(CT-21015) Software Engineering Mini Project Stage- II

Unit III - System Architecture and Design Overview

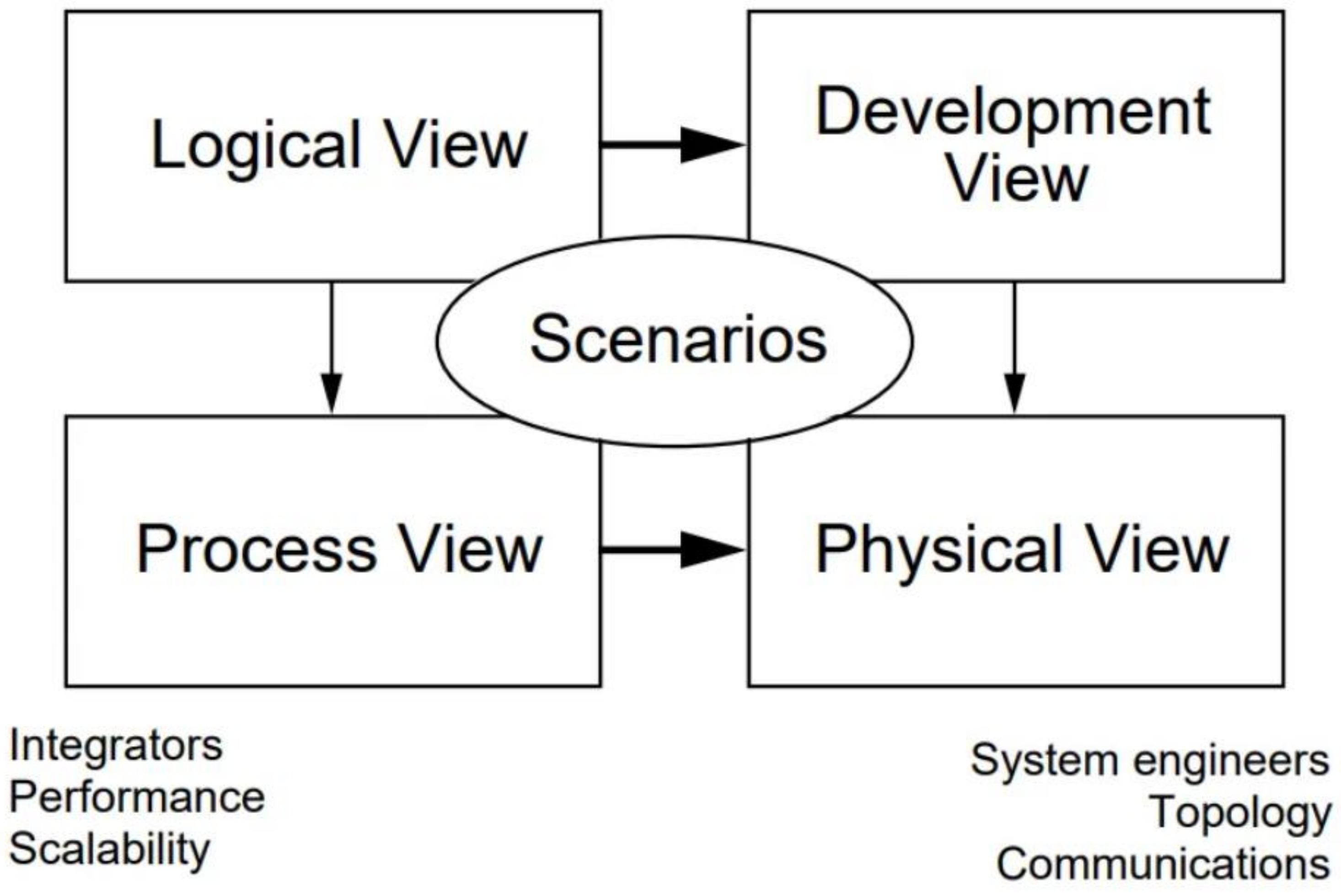
Unit III - System Architecture and Design Overview

Contents

4+1 View Model of Software Architecture

4+1 is a view model used for "describing the architecture of software-intensive systems, based on the use of multiple, concurrent views".

The views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers, system engineers, and project managers.



4+1 View Model of Software Architecture

- The **4+1 View Model** was introduced by **Philippe Kruchten** to describe software architecture from different perspectives. It helps address concerns of different stakeholders (developers, architects, users, system engineers, etc.) by organizing architectural descriptions into **five views**.

The Five Views

1. **Logical View** (What the system should do - Functional Requirements)
2. **Development View** (How the system is built - Code Organization)
3. **Process View** (How the system runs - Performance, Scalability, and Concurrency)
4. **Physical View** (How the system is deployed - Hardware and Network Topology)
5. **Scenarios (Use Case View) - The +1 View** (Validates the architecture with real-world use cases)

1. Logical View (Functional View)

- **Purpose:** Represents the system’s functionality from a **developer** and **end-user perspective**.
- **Elements:** Classes, objects, relationships, interactions, and key abstractions.
- **UML Diagrams Used:** Class diagrams, Object diagrams, Sequence diagrams.
- **Example**: A Banking System’s logical view includes entities like Account, Customer, Transaction, etc.

2. Development View (Implementation View)

- **Purpose:** Shows the system's internal **code structure** and how it is divided into modules and components.
- **Focus:** Organization of software in terms of **packages, libraries, and layers**.
- **UML Diagrams Used:** Component diagrams, Package diagrams.
- **Example:** A Microservices architecture may have different services like User Service, Payment Service, and Order Service in separate repositories.

3. Process View (Concurrency View)

- **Purpose:** Describes how the system behaves at runtime, handling **performance, scalability, and concurrency**.
- **Focus:** Threads, processes, communication, inter-process interactions.
- **UML Diagrams Used:** Activity diagrams, Sequence diagrams, State Machine diagrams.
- Example: A real-time video streaming application requires multiple threads/processes for buffering, compression, rendering, and network communication.

4. Physical View (Deployment View)

- **Purpose:** Represents **hardware deployment** and **physical nodes** where the system runs.
- **Focus:** Servers, databases, network topology, cloud architecture.
- **UML Diagrams Used:** Deployment diagrams.
- **Example**: A **cloud-based system** may have services **deployed on AWS**, with **load balancers**, **databases**, and **microservices** distributed across different regions.

5. Scenarios (Use Case View - The +1)

- **Purpose: Validates the architecture** by applying real-world use cases.
- **Focus:** Key scenarios, end-to-end system interactions.
- **UML Diagrams Used:** Use Case diagrams, Sequence diagrams.
- **Example**: A **user logging into an e-commerce site**, adding items to the cart, and making a payment.

Summary of 4+1 View Model

View	Focus	Stakeholders	UML Diagrams
Logical View	Functionality, objects, relationships	Developers, End Users	Class, Object, Sequence
Development View	Code structure, modules, layers	Developers	Component, Package
Process View	Performance, concurrency, execution flow	System Engineers	Activity, Sequence, State Machine
Physical View	Deployment, hardware, servers	DevOps, IT Team	Deployment
Scenarios (+1 View)	Use cases, validation	Analysts, Testers	Use Case, Sequence

Case Study: 4+1 View Model for An E-Commerce System

Let’s apply the **4+1 View Model** to an **E-Commerce System**, such as **Amazon or Flipkart**, where users can browse products, add them to a cart, and complete purchases.

Logical View (Functional View)

- What it Represents -The Logical View focuses on the system's functionality and how objects interact. It defines key modules and relationships.
- Key Components
    - User → Registers, logs in, and makes purchases.
    - Product → Represents items for sale, with details like name, price, and stock.
    - Cart → Holds selected products before checkout.
    - Order → Represents a completed purchase, including payment and shipping details.
    - Payment → Handles transactions via credit cards, PayPal, or other methods.
    - Inventory → Tracks stock levels for each product.
- UML Representation
    - Class Diagram → Defines relationships between User, Cart, Order, and Product.
    - Sequence Diagram → Shows user interactions from adding a product to checkout.
- 📌 Example: A User adds a Product to a Cart, then completes an Order via Payment.

Development View (Implementation View)

- What it Represents -This view describes the software structure in terms of components, modules, and packages.
- Key Components
  - Frontend (React, Angular) → Handles UI/UX.
  - Backend (Spring Boot, Node.js) → Manages business logic and API communication.
  - Database (MySQL, MongoDB) → Stores user data, products, orders.
  - Microservices:
    - User Service (Handles authentication & profile)
    - Product Service (Manages product catalog)
    - Order Service (Processes orders)
    - Payment Service (Handles transactions)
- UML Representation
  - Component Diagram → Shows how modules interact.
  - Package Diagram → Groups related classes into subsystems.
- 📌 Example: The Order Service calls the Payment Service to process a transaction.

