Design and Architectural Engineering

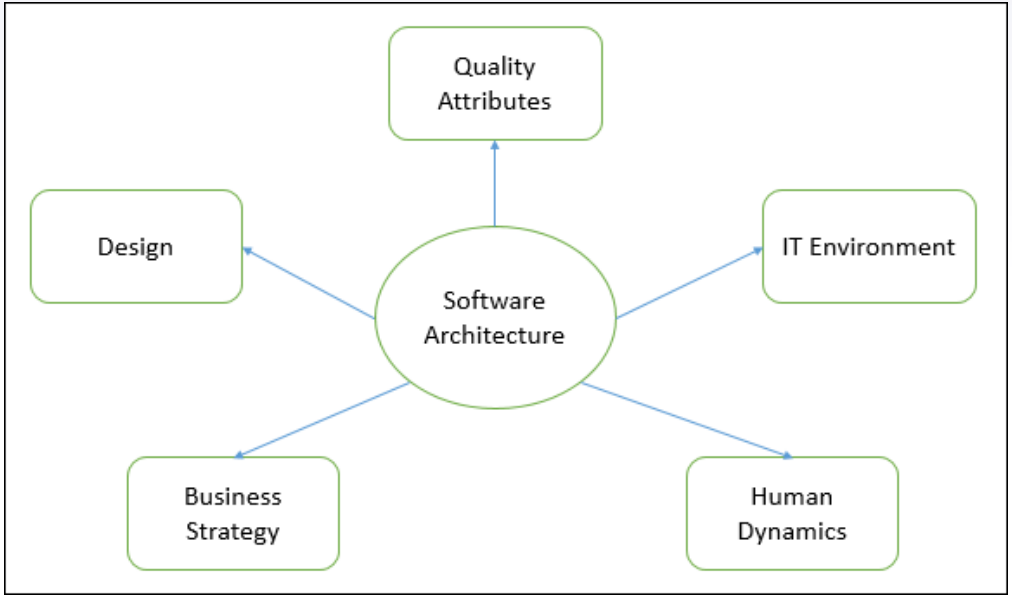
STDC, CDAC, Kochi

Sunitha C S
Joint Director
STDC, Kochi

Design and Architecture

The architecture of a system describes its major components, their relationships (structures), and how they interact with each other.

Software architecture and design includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.



IEEE defines architectural design as **“the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.”** The software that is built for computer-based systems can exhibit one of these many architectural styles.

Each style will describe a system category that consists of :

- A set of components(eg: a database, computational modules) that will perform a function required by the system.
- The set of connectors will help in coordination, communication, and cooperation between the components.
- Conditions that how components can be integrated to form the system.
- Semantic models that help the designer to understand the overall properties of the system.

Software architectures can be designed at **two levels of abstraction**:

- **Architecture in the small** is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.
- **Architecture in the large** is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Three **advantages** of explicitly designing and documenting software architecture:

- **Stakeholder communication**: Architecture may be used as a focus of discussion by system stakeholders.
- **System analysis**: Well-documented architecture enables the analysis of whether the system can meet its non-functional requirements.
- **Large-scale reuse**: The architecture may be reusable across a range of systems or entire lines of products.

Software architecture is most often represented using simple, informal **block diagrams** showing entities and relationships. **Pros**: simple, useful for communication with stakeholders, great for project planning. **Cons**: lack of semantics, types of relationships between entities, visible properties of entities in the architecture.

Architectural design is a **creative process** so the process differs depending on the type of system being developed. However, a number of **common decisions** span all design processes and these decisions affect the non-functional characteristics of the system:

- Is there a generic application architecture that can be used?
- How will the system be distributed?
- What architectural styles are appropriate?
- What approach will be used to structure the system?
- How will the system be decomposed into modules?
- What control strategy should be used?
- How will the architectural design be evaluated?
- How should the architecture be documented?

The particular architectural style should depend on the **non-functional system requirements**:

- **Performance**: localize critical operations and minimize communications. Use large rather than fine-grain components.
- **Security**: use a layered architecture with critical assets in the inner layers.
- **Safety**: localize safety-critical features in a small number of sub-systems.
- **Availability**: include redundant components and mechanisms for fault tolerance.
- **Maintainability**: use fine-grain, replaceable components.

1 Architectural views

Each architectural model only shows one view or perspective of the system. It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network. For both design and documentation, you usually need to present multiple views of the software architecture.

4+1 view model of software architecture:

- A **logical** view, which shows the key abstractions in the system as objects or object classes.
- A **process** view, which shows how, at run-time, the system is composed of interacting processes.
- A **development** view, which shows how the software is decomposed for development.
- A **physical** view, which shows the system hardware and how software components are distributed across the processors in the system.
- Related using **use cases** or scenarios (+1).

Architectural patterns

Patterns are a means of representing, sharing and reusing knowledge. An architectural pattern is a **stylized description of a good design practice**, which has been tried and tested in different environments. Patterns should include information about when they are and when the are not useful. Patterns may be represented using tabular and graphical descriptions.

- Model-View-Controller
- Layered Architecture
- Repository Architecture
- Client-server Architecture
- Pipe and filter Architecture

