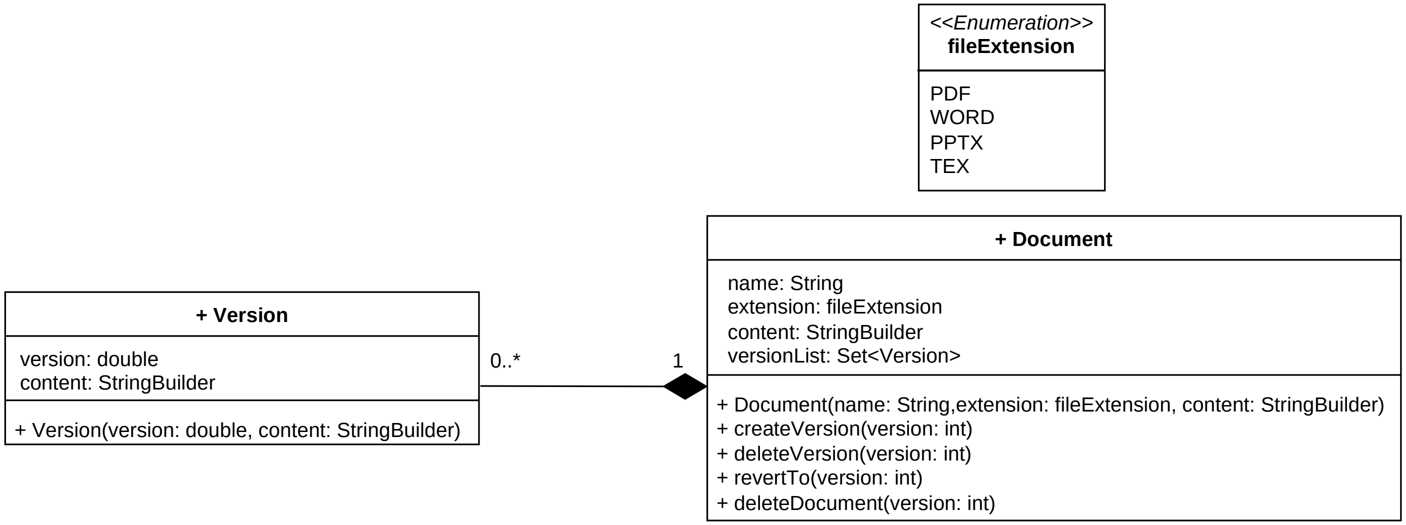
Composition

Composition

Composition is a stronger relationship than aggregation. In composition, when the whole class is deleted, the part class is also deleted, as the part is tightly bound and dependent to the whole.

Example

UML


<<Enumeration>>
fileExtension
PDF
WORD
PPTX
TEX
+ Document
name: String
extension: fileExtension
content: StringBuilder
versionList: Set<Version>
+ Document(name: String,extension: fileExtension, content: StringBuilder)
+ createVersion(version: int)
+ deleteVersion(version: int)
+ revertTo(version: int)
+ deleteDocument(version: int)
+ Version
version: double
content: StringBuilder
+ Version(version: double, content: StringBuilder)
0..*
1

Version Class

```java
public class Version {

double version;
StringBuilder content;

public Version(double version,StringBuilder content) {
  this.version = version;
  this.content = content;
}

}
```

fileExtension Enumeration

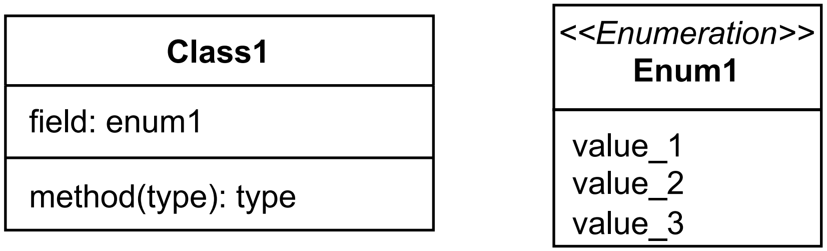
```java
public enum fileExtension {
PDF,WORD,PPTX,TEX;
}
```

Document Class

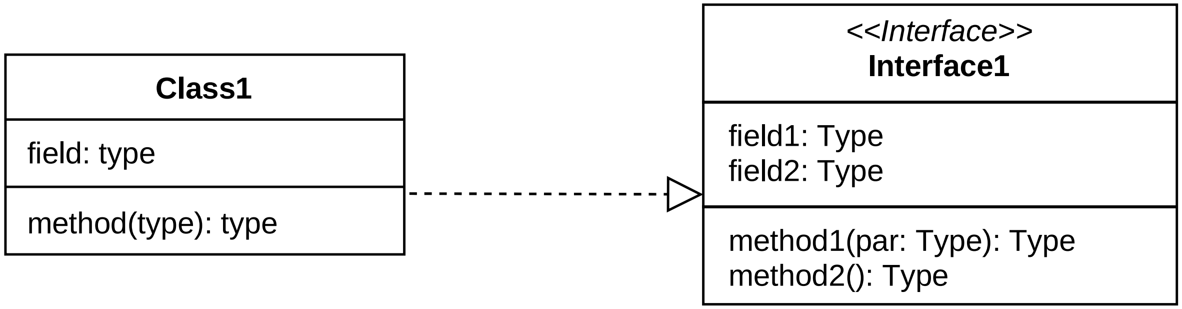
```java
import java.util.LinkedHashSet;
import java.util.Set;

public class Document {

String name;
fileExtension extension;
StringBuilder content;
Set<Version> versionList = new LinkedHashSet<>();

public Document(String name ,FileExtension extension,StringBuilder content) {
  this.name = name;
  this.extension = extension;
  this.content =content;
}

public void createVersion(int version) {
   Version newVersion = new Version(version,this.content);
   this.versionList.add(newVersion);
}

public void deleteVersion(int version) {
   for (Version ver : this.versionList) {
    if(ver.version == version) {
              this.versionList.remove(ver);
        System.out.println("Successfully Deleted Version : "+version);
        return;
      }
       }

   System.out.println("Version "+version+" Not Found");
}

public void revertTo(int version) {

   for (Version ver : this.versionList) {
  if(ver.version == version) {
    this.content = ver.content;
    System.out.println("Successfully Reverted To Version : "+version);
    return;
  }
   }

   System.out.println("Version "+version+" Not Found");
}

public void deleteDocument() {
  versionList.clear();
}

}
```