Process View (Concurrency & Performance)

- What it Represents-The Process View focuses on system performance, parallel processing, and reliability.
- Key Aspects
  - Concurrency: Multiple users placing orders simultaneously.
  - Scalability: Load balancers distributing requests across multiple backend servers.
  - Fault Tolerance: If the Payment Service fails, orders go into a retry queue.
  - Asynchronous Processing: Background jobs update stock levels after purchases.
- UML Representation
  - Activity Diagram → Illustrates concurrent tasks (e.g., payment processing).
  - Sequence Diagram → Shows interactions between services (e.g., checkout flow).
- 📌 Example: A user places an order, and while the payment is processed, an inventory update job runs in the background.

Physical View (Deployment View)

- What it Represents-The Physical View maps software components to the hardware environment.
- Deployment Setup
  - Cloud-based Deployment (AWS, Google Cloud)
  - Load Balancer → Distributes requests across multiple backend servers.
  - Web Server (Nginx, Apache) → Serves static files and API requests.
  - Database Cluster (Primary + Replica Nodes) → Handles high availability.
  - CDN (Content Delivery Network) → Caches product images for faster access
- UML Representation
  - Deployment Diagram → Shows how services are deployed across servers.
- 📌 Example: When a user in the US accesses the site, product images are served via a nearby CDN server, reducing latency.

Scenarios View (+1 View)

What it Represents-This view validates the system by defining key use cases.

Use Case: Placing an Order

1. **User logs in** → Authentication Service verifies credentials.
2. **User adds items to the cart** → Cart Service updates the session.
3. **User checks out** → Order Service creates an order.
4. **Payment is processed** → Payment Service interacts with a third-party gateway (e.g., Stripe, PayPal).
5. **Order is confirmed** → User gets an email, and Inventory Service updates stock.
6. **Shipment processing** → Logistics Service schedules delivery.

UML Representation

- **Use Case Diagram** → Represents user interactions.
- **Sequence Diagram** → Shows the order-processing flow.
- 📌 **Example**: If **payment fails**, the system **sends an error message** and allows a retry.

Summary Table: 4+1 View for E-Commerce

View	Focus	Example for E-Commerce	UML Used
Logical View	Functional modules	Users, Products, Orders, Payments	Class, Sequence
Development View	Code organization	Microservices (User, Product, Order)	Component, Package
Process View	Performance, concurrency	Payment processing, load balancing	Activity, Sequence
Physical View	Deployment, servers	AWS, Load Balancers, Databases	Deployment
Scenarios View	Use cases, validation	Placing an order, handling failures	Use Case, Sequence

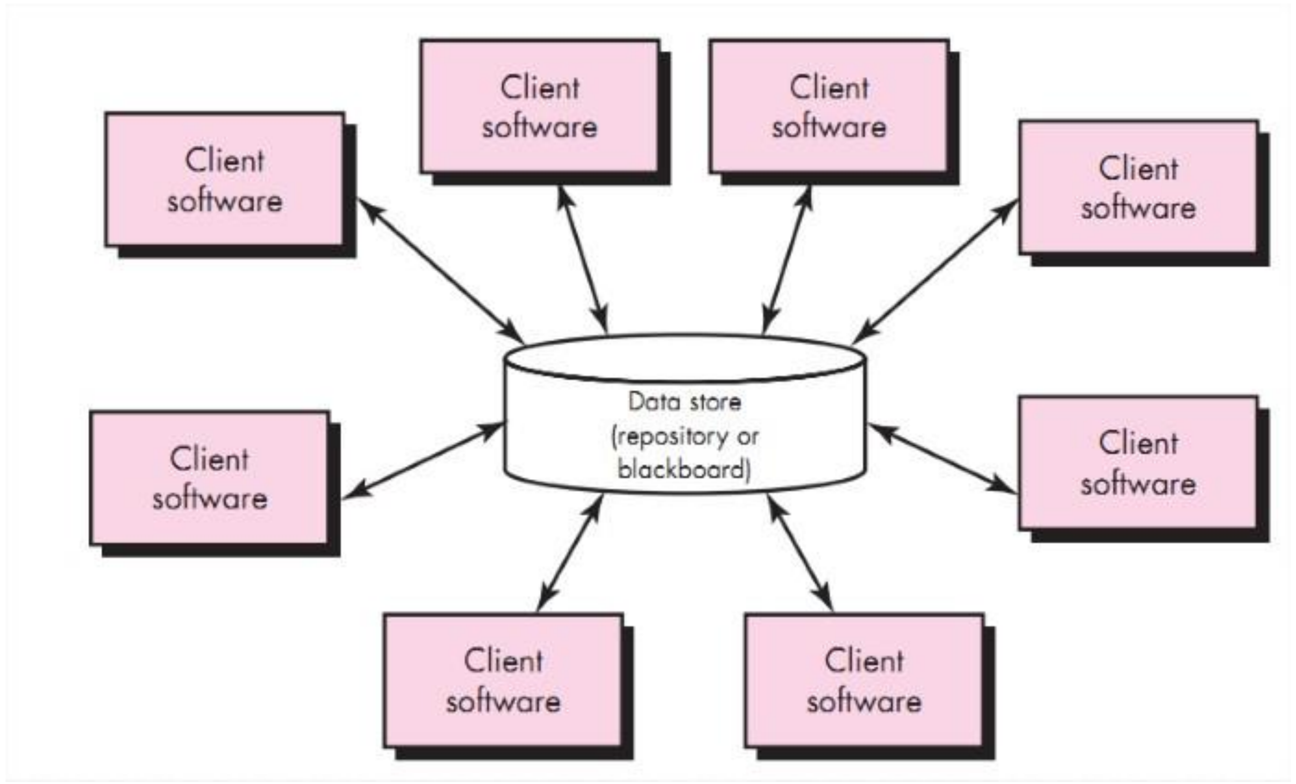
Architecture Styles

Architecture styles

- Software architecture **is a blueprint of step-by-step instructions in the development of software**. It aids in determining the true purpose of the project because **it contains detailed descriptions of the functionalities** that will be provided by the software project. It also helps with the proper planning of all necessary
- Architecture styles give a specific solution to a particular software which further particularly helps in the organization of the code.
  - Data-centered Architecture,
  - Data-flow Architecture,
  - Call and Return Architecture
  - Object-oriented Architecture
  - Layered Architecture

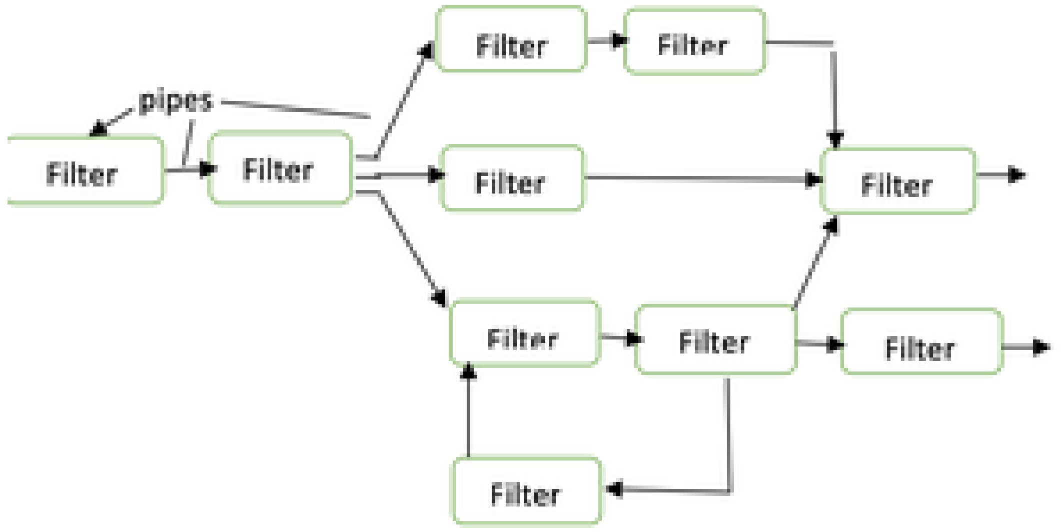
1) Data-centered architecture:

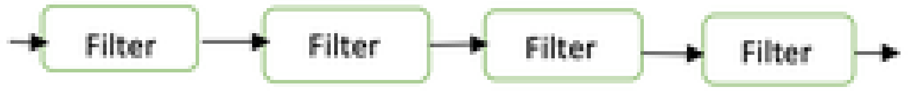
- A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete, or modify the data present within the store.
- The client software accesses a **central repository**.
- This means that existing components can be updated while new client components can be added to the architecture without affecting other clients.
- The blackboard technique allows data to be exchanged between clients.
- Client work independent of each other



2) Data-flow architecture:

- It represents pipe-and-filter architecture
- Used when input data is transformed into output data through a series of computational manipulative components.
- Pipes are used to transmitting data from one component to the next.
- Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form
- If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.