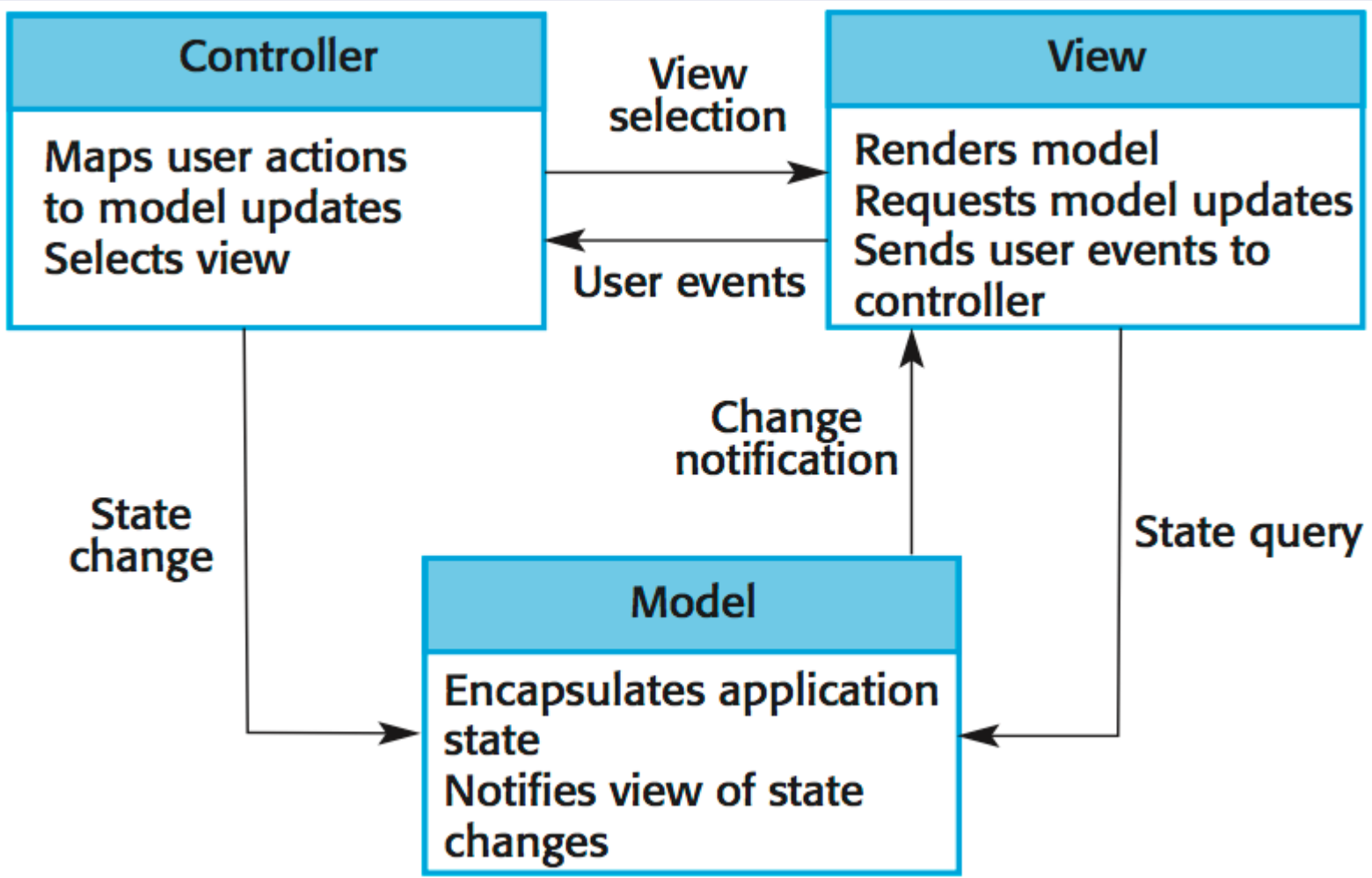
1. Model-View-Controller

Serves as a basis of interaction management in many web-based systems.

Decouples three major interconnected components:

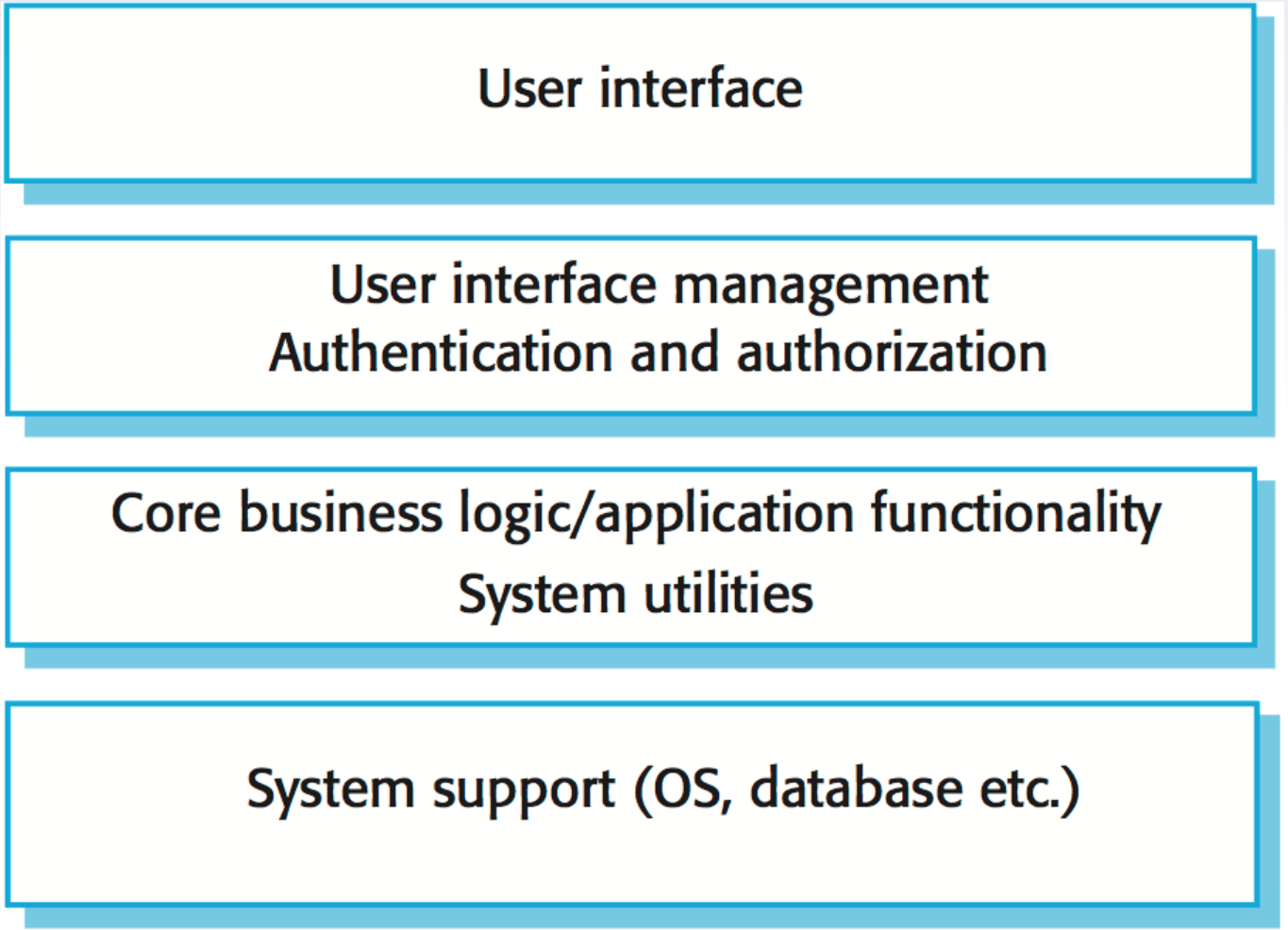
- The **model** is the central component of the pattern that directly manages the data, logic and rules of the application. It is the application's dynamic data structure, independent of the user interface.
- A **view** can be any output representation of information, such as a chart or a diagram. Multiple views of the same information are possible.
- The **controller** accepts input and converts it to commands for the model or view.