5.5 Summary

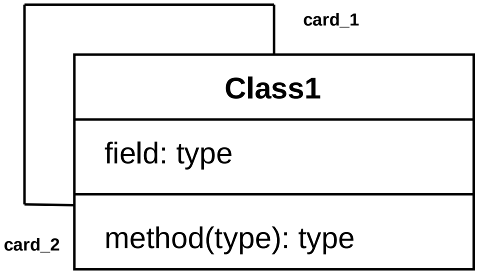
Enumeration



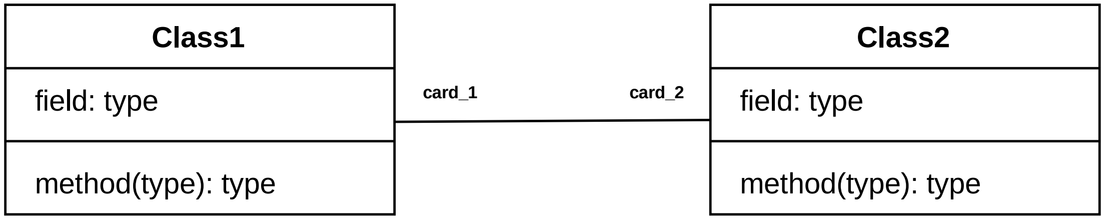
Interface



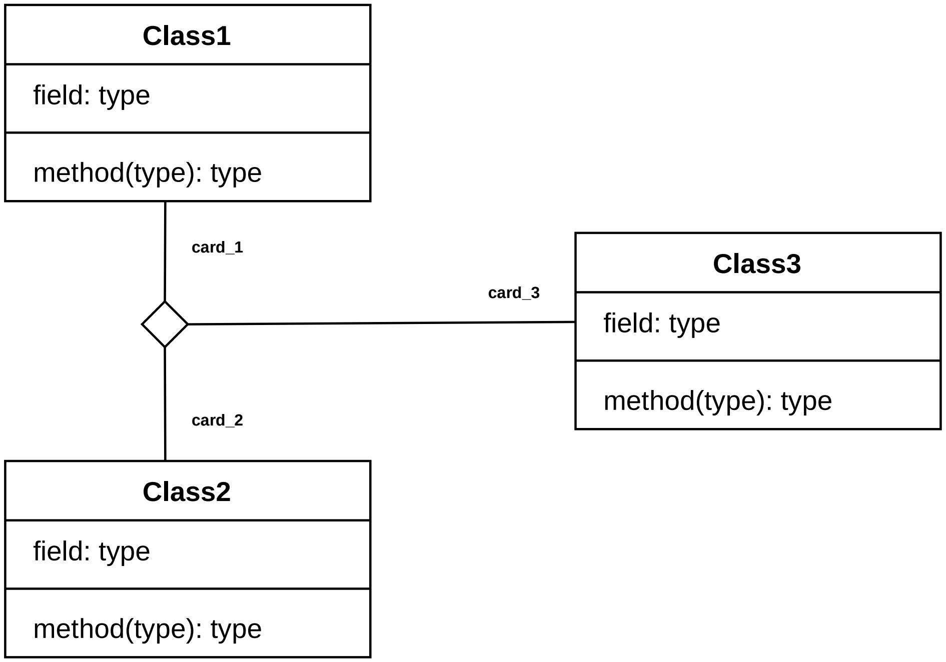
Reflexive Association



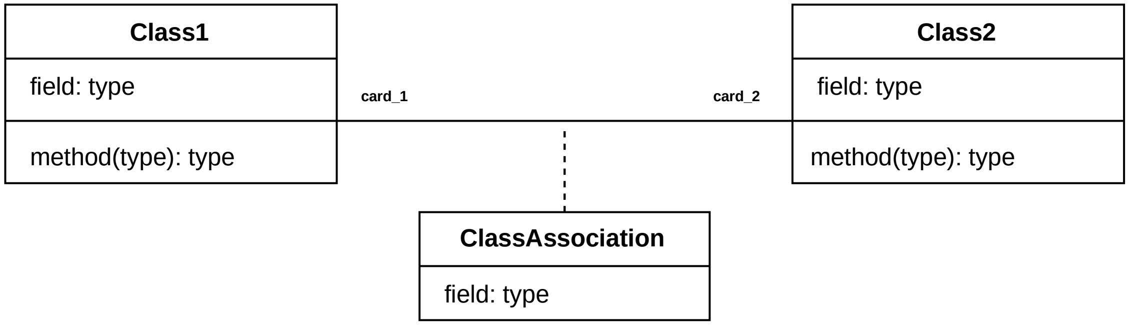
Simple Association



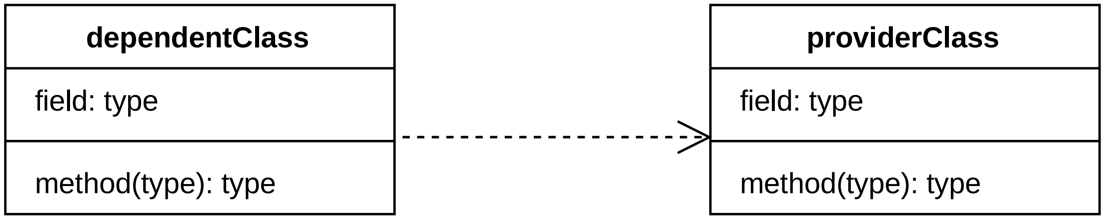
N-ary Association



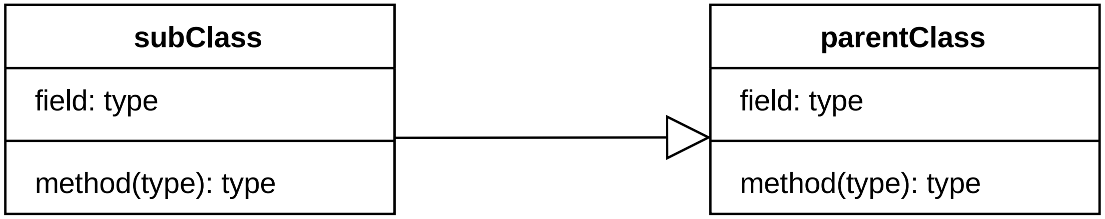
Class Association



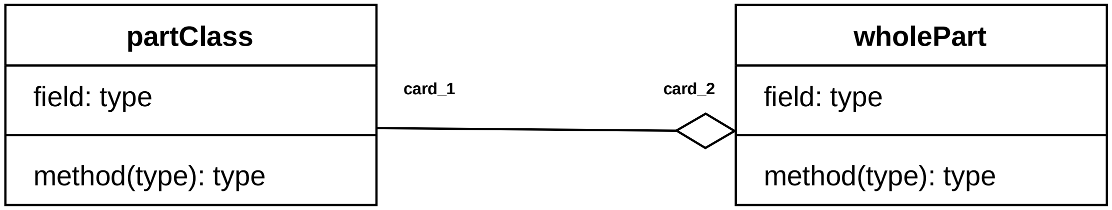
Dependency



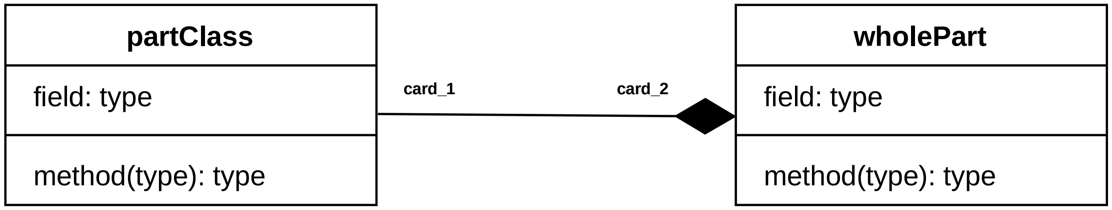
Inheritance



Aggregation



Composition



Note

We can read composition and aggregation as whole class contains part class.

6 Conception

6.1 Introduction

Conception Introduction

Conception represents the solution to a problem, expressed through structured diagrams such as UML. Creating a conception is an iterative process that requires significant time and creativity. Initially, a solution is developed, and subsequent iterations focus on optimizing it.

Each iteration refines and expands the conception until reaching a final result that is easy to maintain. This ensures better implementation, facilitates adding or removing features, and simplifies bug fixes.

Difference Between Conception & UML

Conception $\nLeftrightarrow$ **UML**

UML is merely a tool used to represent the conception, it is not the conception itself. Conception encompasses more than just diagrams—it includes algorithms, explanations of diagrams, and other documents.

Importance of Maintainability

Easy maintainability is one of the key qualities of a good conception and arguably the most important criteria. This is because we want the software to be long-lasting, and effective maintenance is essential to achieving that.

6.2 Itterative Process Of Conception

Global & Detailed Conception

- **Globale Conception :** Modules must be identified (some modules can be divided into sub-modules) , and interaction between modules must be defined
- **Detailed Conception :** Each module must be defined independently in detail.

The conception should have high ratio of cohesion and low ratio of coupling

Why Global Conception Then Detailed Conception

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 High Cohesion & Low Coupling

- **Cohesion**: How related the responsibilities of a module are.
  - **High Cohesion**: The module has a clear, well-defined responsibility, making it easier to understand, maintain, and modify.
  - **Low Cohesion**: The module handles multiple, unrelated responsibilities, making it harder to maintain and understand.
- **Coupling**: How dependent the modules are on each other.
  - **High Coupling**: Modules are highly dependent on each other. A failure in one critical module may cause the entire system to fail.
  - **Low Coupling**: Modules are loosely connected, and changes or failures in one module are less likely to impact the others.

Why We Want High Cohesion and Low Coupling:

- **High Cohesion**: Ensures that each module has a clear, understandable purpose, making the system easier to maintain and extend.
- **Low Coupling**: Reduces dependencies between modules, minimizing the risk of widespread system failure and increasing flexibility.

6.3 Classification Of Conception Method

6.3.1 Function Oriented Conception

Function-Oriented

The software is structured using a functional paradigm. It is divided into a set of functions that interact with each other. The software is viewed as a complex main function that is progressively decomposed into smaller, less complex sub-functions. This process continues until we reach a detailed conception.

Each function has its own local state (local variables), while the software has a global state (global variables) that is shared among all functions.

6.3.2 Object Oriented Conception

Object-Oriented Design

The software is viewed as a collection of encapsulated and independent objects. These objects communicate with each other by sending messages (method calls).

Each object is identified by its name and encapsulated attributes (variables and methods).

6.3.3 Data Oriented Conception

Data-Oriented

The software's structure must reflect the structure of the data it traits . Therefore the conception is influenced by the ouput input data.

6.4 Conception's Principales

To make sure the conception ensures an easily maintainable software we must follow some printcipales :

Principles

- **Abstraction:** Focuses on the essential characteristics while hiding unnecessary details.
- **Modularity:** Divides the system into modules with well-defined interactions, adhering to the principle of high cohesion and low coupling.
- **Encapsulation:** Hides internal details of a module from other modules.
- **Structuring:** Ensures a structured conceptions (levels) , we can at least have general & detailed conception.

6.5 Notation For Fonctional Conception

Notation

- **Data Flow Diagram (DFD):** Shows how data is transformed and passed from one module to another.
- **Structure Diagram (SD):** A hierarchical diagram that illustrates the structured relationships between the components of the software.

Relation Between DFD & SD

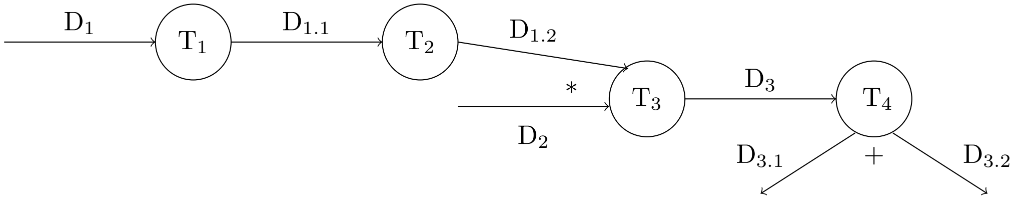
DFD and SD are complementary to each other, working together to clearly describe the functional design of a software system.

6.5.1 DFD

DFD

A DFD diagram consists of four components:

- **Transformations**: Represented as circles.
- **Data**: Represented as axis.
- **Logical AND**: Represented by the symbol *.
- **Logical OR**: Represented by the symbol +.