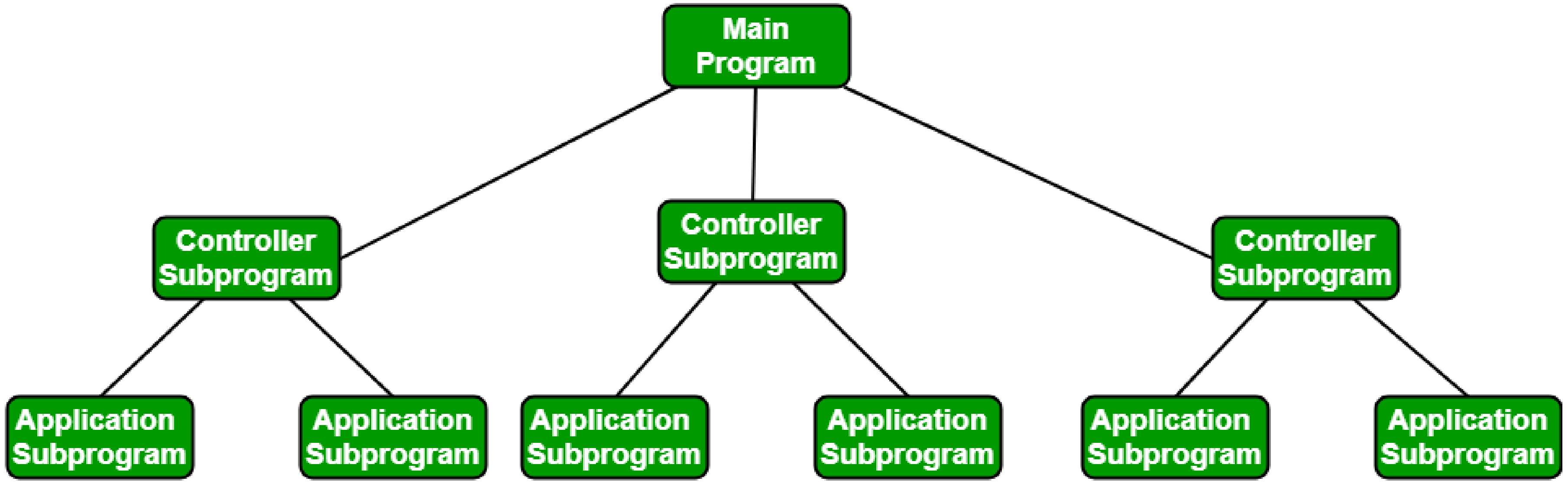


3) Call and Return architectures

It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.

- **Main program or Subprogram architectures:** The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components.

- **Remote procedure call architecture:** This components is used to present in a main program or sub program architecture distributed among multiple computers on a network.



4) Object Oriented architecture

The components of a system encapsulate data and the operations that must be applied to manipulate the data.

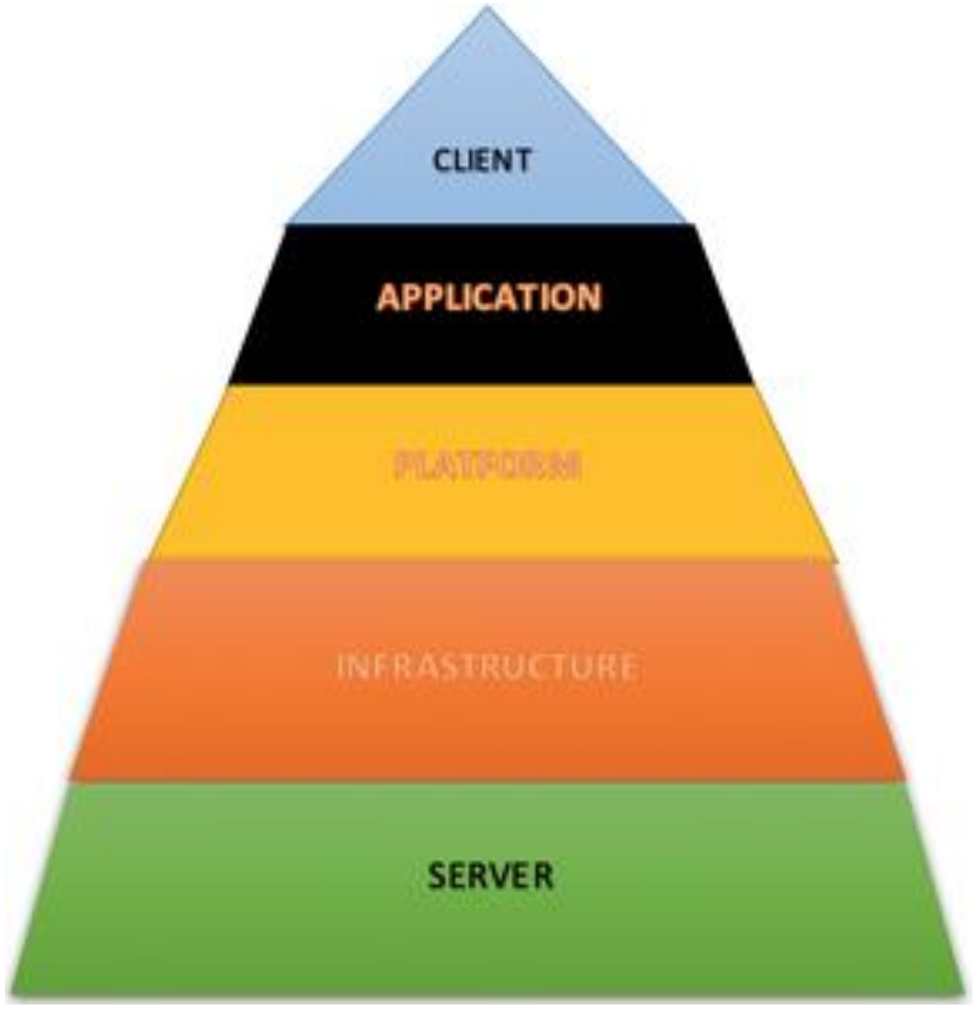
The coordination and communication between the components are established via the message passing.

Characteristics of Object-Oriented architecture:

- Object protect the system’s integrity.
- An object is unaware of the depiction of other items.

5) Layered architecture

- A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that becomes closer to machine instruction set progressively.
- At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing(communication and coordination with OS)
- Intermediate layers to utility services and application software functions.
- One common example of this architectural style is OSI-ISO (Open Systems Interconnection-International Organisation for Standardisation) communication system.



Software Design Process – Software Engineering

- The **design phase** translates customer requirements from the **Software Requirements Specification (SRS)** into a blueprint for implementation.
- It defines the system's structure, behavior, and interaction, ensuring that the software meets functional and non-functional requirements.
- The software design process is structured into three key levels:

    1. Interface Design
    2. Architectural Design
    3. Detailed Design

1. Interface Design

Objective: Define how system components and users interact.

Types of Interfaces:

- **User Interface (UI)**: Ensures a smooth and intuitive experience for users (GUI, CLI, web interfaces).
- **System Interface**: Defines interactions between different subsystems or external systems (APIs, web services).
- **Hardware Interface**: Specifies how the software interacts with physical components (sensors, devices).
- **Communication Interface**: Defines protocols for data exchange (REST, SOAP, message queues).

Key Considerations:

- Usability and accessibility
- Consistency in UI elements
- API design principles (RESTful, GraphQL)
- Security and authentication mechanisms (OAuth, JWT)

2. Architectural Design

Objective: Define the overall system structure, including high-level components and their interactions.

Common Architectural Patterns:

- **Layered Architecture** (Presentation, Business Logic, Data)
- **Client-Server Architecture**
- **Model-View-Controller (MVC)**
- **Microservices Architecture**
- **Event-Driven Architecture**
- **Service-Oriented Architecture (SOA)**

Key Considerations:

- Scalability and performance
- Maintainability and modularity
- Security and fault tolerance
- Technology stack selection

3. Detailed Design

Objective: Specify internal logic, algorithms, data structures, and interactions within modules.