- Supported by most language frameworks.


Controller
Maps user actions to model updates
Selects view
View selection
User events
View
Renders model
Requests model updates
Sends user events to controller
State change
Change notification
State query
Model
Encapsulates application state
Notifies view of state changes

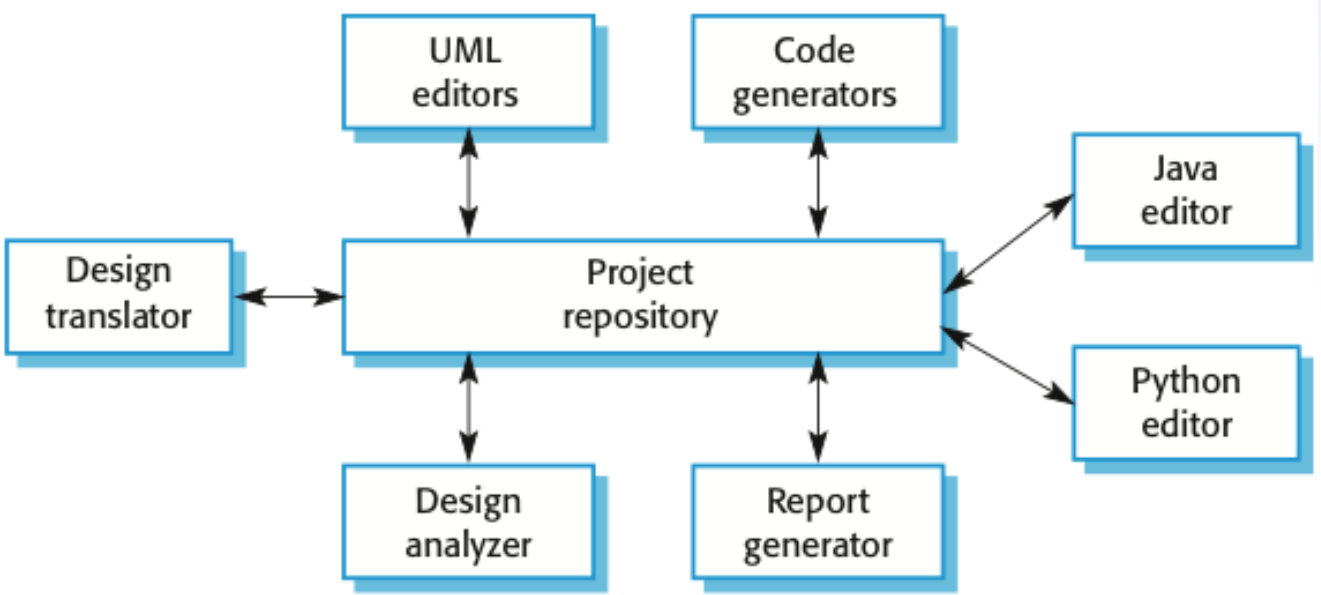
2. Layered Architecture

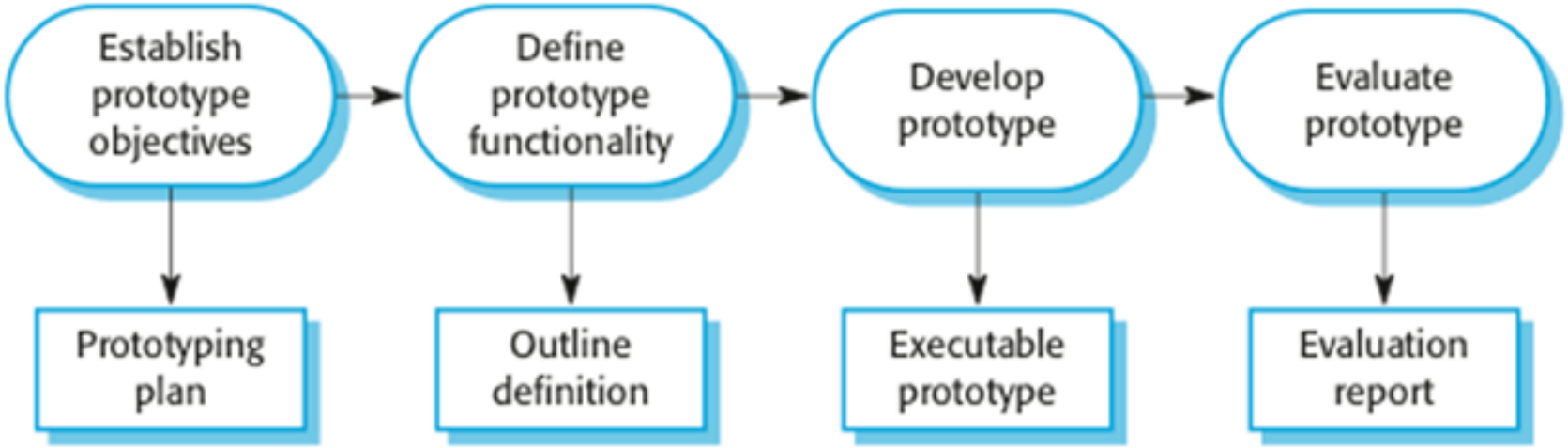
- Used to model the **interfacing of sub-systems**.
- Organizes the **system into a set of layers** (or abstract machines) each of which provide a set of services.
- Supports the **incremental development** of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, often artificial to structure systems in this way.