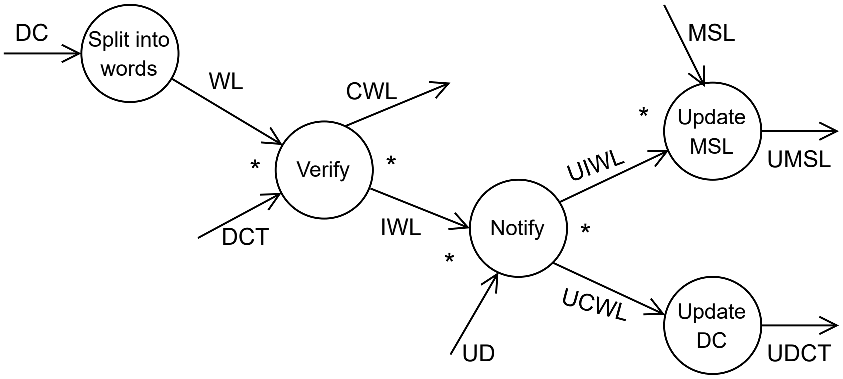


Note

A DFD specifies the operations without detailing how they are performed. Each node in the diagram can be further described with another DFD, allowing for a hierarchical decomposition of processes.

Example :

A Software that processes a text document by splitting it into individual words. It then checks each word against a dictionary. If the word exists in the dictionary, it is marked as correct. Otherwise, the software notifies the user and allows them to decide whether the word is valid. If the user identifies the word as incorrect, it is added to the list of misspelled words. If the user confirms the word is valid, it is added to the dictionary.



Legends

DC : Document
WL: Word List
DCT: Dictionary
CWL: Correct Word List
IWL: Incorrect Word List
UD: User Decision
UIWL: User Incorrect Word List
MSL: Miss Spelled Word List
UMSL: Updated MSL
UCWL: User Correct Word List
UDCT: Updated DCT

6.5.2 SD

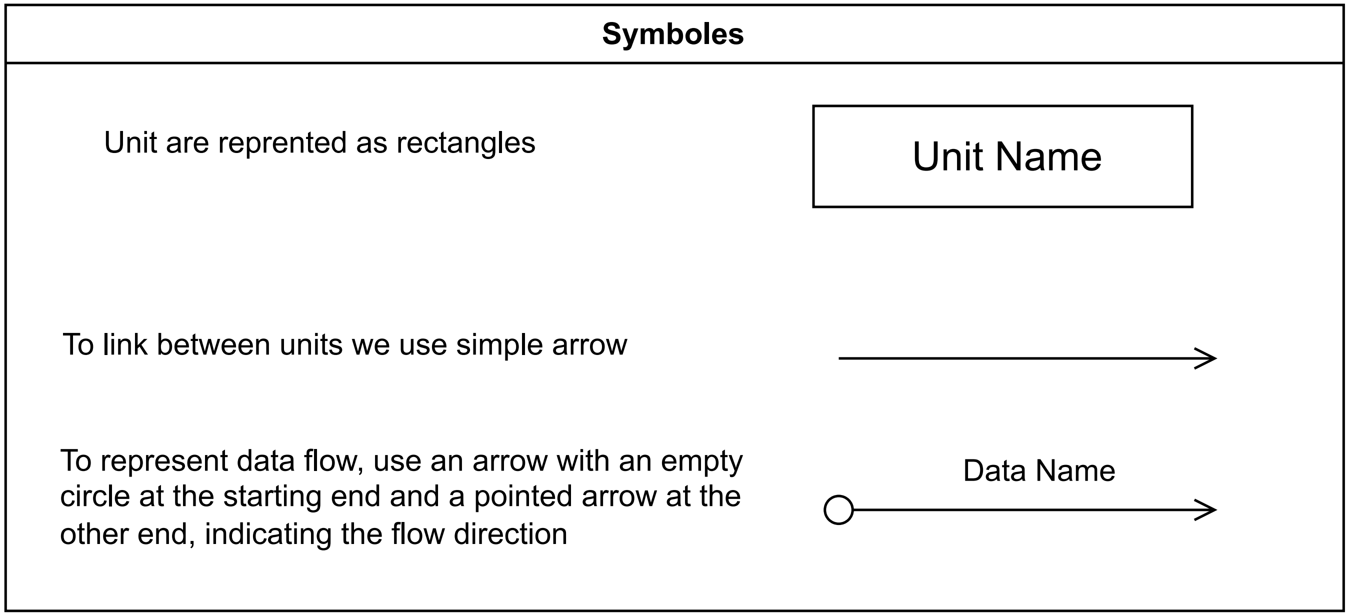
SD

A structure diagram is a hierarchical representation of a system in the form of a tree. It illustrates how the transformation elements of a Data Flow Diagram (DFD) can be implemented within the hierarchical units of software architecture.

Difference Between SD & DFD

They might seem similar but they serve different purposes and compliment each other , SD shows how the data changes and flows with the transformation , and DFD focuses on the hiertachy of the software architecture

Symboles



Unit are reprented as rectangles

To link between units we use simple arrow

To represent data flow, use an arrow with an empty circle at the starting end and a pointed arrow at the other end, indicating the flow direction

Tree Nodes

- **Synchronisation Unit** : The root node at the top represent the the global idea of the software
- **Transformation Unit** : The parent node has to point to at least one other units , represent a function of the software
- **Input/Output Unit** : The leaf node same as parent node just never points to another unit

Note

Synchronisation Unit's Role

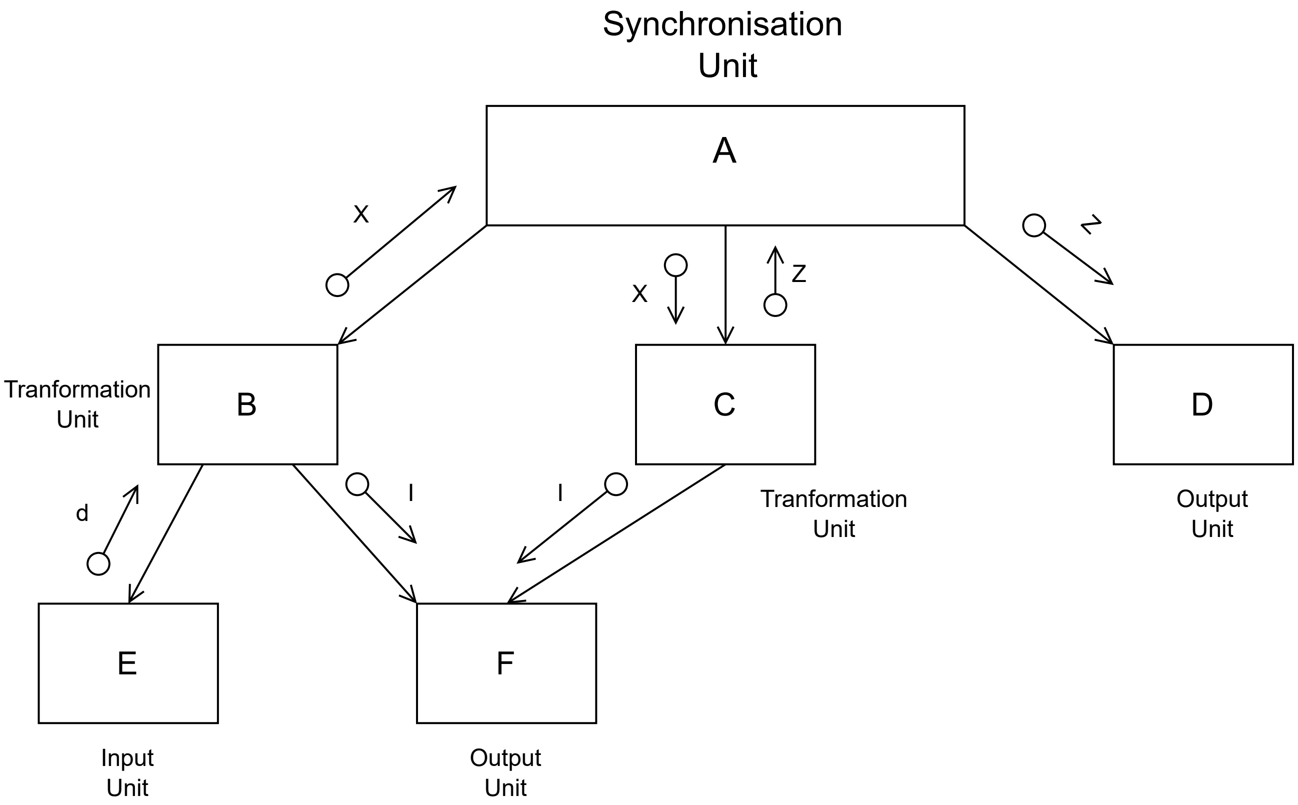
The synchronisation unit (root node) never creates the data it only pass it to other units.

No Link Between Unit Of Same Level

There can be link between units with simple arrow only with units of different level.