Detailed Design Aspects:

- **Class Diagrams** (Object-oriented design)
- **Sequence Diagrams** (Interaction flows)
- **State Machines** (Object lifecycle)
- **Database Design** (ER diagrams, normalization)
- **Algorithm Design** (Sorting, searching, encryption)

Key Considerations:

- Code reusability and modularity
- Performance optimization
- Error handling and logging
- Integration with third-party services

Software Quality

Software quality refers to the degree to which software conforms to its requirements and meets the needs of its users. It is formally defined as “the capability of a software product to satisfy stated and implied needs when used under specified conditions.” Another definition states that software quality depends on “the degree to which those established requirements accurately represent stakeholder needs, wants, and expectations.” High quality software meets its requirements, which in turn should accurately reflect stakeholder needs. Quality is about aligning the software with both its formal requirements as well as true user needs.

Main Challenges Faced in Ensuring Software Quality

Difficulty in clearly defining requirements – If requirements are unclear, incomplete, or ambiguous, it becomes extremely difficult to determine whether the final software product meets expectations.

Maintaining effective communication with stakeholders – Software quality needs and priorities often change over time as stakeholders gain more understanding of the system under development. Constant communication and collaboration with stakeholders are key to ensure the software developers are keeping pace with evolving needs.

Deviations from specifications – If the software substantially deviates from original specifications, this directly undermines overall quality standards. Any deviations should be formally approved through change management processes.

Architecture and design errors – Flaws in the fundamental architecture and design cascade down into the actual software development process and software application. It becomes very expensive to remedy architectural issues late in development.

Coding errors – Bugs introduced into source code during coding and implementation activities that deviate from coding standards further degrade quality. Unit testing and code reviews aim to catch these issues early.

Non-compliance with current processes/procedures – When schedules slip, teams often ignore defined processes and take shortcuts. But straying from disciplined, proven software processes increases risk.

Inadequate work product reviews and tests – Without sufficient peer reviews, testing tool for functional testing, regression testing, QA testing, and other verification activities, defects persist into final product. Rigorous quality control is essential.

Documentation errors – Incorrect, outdated, or altogether missing documentation makes it difficult for a software engineer to track issues, onboard a QA team, and evolve the system.

Difference between an error, a defect, and a failure?

The terminology around software anomalies can be confusing. It’s important to distinguish the subtle differences between errors, defects and failures.

Error – This is a human mistake made by a QA engineer, software quality assurance analyst, tester or other stakeholder. An example is misunderstanding a requirement and coding to the wrong specification.

Defect – A defect is a flaw or imperfection inserted into a software work product due to an error. This could be a bug in the code or issues with other artifacts like requirements. Defects get inserted when errors are made.

Failure – A failure represents the termination of the software’s ability to function as intended. Failures occur when the software executing encounters a defect. Failures are user-facing; the user experiences the software failing in some unintended way.

- An error leads to the insertion of a defect, which in turn can lead to observable software failures upon execution. Engineers aim to prevent errors and remove defects before they turn into failures.

Software Quality_Management

Software quality management refers to the oversight, control, and coordination of policies, procedures, activities, and people to achieve quality objectives. Key elements include:

- **Quality planning** – Defining quality objectives, requirements, targets, and planning of quality assurance activities.
- **Quality control** – Techniques to measure quality characteristics, review work products, and find defects.
- **Quality assurance** – Processes and audits to ensure compliance with procedures.
- **Quality improvement** – Defect analysis and process enhancements to improve quality.
- **Resources** – Infrastructure, tools, training that enable quality processes.
- **Standards** – Regulations, models, certifications that guide quality work.
- **Culture** – Values, behaviors that encourage quality mindset.

How do you measure software quality?

Software quality measurement provides data to help assess current quality levels and drive improvement initiatives. Common measures include:

- Error density – errors per size of work product (requirements, design, code). Helps find problem areas.
- Defect density – defects per size of software. Used to gauge release readiness.
- Failure rate – mean time between failures. Tracks system reliability.
- Reliability models – estimate of future failures based on defect data. Predicts field quality.
- CoSQ – cost of software quality analysis. Justifies quality spending.
- Escaped defects – defects missed during development. Assesses testing effectiveness.

Measurements are used to monitor trends, compare benchmarks, predict failures, optimize testing, and prioritize improvements. Statistical analysis like Pareto charts helps interpret the data.