User interface
User interface management
Authentication and authorization
Core business logic/application functionality
System utilities
System support (OS, database etc.)

3. Repository Architecture

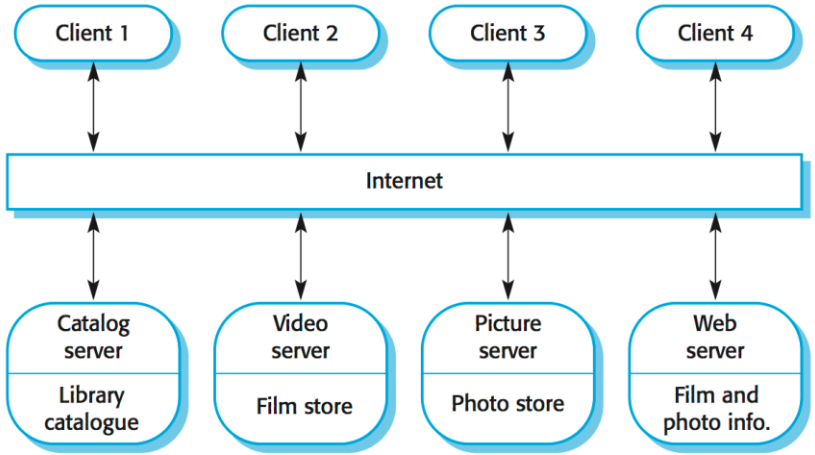
- Sub-systems must **exchange data**. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used a this is an **efficient data sharing mechanism**.


UML editors
Code generators
Java editor
Design translator
Project repository
Python editor
Design analyzer
Report generator


Establish prototype objectives
Define prototype functionality
Develop prototype
Evaluate prototype
Prototyping plan
Outline definition
Executable prototype
Evaluation report

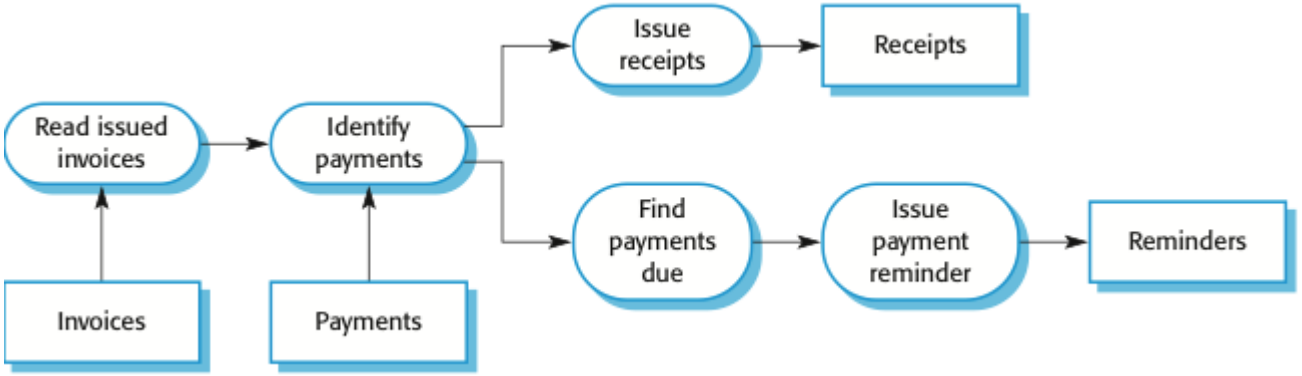
4. Client-server Architecture

- Distributed system model which shows how data and processing is distributed across a range of components, but can also be implemented on a single computer.
- Set of stand-alone servers which provide specific services such as printing, data management, etc.
- Set of clients which call on these services.
- Network which allows clients to access servers.



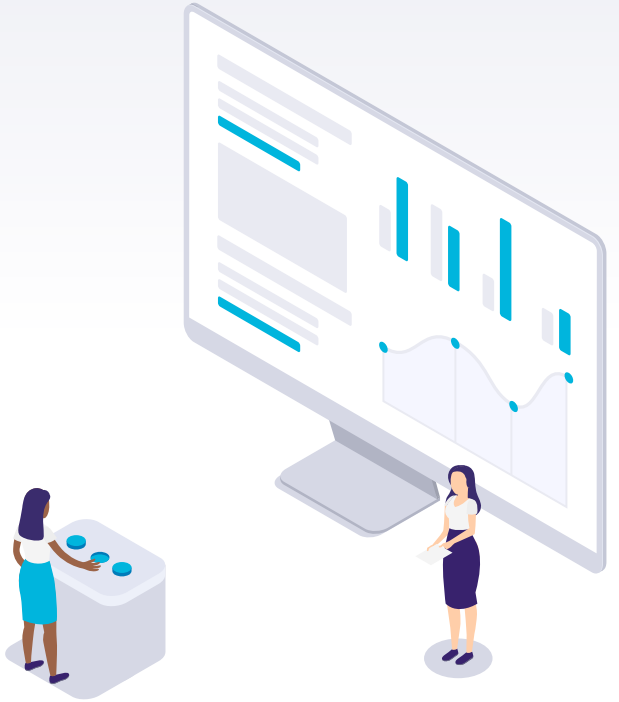
5. Pipe and filter architecture

- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.



2

Characteristics of Good Design



- Six characteristics of good software design**—simplicity, coupling, cohesion, information hiding, performance, and security**

Modularity

- Software engineering modularity allows typical applications to be divided into modules, as well as integration with similar modules, which helps developers use prewritten code. Modules are divided based on functionality, and programmers are not involved with the functionalities of other modules. Thus, new functionalities may be easily programmed in separate modules.

Cohesion

- In computer programming, **cohesion** refers to the degree to which the elements inside a module belong together. In one sense, it is a measure of the strength of relationship between the methods and data of a class and some unifying purpose or concept served by that class.

3

UML

UML

UML (Unified Modeling Language) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.

- UML is different from the other common programming languages such as C++, Java, COBOL, etc.
- UML is a pictorial language used to make software blueprints.
- UML can be described as a general purpose visual modeling language to visualize, specify, construct, and document software system.
- Although UML is generally used to model software systems, it is not limited within this boundary. It is also used to model non-software systems as well. For example, the process flow in a manufacturing unit, etc.