SD Level

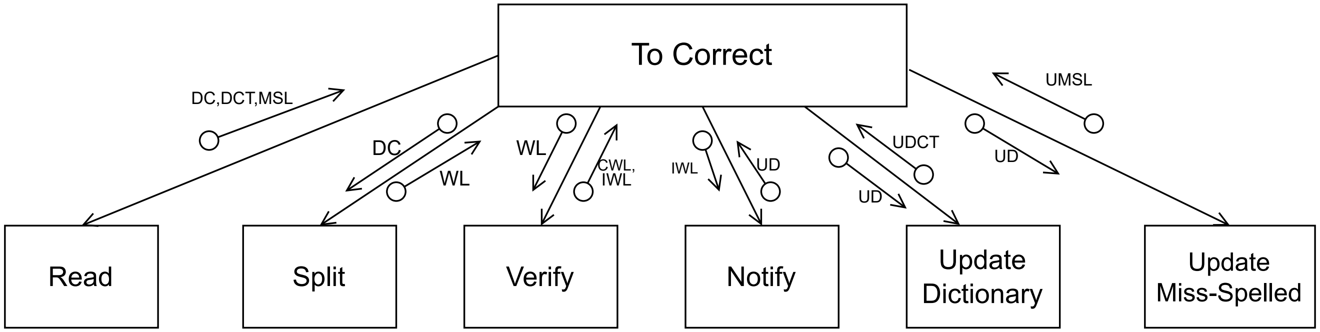
It's the number of levels beside the root node (don't count the synchronisation unit).



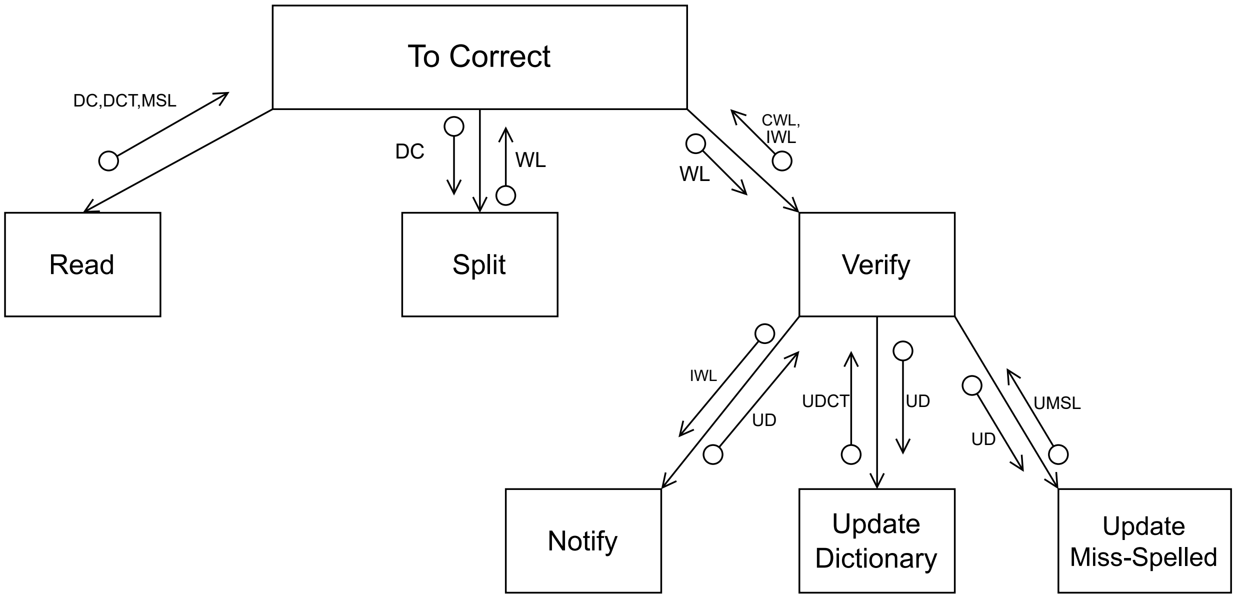
Example :

We will take same example of word checker, we will make level 1 SD then level 2 SD

Level 1 SD :



Level 2 SD :

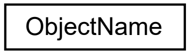
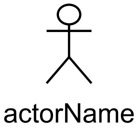
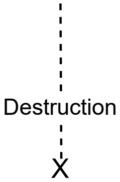
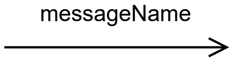
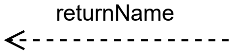
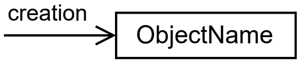
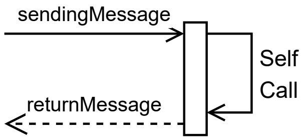


6.5.3 Sequence Diagram

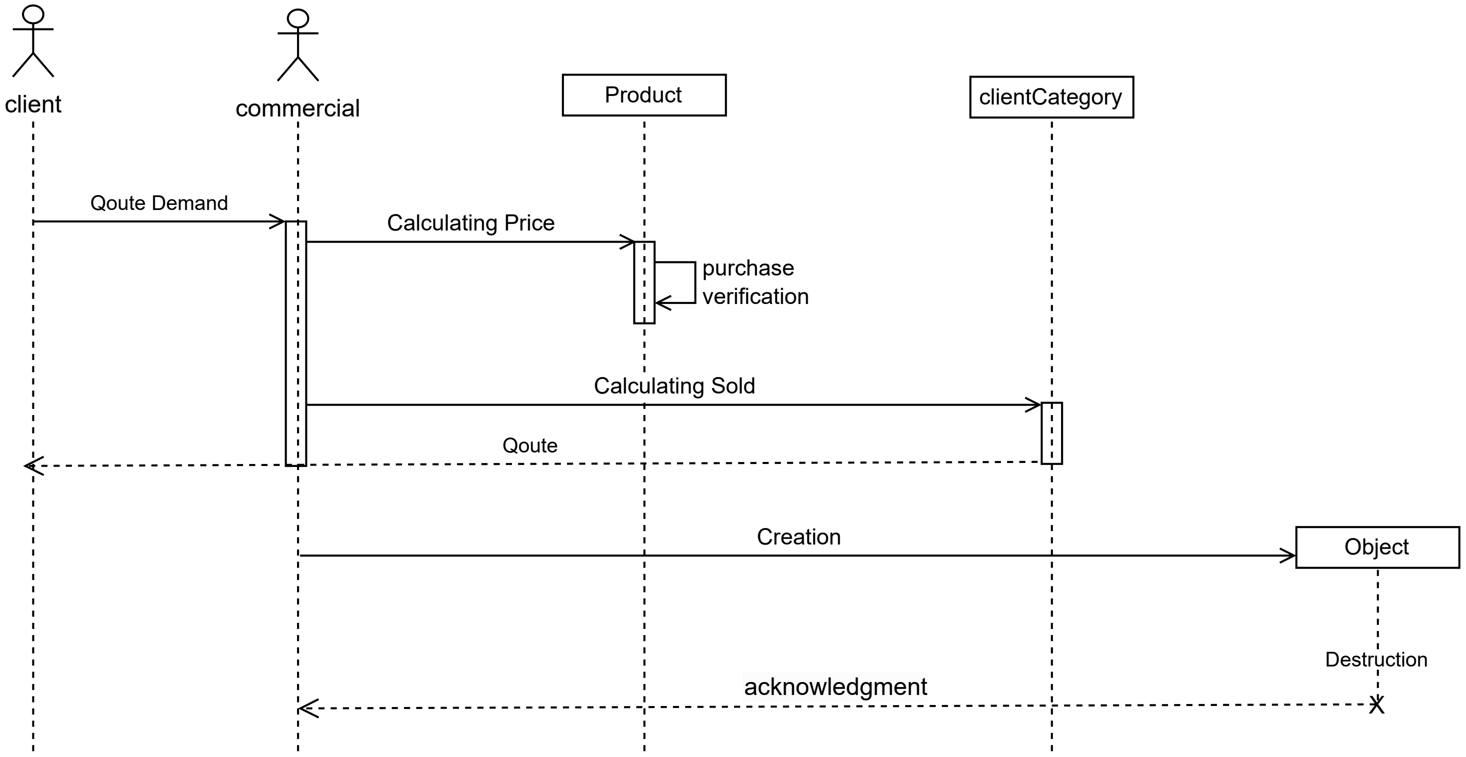
Sequence Diagram

A sequence diagram illustrates the dynamic behavior of a system over time (notion of time). It visualizes the messages exchanged between objects/actors and is read from top to bottom.

Symboles :

Symboles	
Rectangle represent internal object of the software.	
Stick figures represent external actor (people interacting with the software, external software and hardware).	
Vertical dotted line are life line that represents the time span of an object/actor its life ends with an **X.**	
To send a message between an object/actor, we use a **solid line arrow** labeled with the message name.	
To send a return message between an object/actor, we use a **dotted line arrow** labeled with the return value or acknowledgment.	
To create an object , we use a **solid line arrow** labeled with Creation , pointing to the newly created object.	
A **horizontal rectangle** represents the activation of an object or actor, indicating when it is performing a task. During this time, the object can send messages, receive responses, and make self-calls.	