Analysis model to Design Model transformation

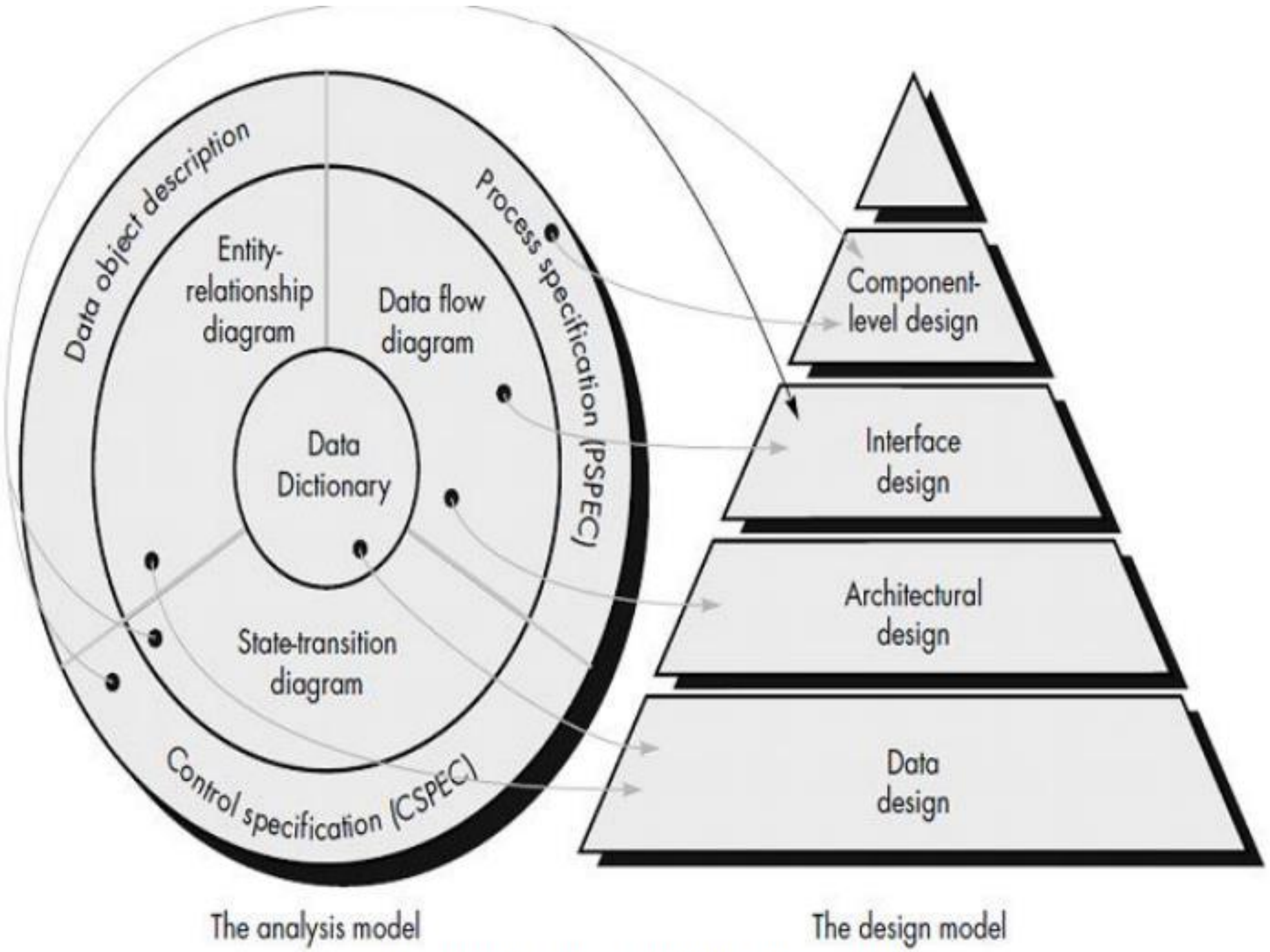


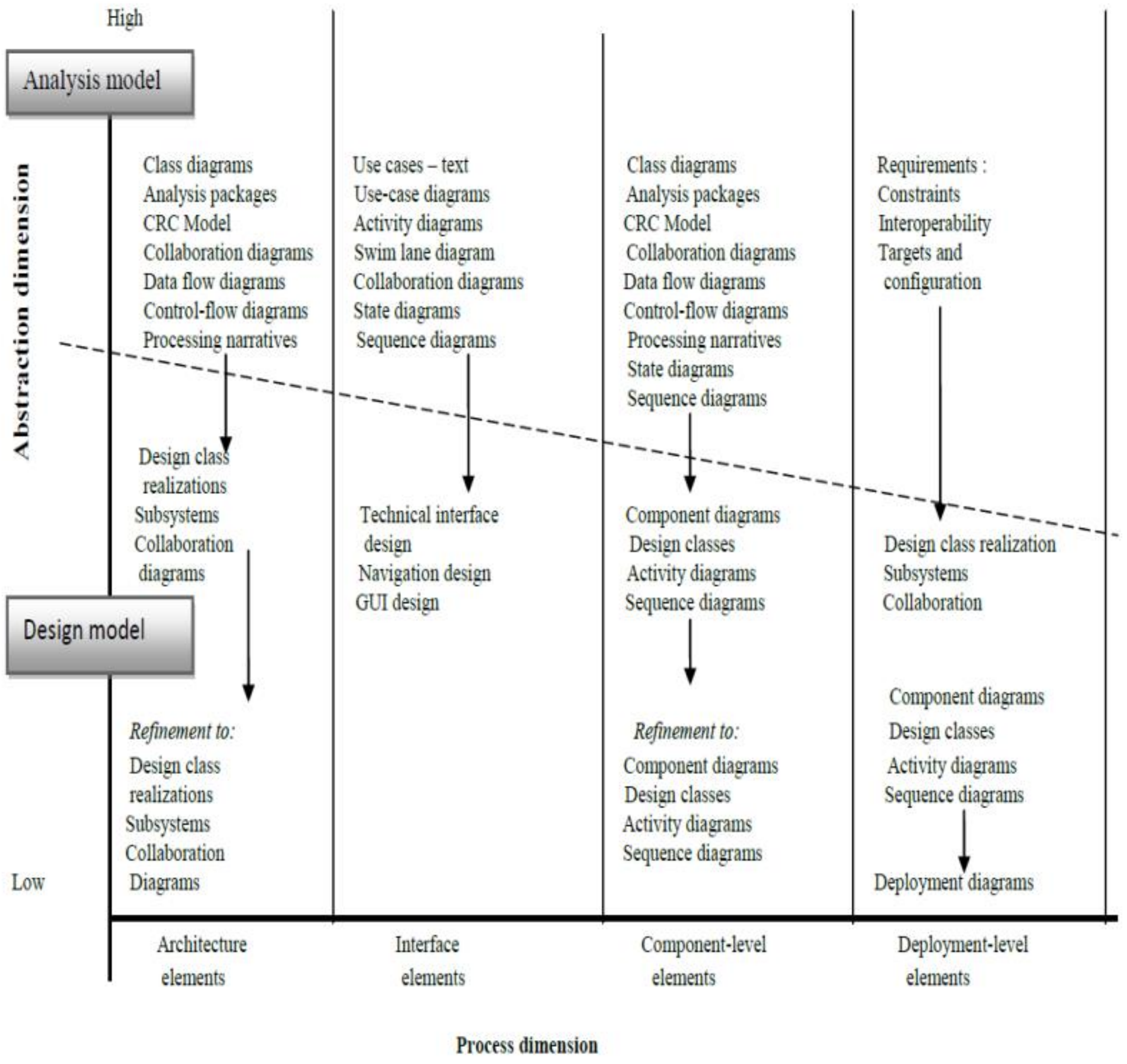
Figure 3. Translating the analysis model into a software design

Analysis model to Design Model transformation

- The transformation from an analysis model to a design model involves:
  - ✓ **Translating** the system's requirements and functionalities, initially described at a high level in the analysis model, **into** a concrete, implementable design, including data structures, architecture, interfaces, and components.

- **Analysis Model:**
  - Focuses on understanding the problem domain and defining the system's requirements.
  - Captures information, behavior, and functions of the system at a high level of abstraction.
  - Prioritizes understanding the "what" rather than the "how".
  - Examples include use case diagrams, data flow diagrams, and class diagrams.

- **Design Model:**
    - Specifies the implementation details of the system, including data structures, architecture, interfaces, and components.
    - Translates the analysis model into concrete design decisions, considering constraints and technical aspects.
    - Focuses on the "how" of implementing the system's requirements.
    - Examples include architectural diagrams, component diagrams, and database schemas.
- **Transformation Process:**
    - The transformation involves refining the analysis model to incorporate implementation details.
    - This may involve decomposing complex functionalities into smaller, manageable components.
    - The design model should be a clear and detailed representation of how the system will be built.
    - The transition from analysis to design should occur when the consequences of the implementation environment start to show.


High
Analysis model
Abstraction dimension
Class diagrams
Analysis packages
CRC Model
Collaboration diagrams
Data flow diagrams
Control-flow diagrams
Processing narratives
Use cases – text
Use-case diagrams
Activity diagrams
Swim lane diagram
Collaboration diagrams
State diagrams
Sequence diagrams
Class diagrams
Analysis packages
CRC Model
Collaboration diagrams
Data flow diagrams
Control-flow diagrams
Processing narratives
State diagrams
Sequence diagrams
Requirements :
Constraints
Interoperability
Targets and configuration
Design class realizations
Subsystems
Collaboration diagrams
Technical interface design
Navigation design
GUI design
Component diagrams
Design classes
Activity diagrams
Sequence diagrams
Design class realization
Subsystems
Collaboration
Design model
Refinement to:
Design class realizations
Subsystems
Collaboration
Diagrams
Refinement to:
Component diagrams
Design classes
Activity diagrams
Sequence diagrams
Component diagrams
Design classes
Activity diagrams
Sequence diagrams
Deployment diagrams
Low
Architecture elements
Interface elements
Component-level elements
Deployment-level elements
Process dimension

Standardisation using UML

Why standardize UML diagrams?

- UML, short for Unified Modeling Language, is a standardized modeling language consisting of an integrated set of diagrams, **developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems**, as well as for business modeling and other non-software systems.
- 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.

Structure diagrams show the static structure of the system and its parts on different abstraction and implementation levels and how they are related to each other.

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.