Object-Oriented Concepts

- UML can be described as the successor of object-oriented (OO) analysis and design.
- Objects are the real-world entities that exist around us and the basic concepts such as abstraction, encapsulation, inheritance, and polymorphism all can be represented using UML.
- The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

UML - An Overview

- The first thing to notice about the UML is that there are a lot of different diagrams (models) to get used to. The reason for this is that it is possible to look at a system from many different viewpoints.
- All of these people are interested in different aspects of the system, and each of them require a different level of detail. For example, a coder needs to understand the design of the system and be able to convert the design to a low level code.

A software development will have many stakeholders playing a part.

For Example:

- Analysts
- Designers
- Coders
- Testers
- QA
- The Customer
- Technical Authors

The **elements in a structure diagram** represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts, there are **seven types of structure diagram** as follows:

- Class Diagram
- Component Diagram
- Deployment Diagram
- Object Diagram
- Package Diagram
- Composite Structure Diagram
- Profile Diagram

Behavior diagrams show the **dynamic behavior** of the objects in a system, which can be described as a series of changes to the system over **time**, there are **seven types of behavior diagrams** as follows:

- Use Case Diagram
- Activity Diagram
- State Machine Diagram
- Sequence Diagram
- Communication Diagram
- Interaction Overview Diagram
- Timing Diagram