Example :

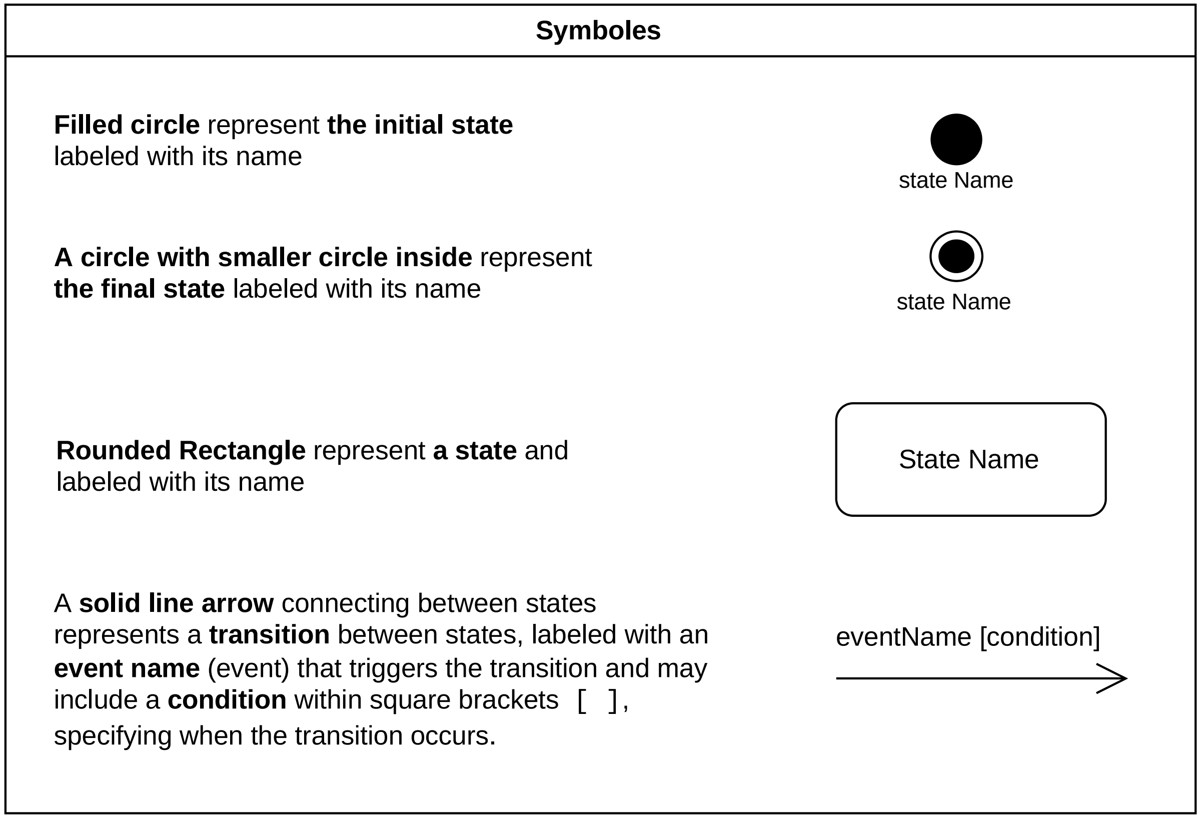


6.5.4 State-Transition Diagram

State Diagram

A state diagram illustrates how the state of an object changes in response to events.

Symboles :



Symboles

Filled circle represent **the initial state** labeled with its name

A circle with smaller circle inside represent **the final state** labeled with its name

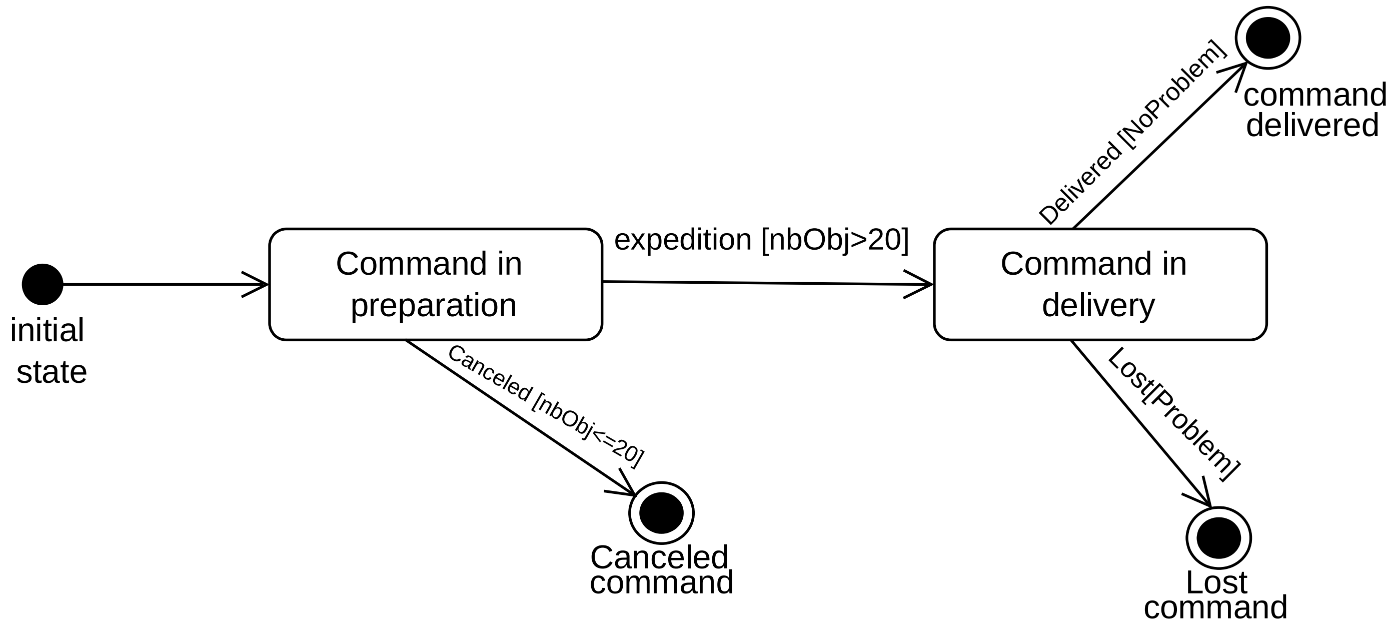
Rounded Rectangle represent **a state** and labeled with its name

A **solid line arrow** connecting between states represents a **transition** between states, labeled with an **event name** (event) that triggers the transition and may include a **condition** within square brackets `[ ]`, specifying when the transition occurs.

Note

We can have only **one initial state** , but we can have **many final state**.

Example :

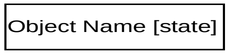
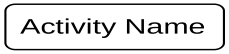
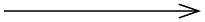
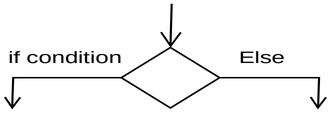
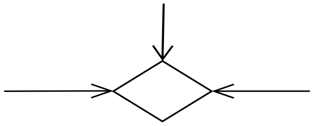
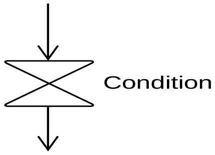
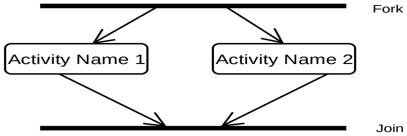
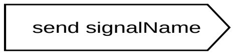
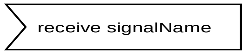
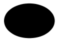
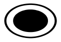