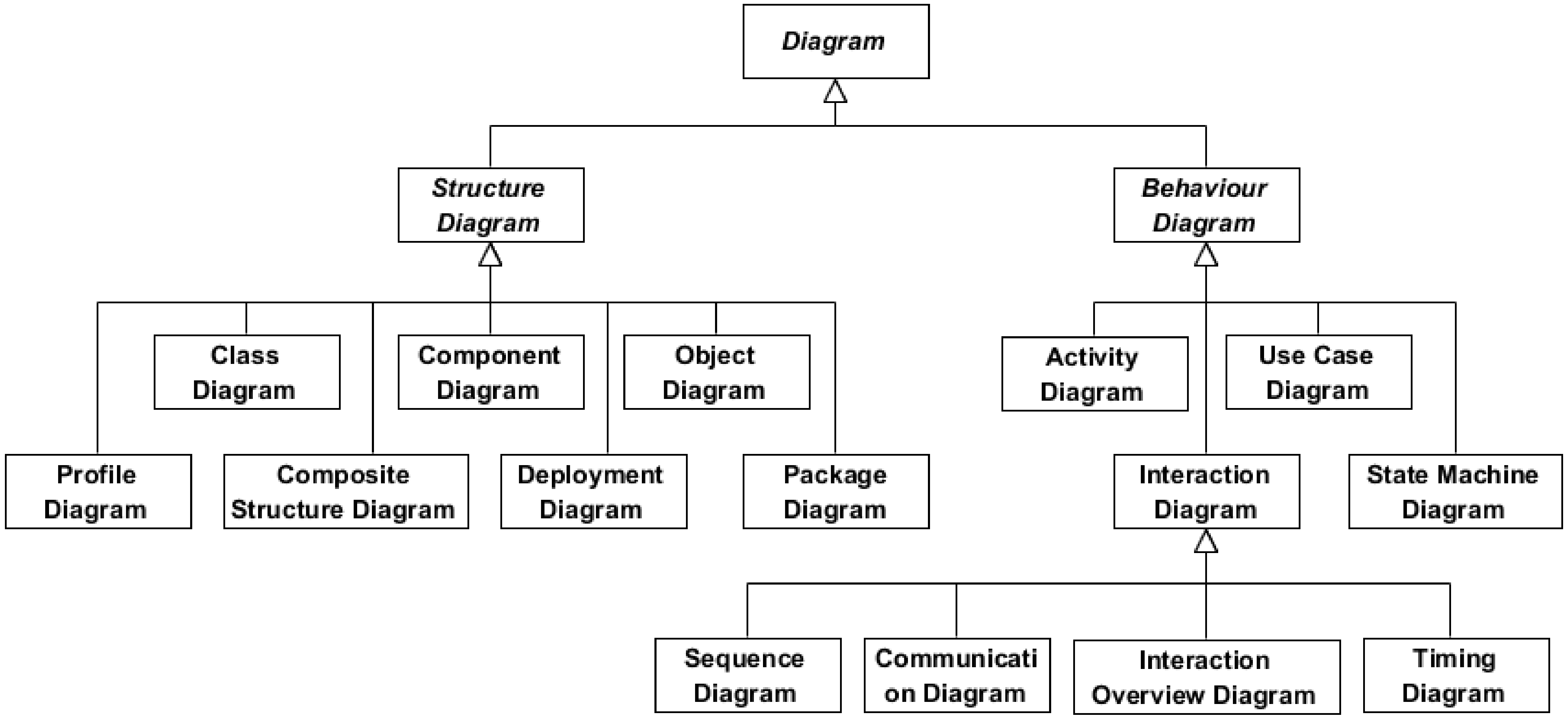


Diagram
Structure Diagram
Behaviour Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
Use Case Diagram
Profile Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Interaction Diagram
State Machine Diagram
Sequence Diagram
Communication Diagram
Interaction Overview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

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.

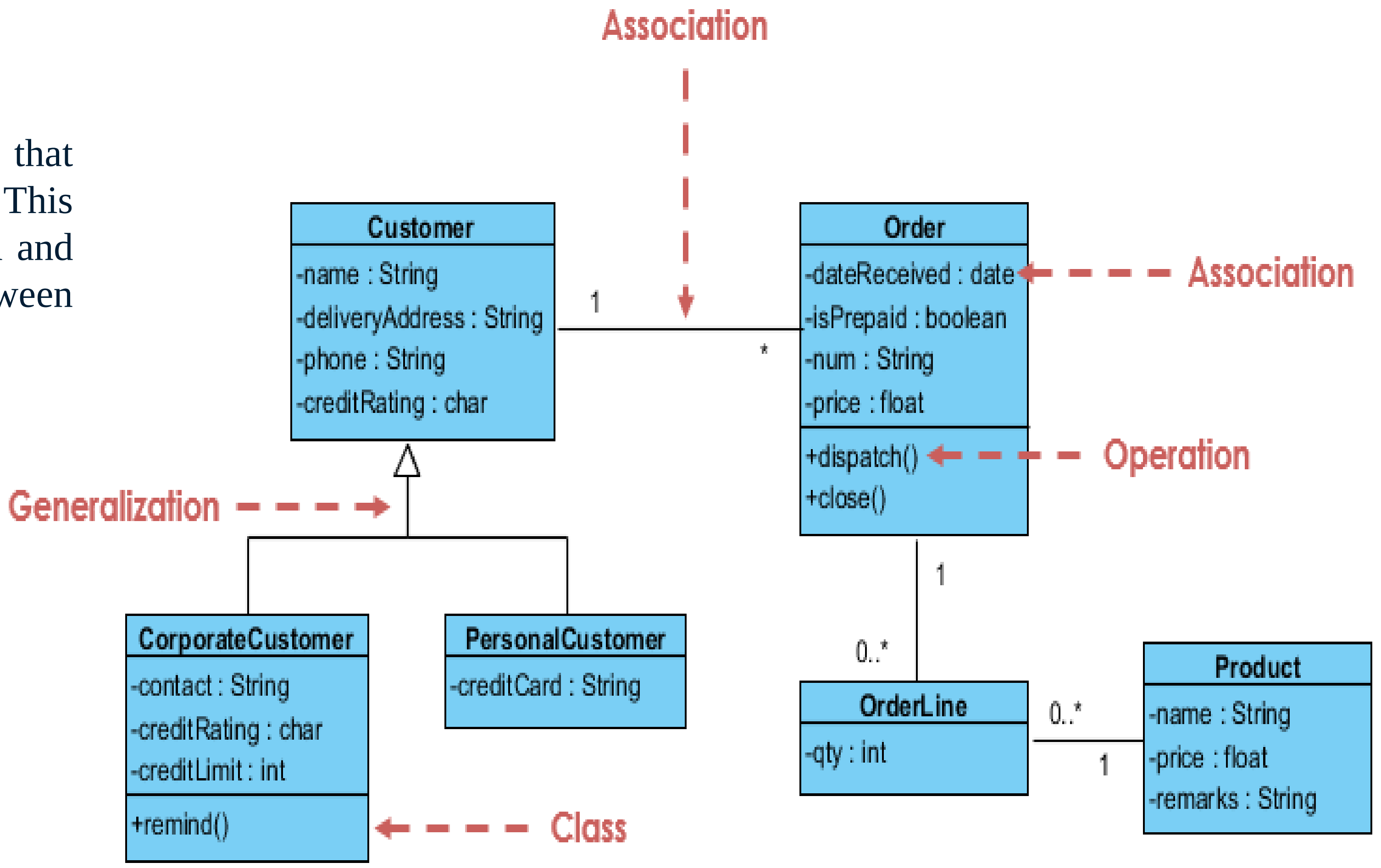
Relationships

There are three principal kinds of relationships which are important:

1.Association - represent relationships between instances of types (a person works for a company, a company has a number of offices.

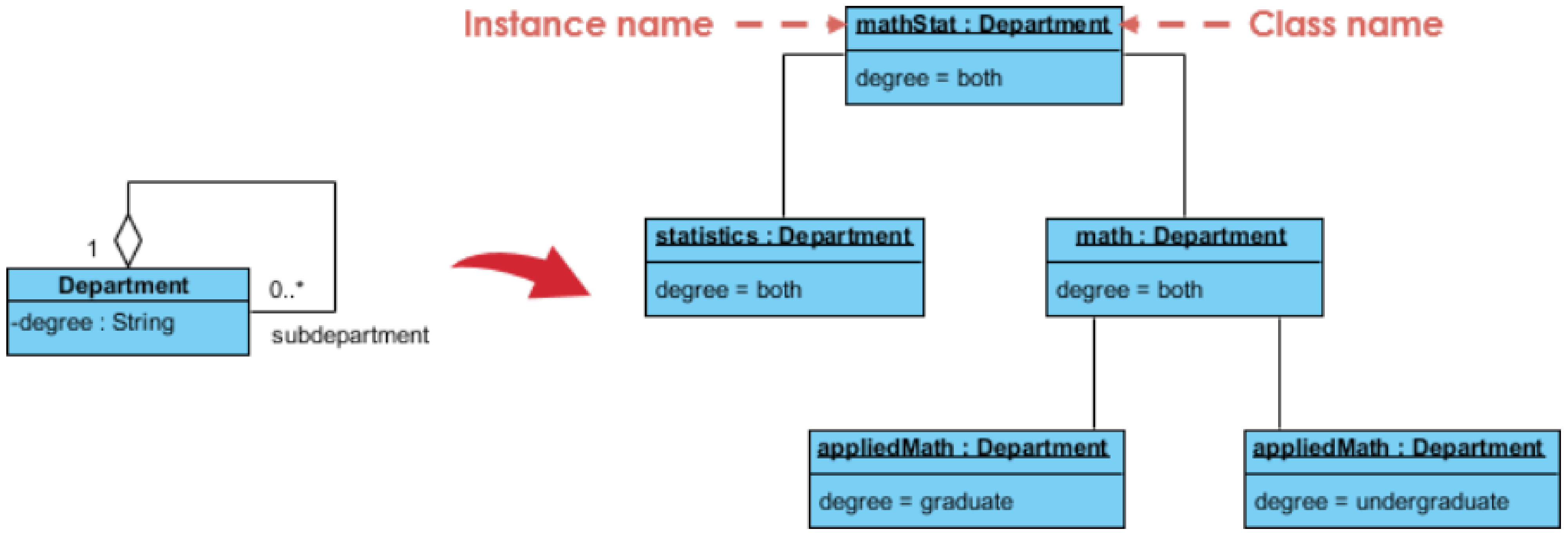
2.Inheritance - the most obvious addition to ER diagrams for use in OO. It has an immediate correspondence to inheritance in OO design.

3.Aggregation - Aggregation, a form of object composition in object-oriented design.



Object Diagram

- Object is an instance of a class in a particular moment in runtime that can have its own state and data values. Likewise a static UML 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, thus an object diagram encompasses objects and their relationships which may be considered a special case of a class diagram or a communication diagram.