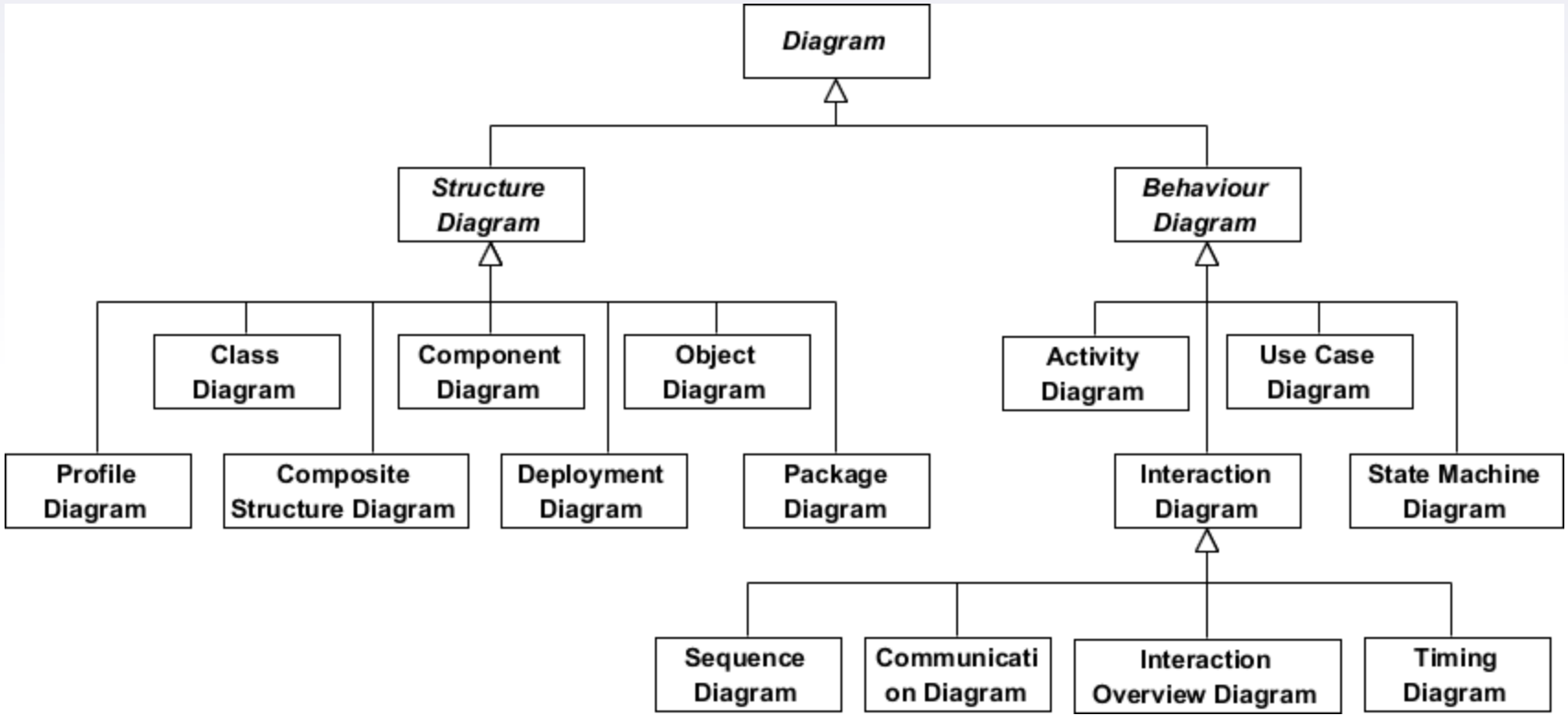


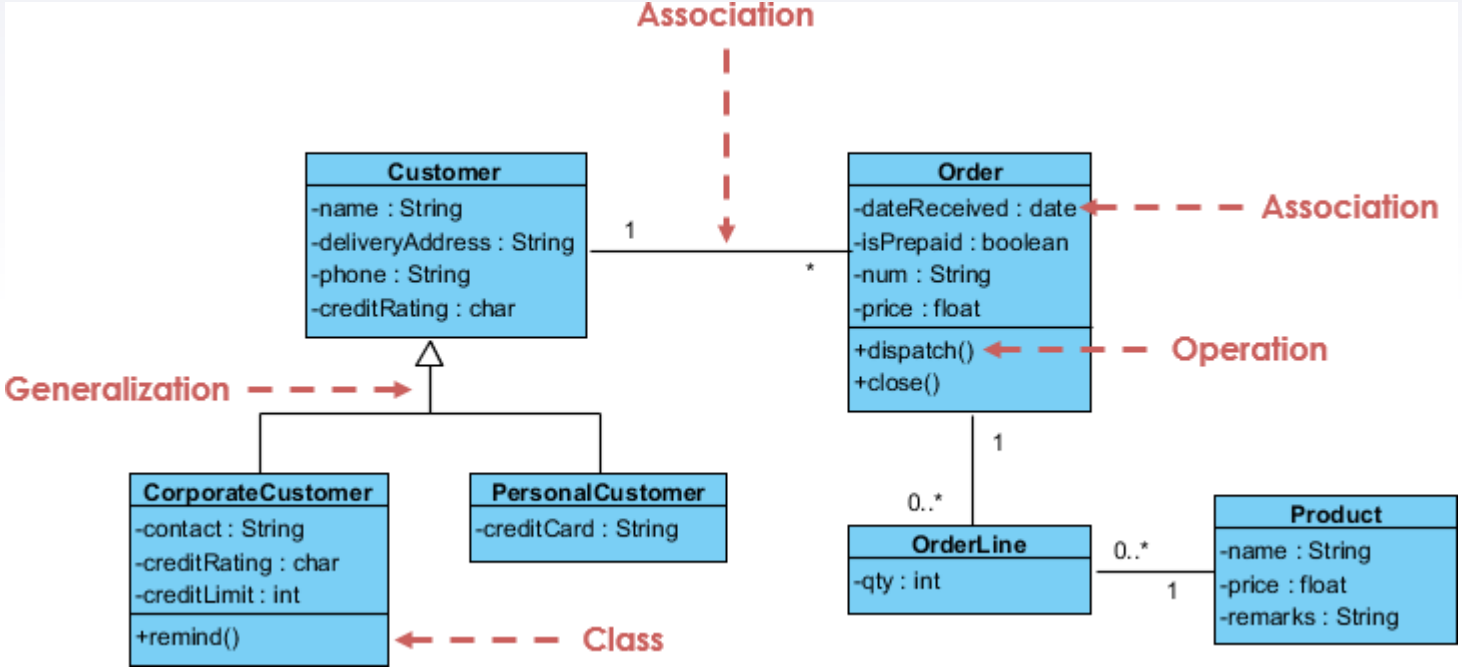
Diagram
Structure Diagram
Behaviour Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
Use Case Diagram
Profile Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Interaction Diagram
State Machine Diagram
Sequence Diagram
Communicati on Diagram
Interaction Overview Diagram
Timing Diagram

What is a Class Diagram?

- The class diagram is a central modeling technique that runs through nearly all object-oriented methods. This diagram describes the types of objects in the system and various kinds of static relationships which exist between them.

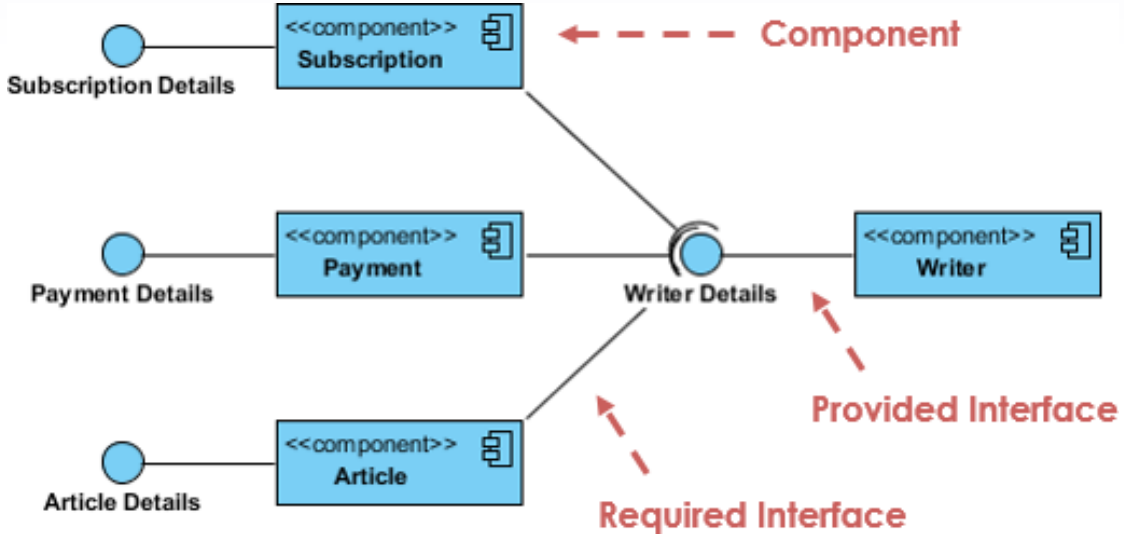
Relationships

- There are three principal kinds of relationships which are important:
- **Association** - represent relationships between instances of types (a person works for a company, a company has a number of offices.
- **Inheritance** - the most obvious addition to ER diagrams for use in OO. It has an immediate correspondence to inheritance in OO design.
- **Aggregation** - Aggregation, a form of object composition in object-oriented design.


Association
Customer
-name : String
-deliveryAddress : String
-phone : String
-creditRating : char
1
*
Order
-dateReceived : date
-isPrepaid : boolean
-num : String
-price : float
+dispatch()
+close()
Association
Operation
Generalization
1
0..*
CorporateCustomer
-contact : String
-creditRating : char
-creditLimit : int
+remind()
PersonalCustomer
-creditCard : String
OrderLine
-qty : int
0..*
1
Product
-name : String
-price : float
-remarks : String
Class

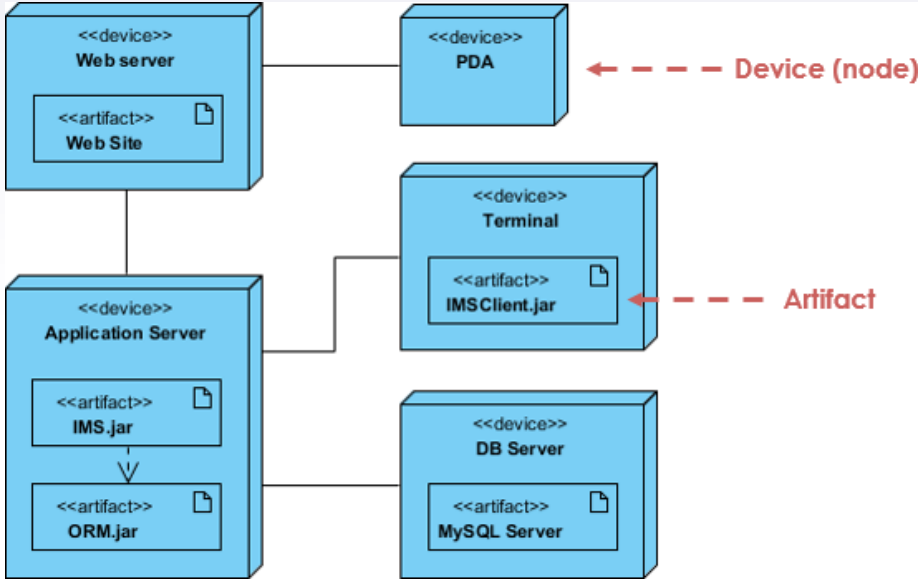
Component Diagram

- In the Unified Modeling Language, a component diagram depicts how components are wired together to form larger components or software systems. It illustrates the architectures of the software components and the dependencies between them. Those software components including run-time components, executable components also the source code components.