6.5.5 Activity Diagram

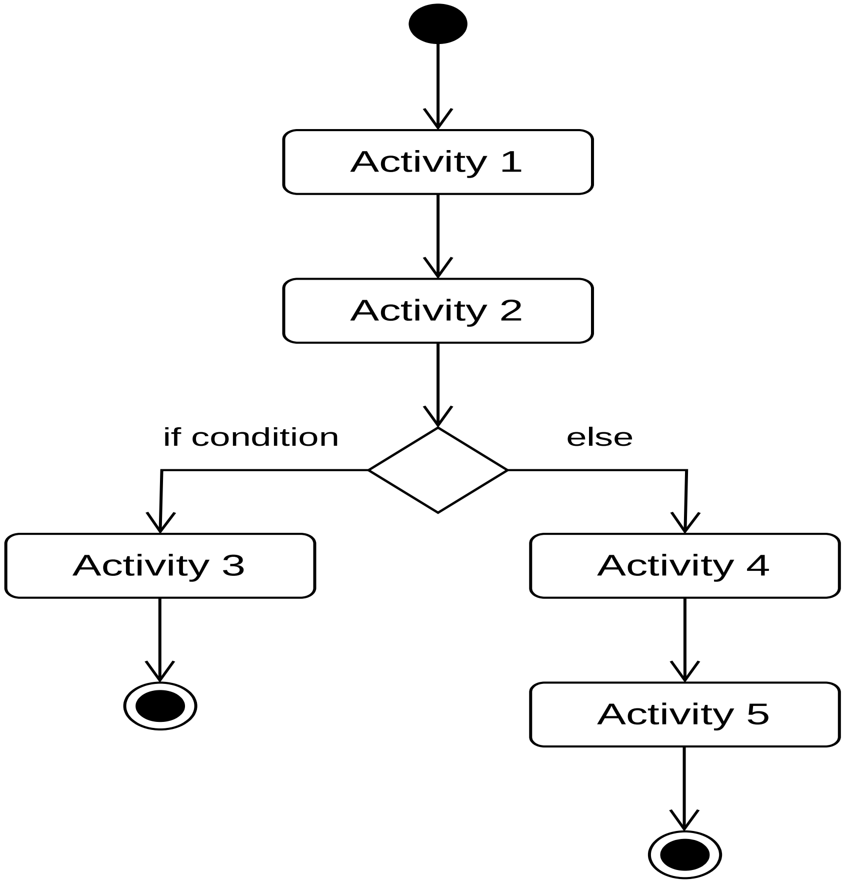
Activity Diagram

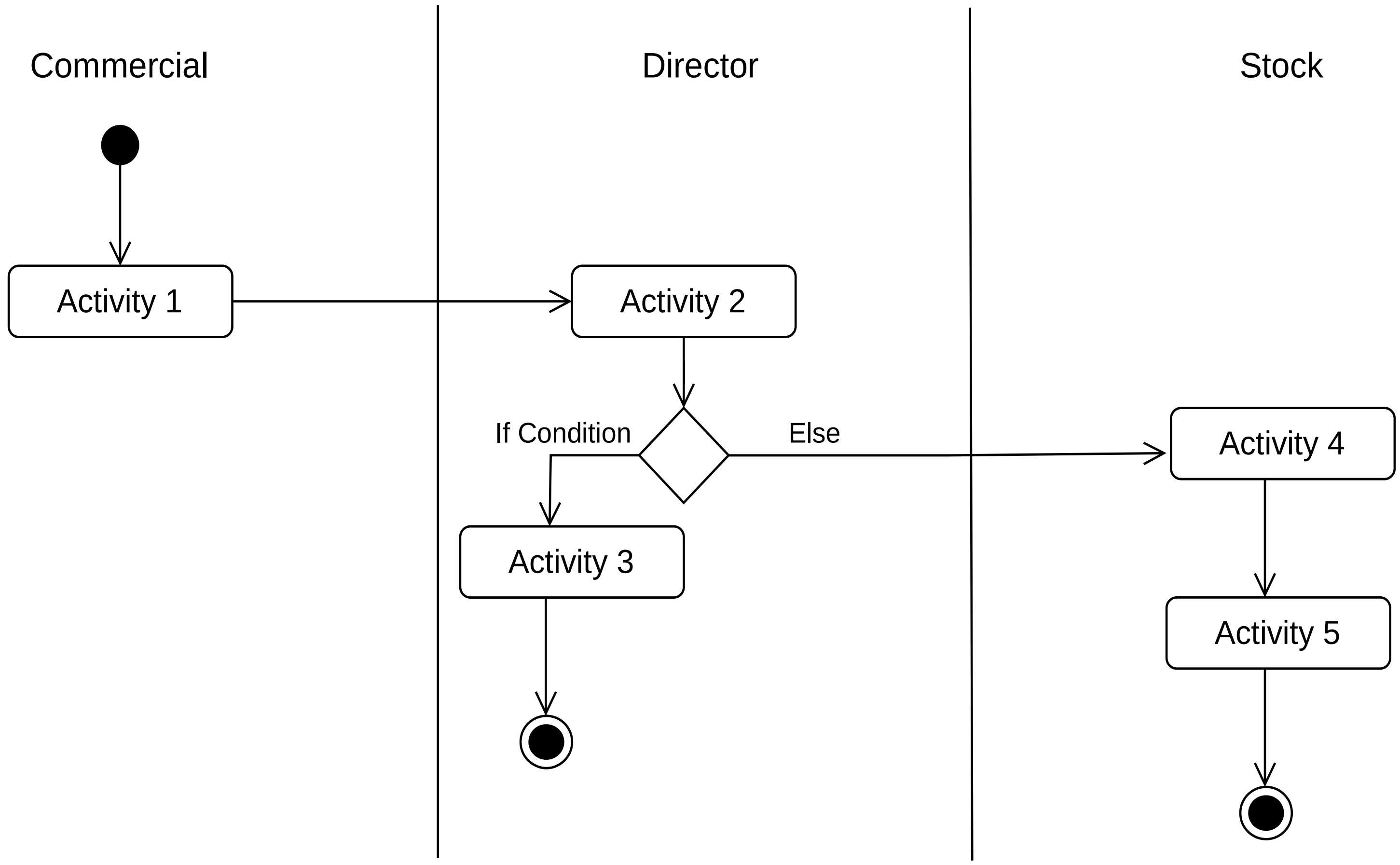
An activity diagram describes the progression of activities within the system and outlines the logical flow of processes.

Symboles :

Symboles	
Rectangle represent an object labeled with its name and state between [ ]	
Rounded Rectangle represent an activity	
To link between activities or any entity we use **solid arrow line**	
Diamond can represent a conditional statement , the if on the left side and else on right side	
Diamond can be a point join between the entities	
Hourglass shape wait until a certain condition is met to proceed to next activity	
To represent **parallelism** in an activity diagram, a **fork** (thick line) splits the flow into parallel activities, and a **join** (another thick line) merges the parallel activities back into a single flow.	
To send signal	
To receive signal	
Filled circle represent the start of the activity diagram or a macro	
Circle with inner filled circle represent the end of the activity diagram or a macro	

Example :


Activity 1
Activity 2
if condition
else
Activity 3
Activity 4
Activity 5



Note

- We can have only **one begin** , but we can have **end**.
- Even though activity diagram has the notion of state it focuses more on the logical flow of activities of the system.

6.5.6 Design Pattern

Definition

A design pattern describes **proven** and **abstract** solutions to **recurring** problems in software design.

Why Proven

Proven solutions have been tested on real projects, ensuring reliability and effectiveness.

Why Abstract

Abstract solutions can be adapted and customized to meet specific needs.

Why Recurring

We want model for recurring problems to address patterns in design problems that repeat across different contexts.

History

The concept of design patterns was first introduced by Christopher Alexander in the field of architecture. It was later adapted to the field of computer science in 1977.

Categories :

The first influential book on design patterns, **Design Patterns: Elements of Reusable Object-Oriented Software** by the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides), was published in its second edition in 1995. It presents a collection of 23 design patterns, classified into three categories.

Creational Design Pattern

Addresses issues related to object creation and configuration, such as the Singleton pattern.

Structural Design Pattern

Describes how to structure classes like: The Composite Pattern

Behavioral Design Pattern

Focuses on the interaction and communication between objects to define the behavior of an application, such as the Observer pattern.

Note

The design patterns from the "Gang of Four" are intended for object-oriented design.

Pros:

- Solves recurring problems with proven and reliable solutions.
- Improves quality and speed.
- Design patterns provide a common language for communication between designers.
- Design patterns are independent of implementation languages and sufficiently generic (abstract) to be applied in various situations.

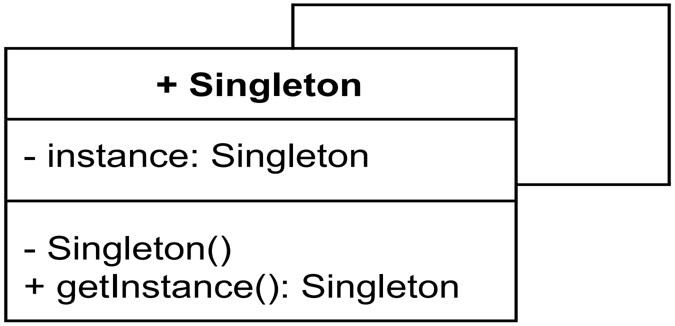
Cons:

- Design patterns need to be mastered and thoroughly studied.
- Design patterns always require adaptation when applied.
- Design patterns can increase the complexity of simple solutions.

Some Design Pattern:

Singleton:

Singleton is a creational design pattern that aims to limit the instanciation of the a class to only one object and give global acess to it.



Explication

We first initialize a static private attribut called instance to null , when ever we want to get instance of singleton we call the public static method getInstance that checks if instance is null if yes it will call the private constructor and in either case it returns the instance

Java Code:

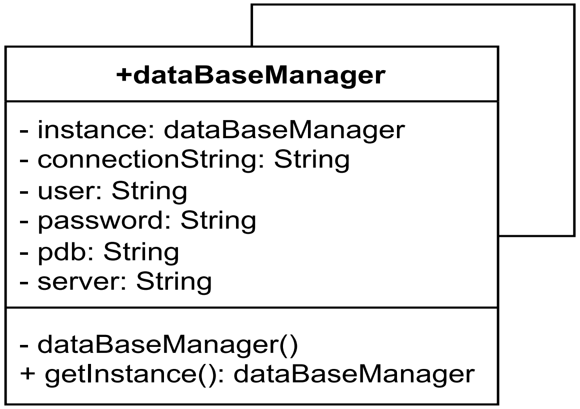
```java
public class Singleton {
  private static Singleton instance = null;

  private Singleton() {
    System.out.println("Instance Of Singleton");
  }

  public static Singleton getInstance() {

    if(instance == null) {
      instance = new Singleton();
    }

    return instance;
  }

}
```

```java
public class Main {

  public static void main(String[] args) {

    Singleton.getInstance();
    Singleton.getInstance();
  }

}
```

Output:

```
Instance Of Singleton
```

Example:



Java Code:

```java
public class Main {

  public static void main(String[] args) {

    dataBaseManager dbManager = dataBaseManager.getInstance();
    System.out.println(dbManager.getConnectionString());
    dbManager.setUser("Dbaiot");
    dbManager.setPassword("psw");
    System.out.println(dbManager.getConnectionString());
    dbManager = dataBaseManager.getInstance();

  }

}
```