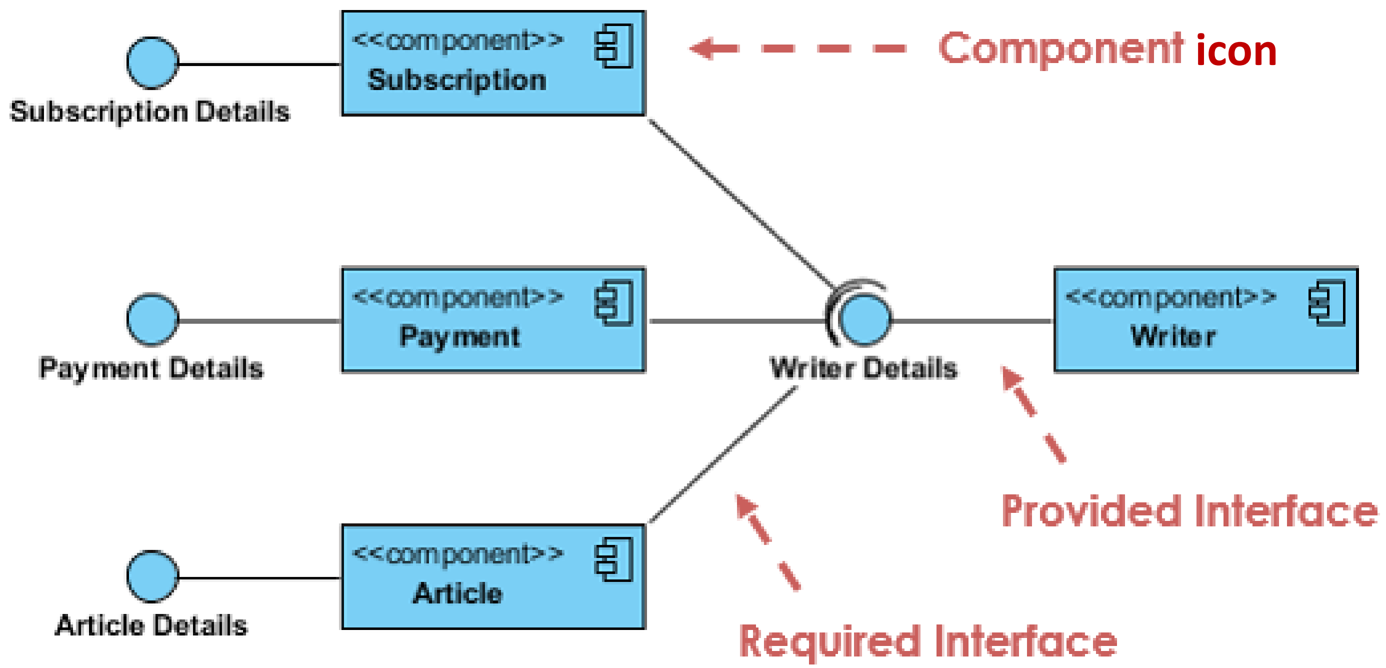


Component Diagram

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.

Component shapes must have the <<component>> stereotype label and/or the component icon in the rectangular shape. A blank rectangle shape with out a component specifier is interpreted as a class element.

It represents relationship and visualize physical aspects in our software eg. Libraries, database. Executables, files, documents



Deployment Diagram

•**Artifact:** A product developed by the software, symbolized by a rectangle with the name and the word “artifact” enclosed by double arrows.

•**Association:** A line that indicates a message or other type of communication between nodes.

•**Component:** A rectangle with two tabs that indicates a software element.

•**Dependency**: A dashed line that ends in an arrow, which indicates that one node or component is dependent on another.

•**Interface**: A circle that indicates a contractual relationship. Those objects that realize the interface must complete some sort of obligation.

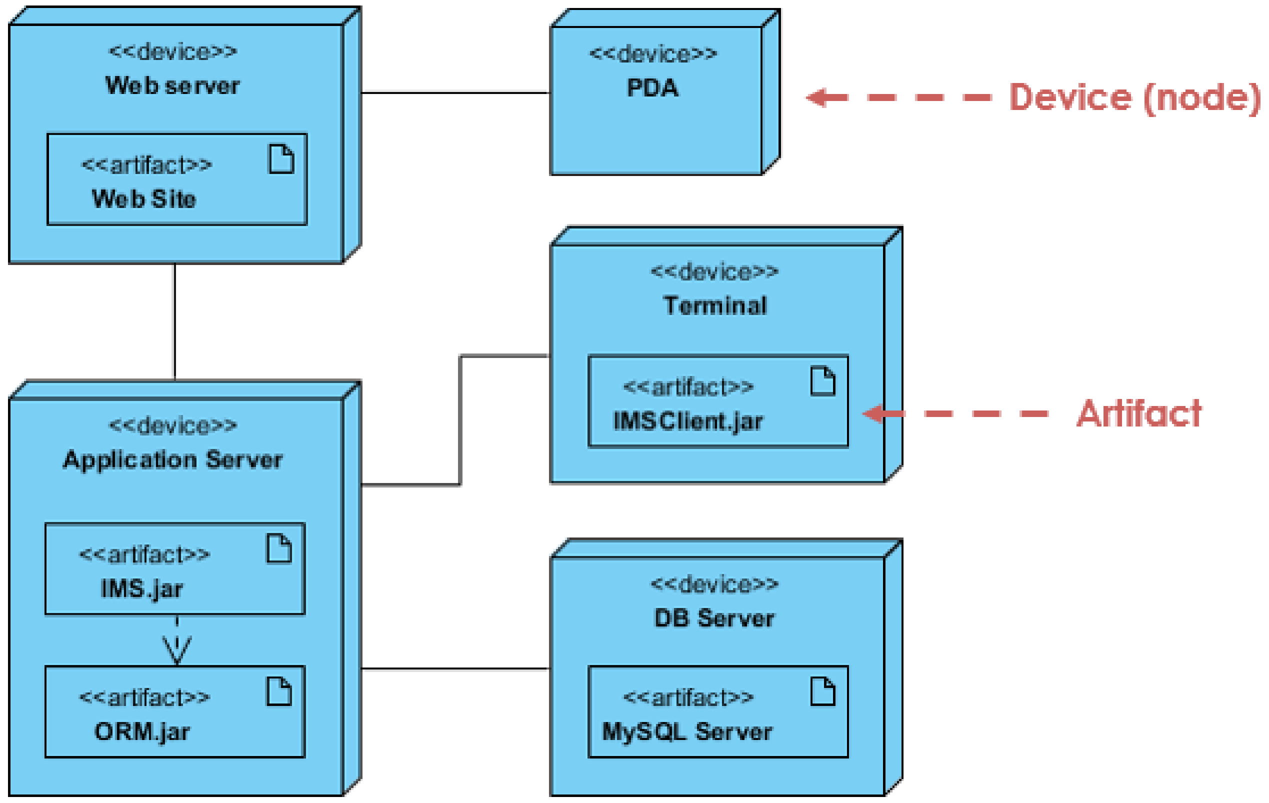
•**Node**: A hardware or software object, shown by a three-dimensional box.

•**Node as container:** A node that contains another node inside of it—such as in the example below, where the nodes contain components.

•**Stereotype:** A device contained within the node, presented at the top of the node, with the name bracketed by double arrows.