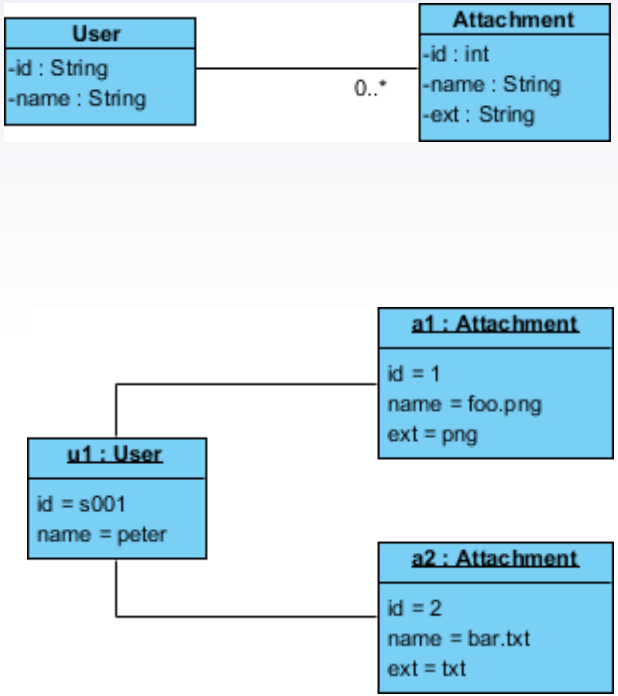
Deployment Diagram

- It is a structure diagram which shows architecture of the system as deployment (distribution) of software artifacts to deployment targets. Artifacts represent concrete elements in the physical world that are the result of a development process. It models the run-time configuration in a static view and visualizes the distribution of artifacts in an application. In most cases, it involves modeling the hardware configurations together with the software components that lived on.



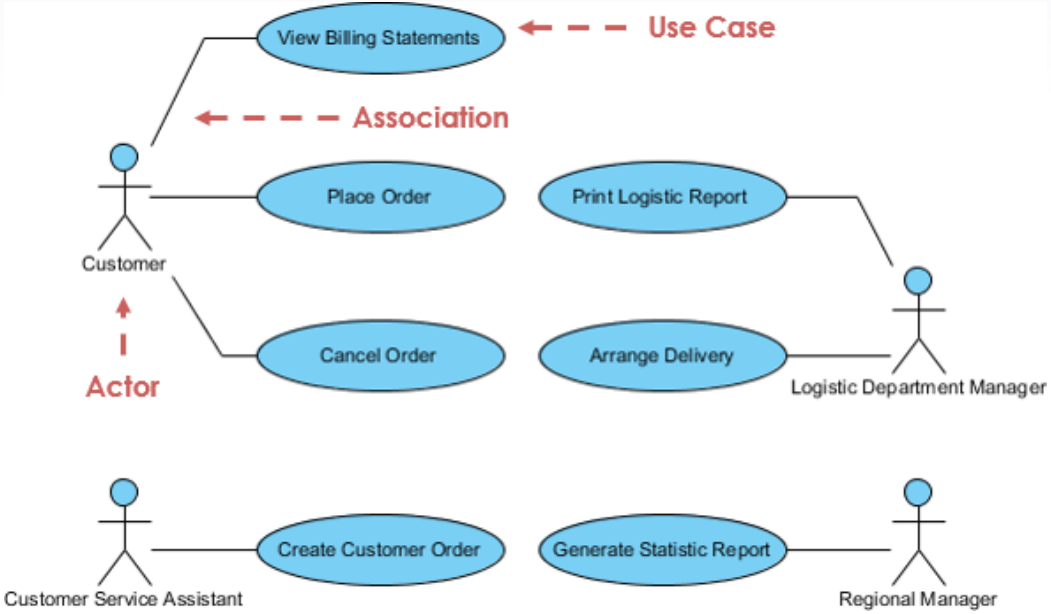
Object Diagram

- An object diagram is a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time. The difference is that a class diagram represents an abstract model consisting of classes and their relationships. However, an object diagram represents an instance at a particular moment, which is concrete in nature. The use of object diagrams is fairly limited, namely to show examples of data structure.



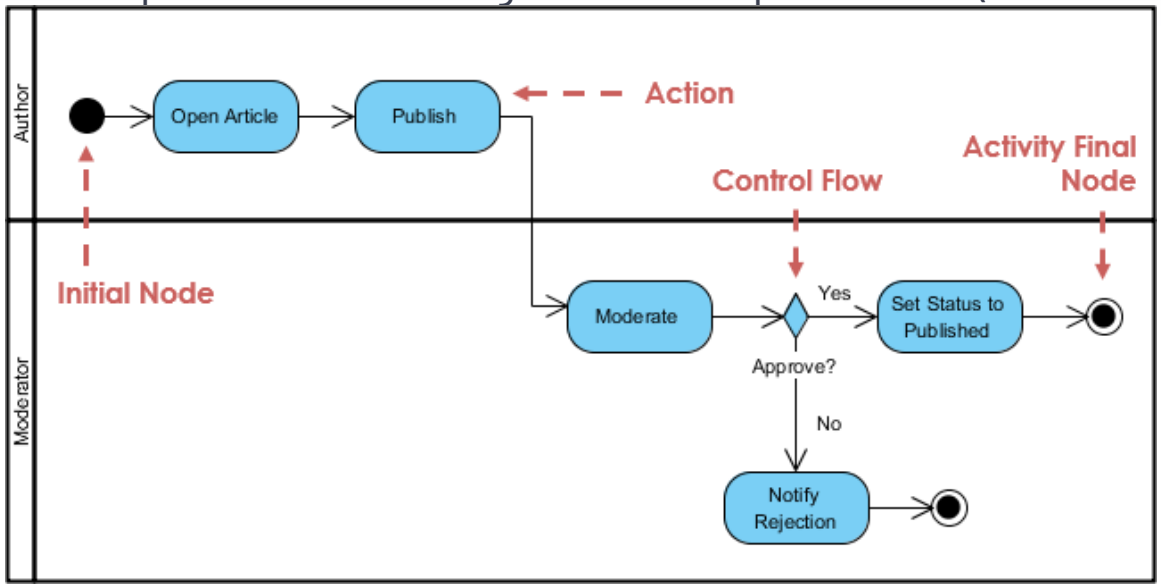
What is a Use Case Diagram?