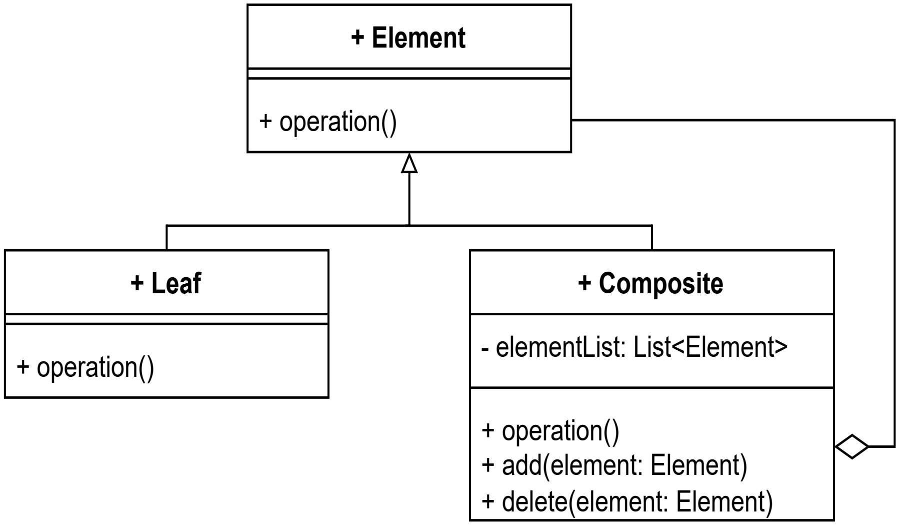
```java
public class dataBaseManager {
    private static dataBaseManager instance = null;
    private String connectionString;
    private String user="system";
    private String password="1234";
    private String pdb="XE";
    private String server = "localhost:1521";

    private dataBaseManager() {
        connectionString = "system/1234//localhost:1521/XE";
        System.out.println("dataBaseManager Initialized");
    }



    public static dataBaseManager getInstance() {
        if (instance == null) {
            instance = new dataBaseManager();
        }
        else {
          System.out.println("Already Initialized");
        }
        return instance;
    }


    public String getConnectionString() {
        return this.connectionString;
    }

    public String getUser() {
        return this.user;
    }

    public String getPDB() {
        return this.pdb;
    }

    public String getPassword() {
      return this.password;
    }

    public String getServer() {
        return this.server;
    }

    public void setUser(String user) {
      this.connectionString = this.connectionString.replace(this.user,user);
    }

    public void setPassword(String password) {
      this.connectionString = this.connectionString.replace(this.password,password);
    }

    public void setServer(String server) {
      this.connectionString = this.connectionString.replace(this.server,server);
    }

    public void setPDB(String pdb) {
      this.connectionString = this.connectionString.replace(this.pdb,pdb);
    }

}
```

Output:

```
dataBaseManager Initialized
system/1234//localhost:1521/XE
Dbaiot/psw//localhost:1521/XE
Already Initialized
```

Composite:

The Composite pattern is a structural design pattern that imposes a hierarchical tree structure, consisting of simple elements and composite elements.



Explication

The Element class serves as an abstract parent class with an unimplemented method, operation(). The Leaf class inherits from Element and overrides the operation() method to provide its specific implementation. Similarly, the Composite class also inherits from Element and implements operation(). However, it has an attribute that is a collection of Element objects. This design allows the collection to hold both simple elements (Leaf) and complex elements (Composite), organizing them into a tree structure.

Note

Element can be an abstract class or an interface both implementations are valid.

Java Code:

```java
public abstract class Element {

  public Element(){

  }

  public abstract void operation();
}
```

```java
public class Leaf extends Element {

  @Override
  public void operation() {
    System.out.println("Leaf Operation");
  }

}
```

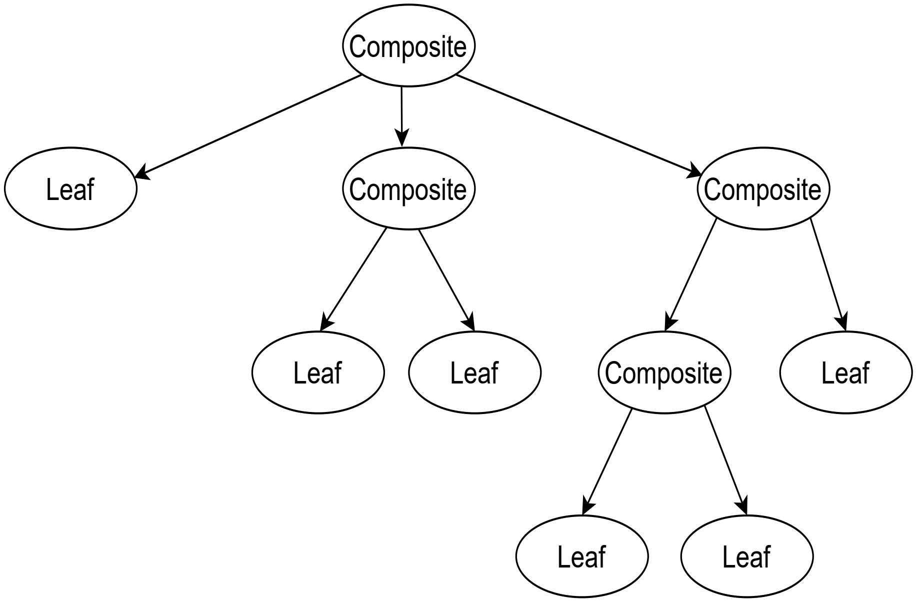
```java
import java.util.LinkedList;
import java.util.List;

public class Composite extends Element {
private List<Element> elementList = new LinkedList<>();

 @Override
 public void operation() {
  System.out.println("Composite Operation");
 }

 public void add(Element element) {
   this.elementList.add(element);
 }

 public void delete(Element element) {
   this.elementList.remove(element);
 }

 public List<Element> getElementList(){
   return this.elementList;
 }

 public void printTree(int deepness,List<Element> elementList) {
   for(Element element : elementList) {
     if(element instanceof Leaf) {
      System.out.println(" ".repeat(deepness)+"Leaf");
     }

     else if(element instanceof Composite) {
        ++deepness;
        System.out.println(" ".repeat(deepness)+"Composite");
        Composite com = (Composite)element;
        printTree(deepness+1,com.elementList);
     }
   }
 }
}
```

```java
public class Main {

  public static void main(String[] args) {
    Composite com = new Composite();

    Leaf l1 = new Leaf();
    Leaf l2 = new Leaf();
    Leaf l3 = new Leaf();
    Leaf l4 = new Leaf();
    Leaf l5 = new Leaf();

    com.add(l1);
    com.add(l2);

    Composite comInner1 = new Composite();
    comInner1.add(l3);

    Composite comInner2 = new Composite();
    comInner2.add(l4);
    comInner2.add(l5);

    comInner1.add(comInner2);

    com.add(comInner1);

    for (Element element : com.getElementList()) {
      element.operation();
    }

    System.out.println("\n");

    com.printTree(0,com.getElementList());

  }

}
```

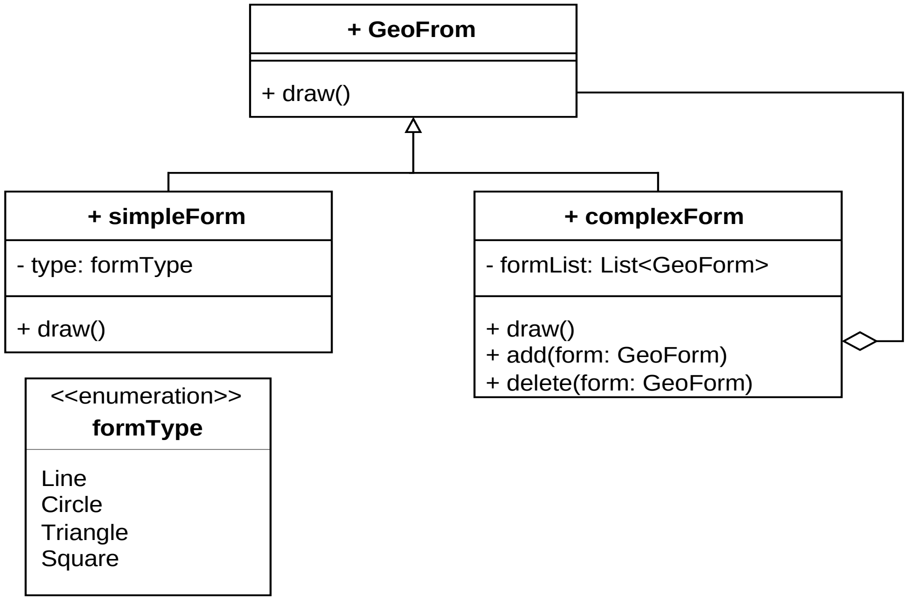
Output:

```
Leaf Operation
Leaf Operation
Composite Operation


Leaf
Leaf

 Composite
  Leaf
   Composite
    Leaf
    Leaf
```

Graphical Representation Of The Composite



Example:



Java Code:

```java
public abstract class GeoForm {

  public GeoForm(){

  }

  public abstract void draw();
}
```

```java
public class simpleForm extends GeoForm {

  private formType type;

  public simpleForm(String type){
    type = type.toUpperCase();
    try {
       formType.valueOf(type);
       this.type = formType.valueOf(type);
     }
    catch(IllegalArgumentException e) {
      System.out.println(e.getMessage());
    }
  }



  @Override
  public void draw() {
    System.out.println("Draw "+this.type);
  }

  public void setType(String type) {
    type = type.toUpperCase();
    try {
       formType.valueOf(type);
       this.type = formType.valueOf(type);
     }
    catch(IllegalArgumentException e) {
      System.out.println(e.getMessage());
    }
  }

  public formType getType() {
    return this.type;
  }

}
```

```java
public enum formType {
LINE,CIRCLE,SQUARE,TRIANGLE;
}
```

```
import java.util.LinkedList;
import java.util.List;

public class complexForm extends GeoForm {
  private List<GeoForm> formList = new LinkedList<>();

   @Override
   public void draw() {
    System.out.println("Draw Complexe Form");
   }

   public void add(GeoForm form) {
     this.formList.add(form);
   }

   public void delete(GeoForm form) {
     this.formList.remove(form);
   }

   public List<GeoForm> getFormList(){
     return this.formList;
   }

   public void printTree(int deepness,List<GeoForm> formList) {
     for(GeoForm form : formList) {
       if(form instanceof simpleForm) {
        simpleForm sim = (simpleForm)form;
        System.out.println(" ".repeat(deepness)+sim.getType());
       }

       else if(form instanceof complexForm) {
          ++deepness;
          System.out.println(" ".repeat(deepness)+"Complex");
          complexForm com = (complexForm)form;
          printTree(deepness+1,com.formList);
       }
     }
   }

  }
```