A use-case model describes a system's functional requirements in terms of use cases. It is a model of the system's intended functionality (use cases) and its environment (actors). Use cases enable you to relate what you need from a system to how the system delivers on those needs.



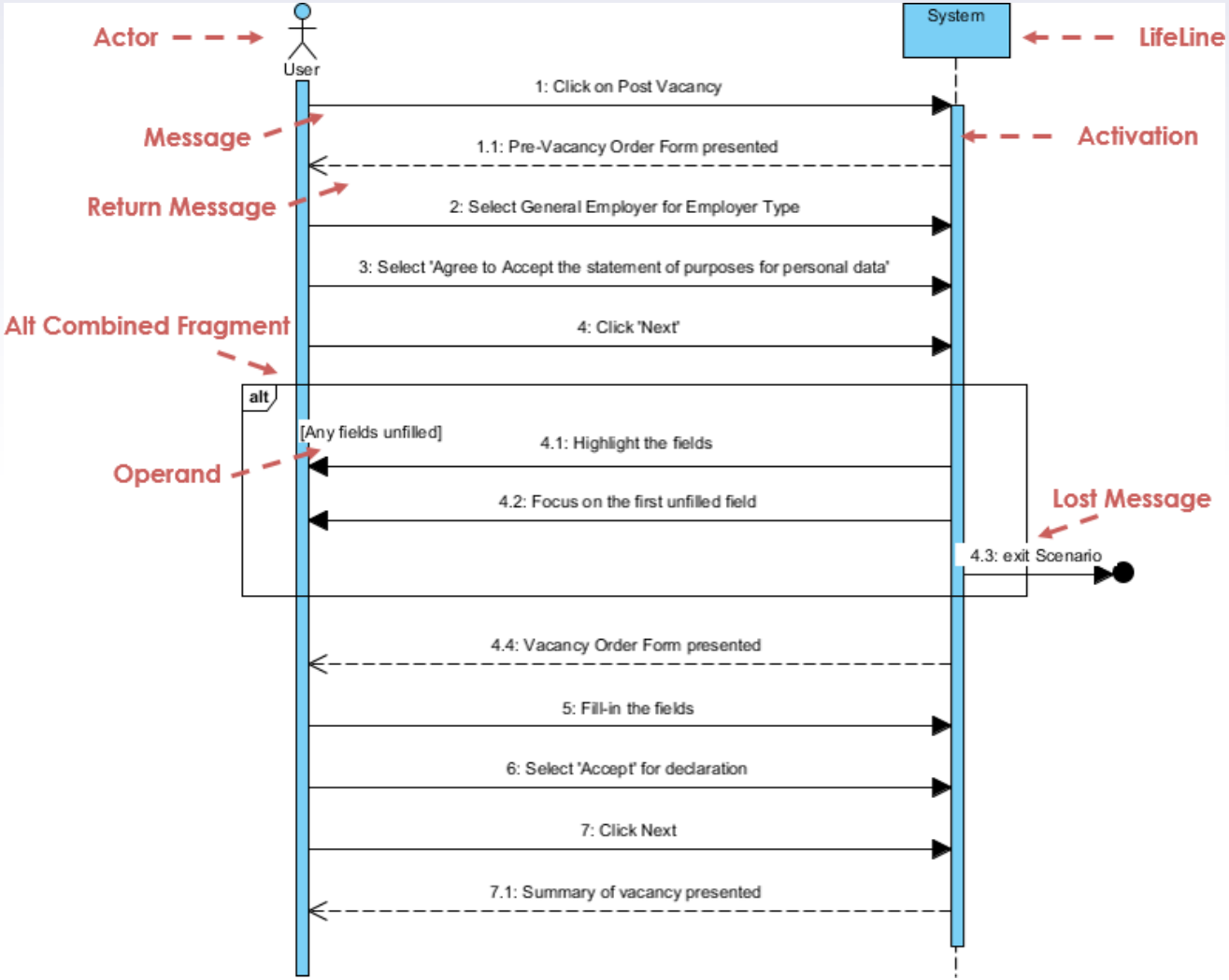
Activity Diagram

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. It describes the flow of control of the target system, such as the exploring complex business rules and operations, describing the use case also the business process. In the Unified Modeling Language, activity diagrams are intended to model both computational and organizational processes (i.e. workflows).



Sequence Diagram

The Sequence Diagram models the collaboration of objects based on a time sequence. It shows how the objects interact with others in a particular scenario of a use case. With the advanced visual modeling capability, you can create complex sequence diagram in few clicks. Besides, some modeling tool such as Visual Paradigm can generate sequence diagram from the flow of events which you have defined in the use case description.


Actor
User
System
LifeLine
1: Click on Post Vacancy
Message
1.1: Pre-Vacancy Order Form presented
Activation
Return Message
2: Select General Employer for Employer Type
3: Select 'Agree to Accept the statement of purposes for personal data'
Alt Combined Fragment
4: Click 'Next'
alt
[Any fields unfilled]
4.1: Highlight the fields
Operand
4.2: Focus on the first unfilled field
Lost Message
4.3: exit Scenario
4.4: Vacancy Order Form presented
5: Fill-in the fields
6: Select 'Accept' for declaration
7: Click Next
7.1: Summary of vacancy presented