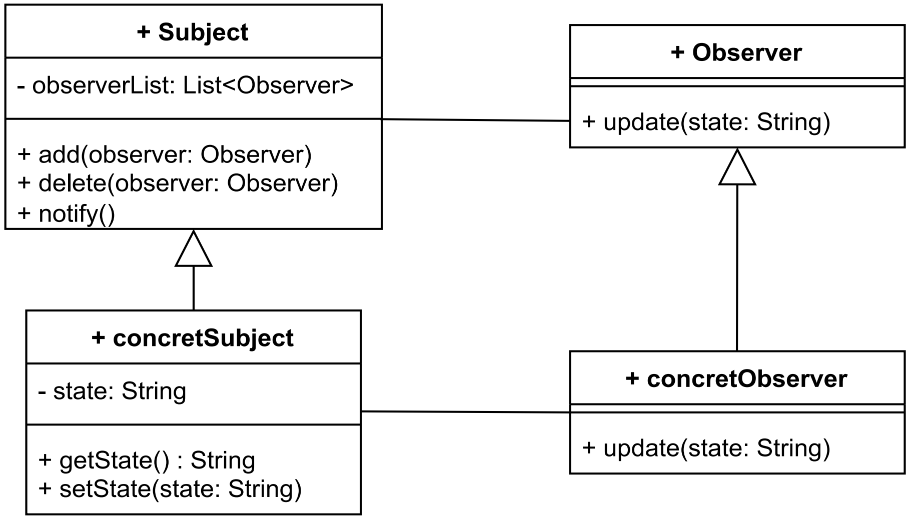
```java
public class Main {

  public static void main(String[] args) {
    simpleForm sim1 = new simpleForm("line");
    simpleForm sim2 = new simpleForm("circle");
    simpleForm sim3 = new simpleForm("square");
    simpleForm sim4 = new simpleForm("triangle");

    complexForm com = new complexForm();
    com.add(sim1);
    com.add(sim4);

    complexForm comInner2 = new complexForm();
    comInner2.add(sim1);
    comInner2.add(sim1);

    complexForm comInner1 = new complexForm();
    comInner1.add(sim3);
    comInner1.add(sim2);

    comInner1.add(comInner2);

    com.add(comInner1);

    for (GeoForm form : com.getFormList()) {
      form.draw();
    }

    System.out.println("\n");

    com.printTree(0,com.getFormList());



  }

}
```

Output:

```
Draw LINE
Draw TRIANGLE
Draw Complexe Form


LINE
TRIANGLE
 Complex
  SQUARE
  CIRCLE
   Complex
    LINE
    LINE
```

Observer:

The Observer pattern is a behavioral design pattern.It monitors the state of a subject object, and when the state changes, the subject notifies the observers, which then update accordingly.



Explication

The Observer is an abstract class that defines an unimplemented method, update(state), which is executed each time the state changes. The Subject is also an abstract class. It holds a collection of observers and defines unimplemented methods: add(observer), delete(observer), and notify(). The notify() method calls the update method of each observer when the state changes. A ConcreteObserver inherits from Observer and implements the update method. A ConcreteSubject inherits from Subject, overrides all unimplemented methods, and includes a state attribute with its getter and setter. Whenever the setter is called, the notify() method is invoked within its body to ensure all observers are updated.

Note

Subject and Observer can be an abstract class or an interface both implementations are valid.

Java Code:

```java
public abstract class Observer {

   private String name;

   public Observer(String name) {
     this.name = name;
   }

   public abstract void update(String state);

   public void setName(String name) {
     this.name = name;
   }

   public String getName() {
     return this.name;
   }

}
```

```java
import java.util.LinkedList;
import java.util.List;

public abstract class Subject {

  private List<Observer> observerList = new LinkedList<>();

  public abstract void add(Observer observer);
  public abstract void delete(Observer observer);
  public abstract void notifyObservers();

  public List<Observer> getObserverList() {
    return this.observerList;
  }

}
```

```java
public class concretObserver extends Observer{

  public concretObserver(String name) {
    super(name);
  }

  @Override
  public void update(String state) {
    System.out.println("Observer "+this.getName()+" New State : "+state);
  }

}
```

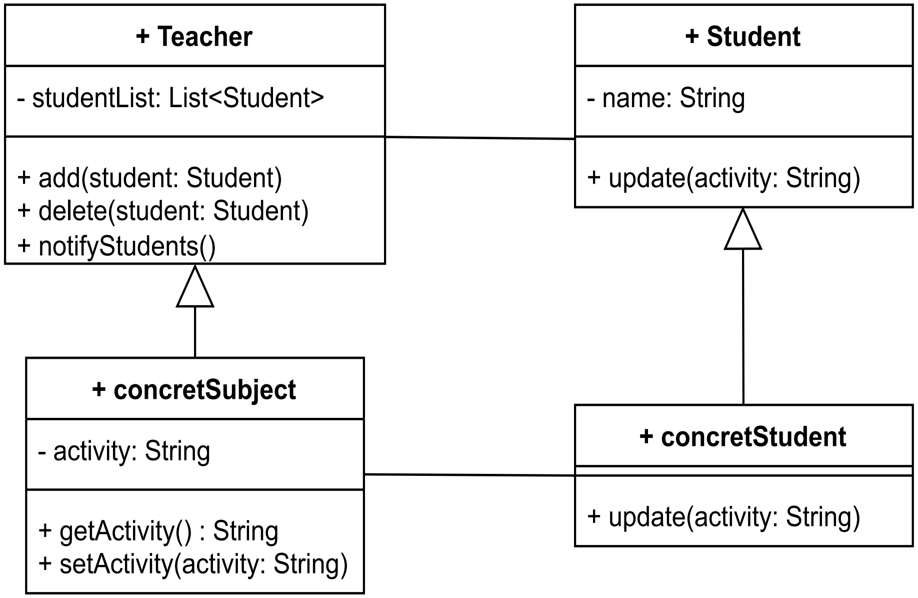
```java
public class concretSubject extends Subject {

  private String state;

  @Override
  public void add(Observer observer) {
    this.getObserverList().add(observer);

  }

  @Override
  public void delete(Observer observer) {
    this.getObserverList().remove(observer);
  }

  @Override
  public void notifyObservers() {
    for(Observer obs : getObserverList()) {
      obs.update(this.state);
    }
  }

  public String getState() {
    return this.state;
  }

  public void setState(String state) {
        this.state = state;
        System.out.println("Setting state to "+state);
        notifyObservers();
    }

}
```

```java
public class Main {
    public static void main(String[] args) {

        concretSubject subject = new concretSubject();

        Observer observer1 = new concretObserver("Alice");
        Observer observer2 = new concretObserver("Bob");

        subject.add(observer1);
        subject.add(observer2);

        subject.setState("State 1");

        subject.delete(observer1);

        subject.setState("State 2");
    }
}
```

Output:

```
Setting state to State 1
Observer Alice New State : State 1
Observer Bob New State : State 1
Setting state to State 2
Observer Bob New State : State 2
```

Example:



Java Code:

```java
public abstract class Student {
  private String name;

  public Student(String name) {
    this.name = name;
  }

  public abstract void update(String activity);

  public void setName(String name) {
     this.name = name;
  }

  public String getName() {
     return this.name;
  }
}
```

```java
import java.util.LinkedList;
import java.util.List;

public abstract class Teacher {
private List<Student> studentList = new LinkedList<>();

  public abstract void add(Student student);
  public abstract void delete(Student student);
  public abstract void notifyStudents();

  public List<Student> getStudentList() {
    return this.studentList;
  }
}
```

```java
public class concretStudent extends Student {

  public concretStudent(String name) {
    super(name);

  }

  @Override
  public void update(String activity) {
    System.out.println("Student "+getName()+" changed activity to "+activity);
  }

}
```

```java
public class concretTeacher extends Teacher{

  private String activity;

  @Override
  public void add(Student student) {
    getStudentList().add(student);
  }

  @Override
  public void delete(Student student) {
    getStudentList().remove(student);
  }

  @Override
  public void notifyStudents() {
    for(Student student : getStudentList()) {
      student.update(this.activity);
    }
  }

  public String getActivity() {
    return this.activity;
  }

  public void setActivity(String activity) {
        this.activity = activity;
        System.out.println("Teacher Setting Activity To "+activity);
        notifyStudents();
    }


}
```

```java
public class Main {

  public static void main(String[] args) {
        concretTeacher teacher = new concretTeacher();

          Student st1 = new concretStudent("Zaki");
          Student st2 = new concretStudent("Rabah");

          teacher.add(st1);
          teacher.add(st2);


          teacher.setActivity("DW 2 EX1");

          teacher.delete(st2);

          teacher.setActivity("LAB2");
  }
}
```

Output:

```
Setting Activity To DW 2 EX1
Student Zaki changed activity to DW 2 EX1
Student Rabah changed activity to DW 2 EX1
Setting Activity To LAB2
Student Zaki changed activity to LAB2
```