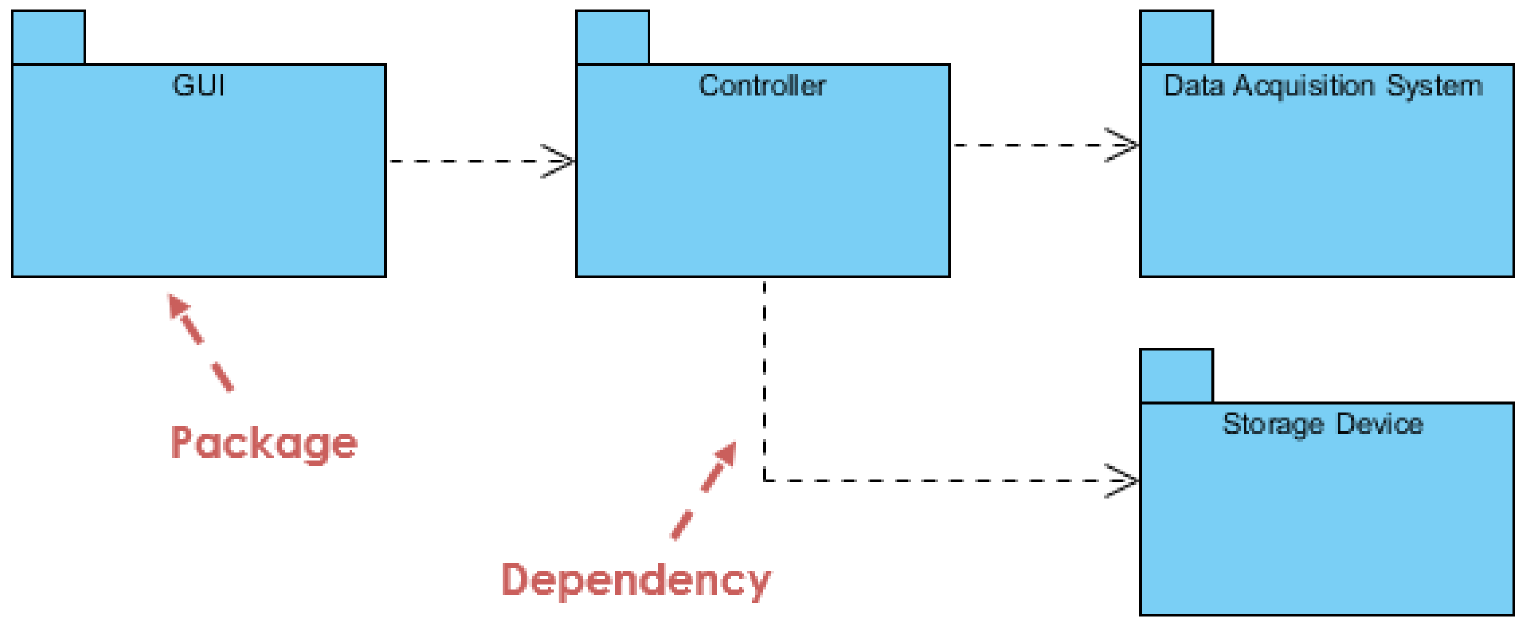
Deployment diagrams have several valuable applications.

- Show which software elements are deployed by which hardware elements.
- Illustrate the runtime processing for hardware.
- Provide a view of the hardware system’s topology.



Package Diagram

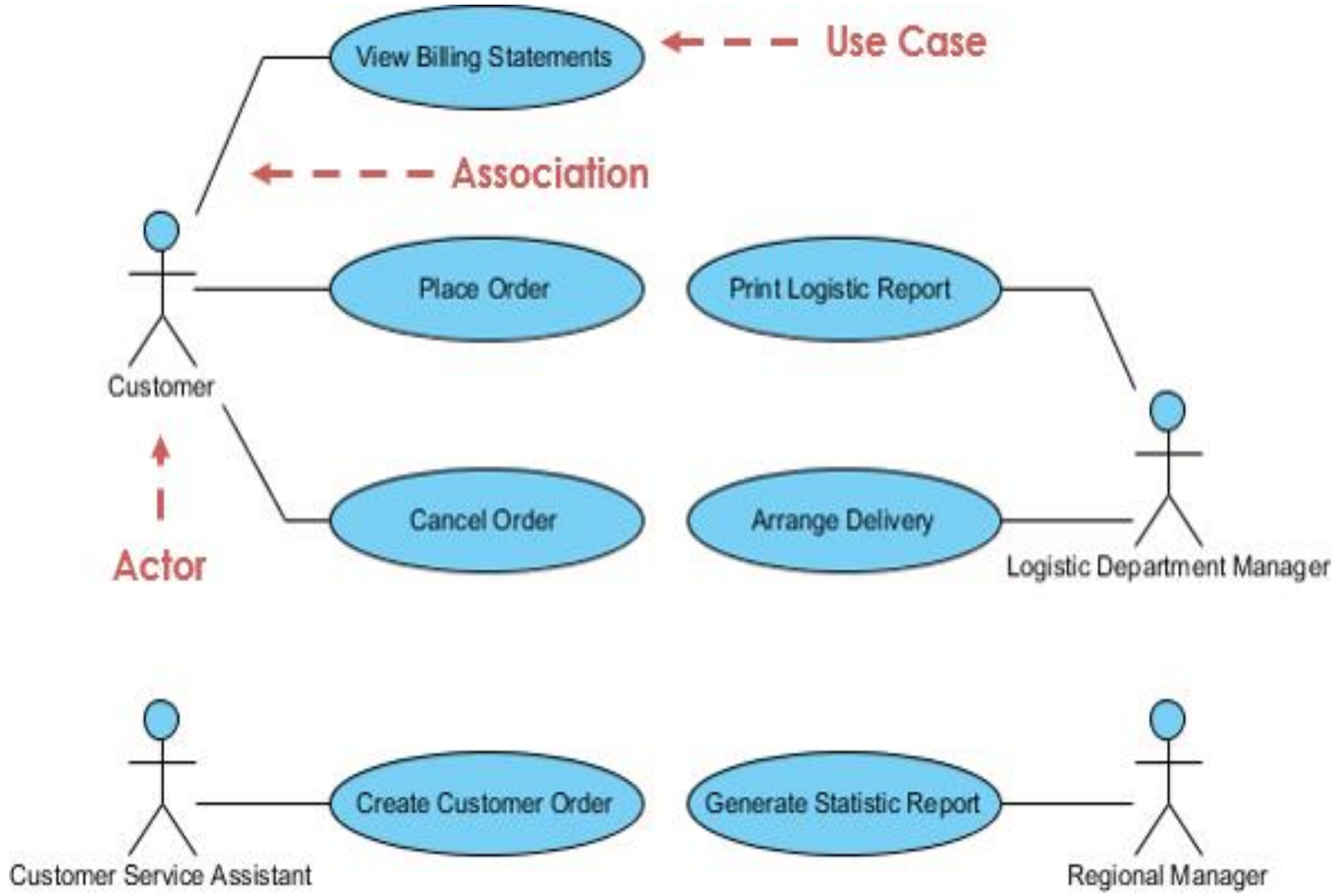
- Package diagram is UML structure diagram which shows packages and dependencies between the packages. Model diagrams allow to show different views of a system, for example, as multi-layered (aka multi-tiered) application - multi-layered application model.

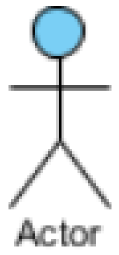


Use Case Diagram

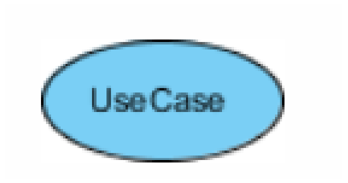
A use case diagram is usually simple.

- It only summarizes **some of the relationships** between use cases, actors, and systems.
- It does **not show the order** in which steps are performed to achieve the goals of each use case.
- Use cases represent only the functional requirements of a system.

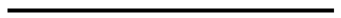




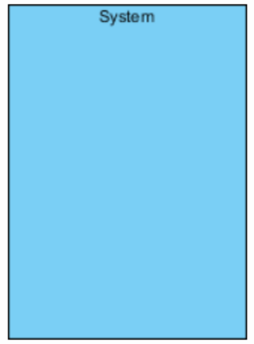
Actor: Someone interacts with use case (system function).



Use Case: System function (process - automated or manual)

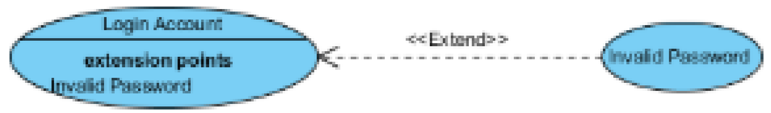


Communication Link: The participation of an actor in a use case is shown by connecting an actor to a use case by a solid link.

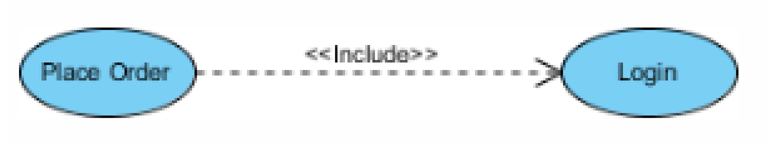


Boundary of system: The system boundary is potentially the entire system as defined in the requirements document.

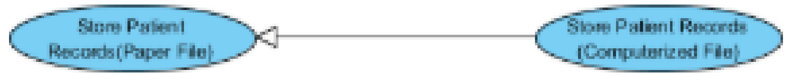
Use Case Relationship



•Depict with a directed arrow having a dotted line. The tip of arrowhead points to the base use case and the child use case is connected at the base of the arrow.
•The stereotype "<<extends>>" identifies as an extend relationship



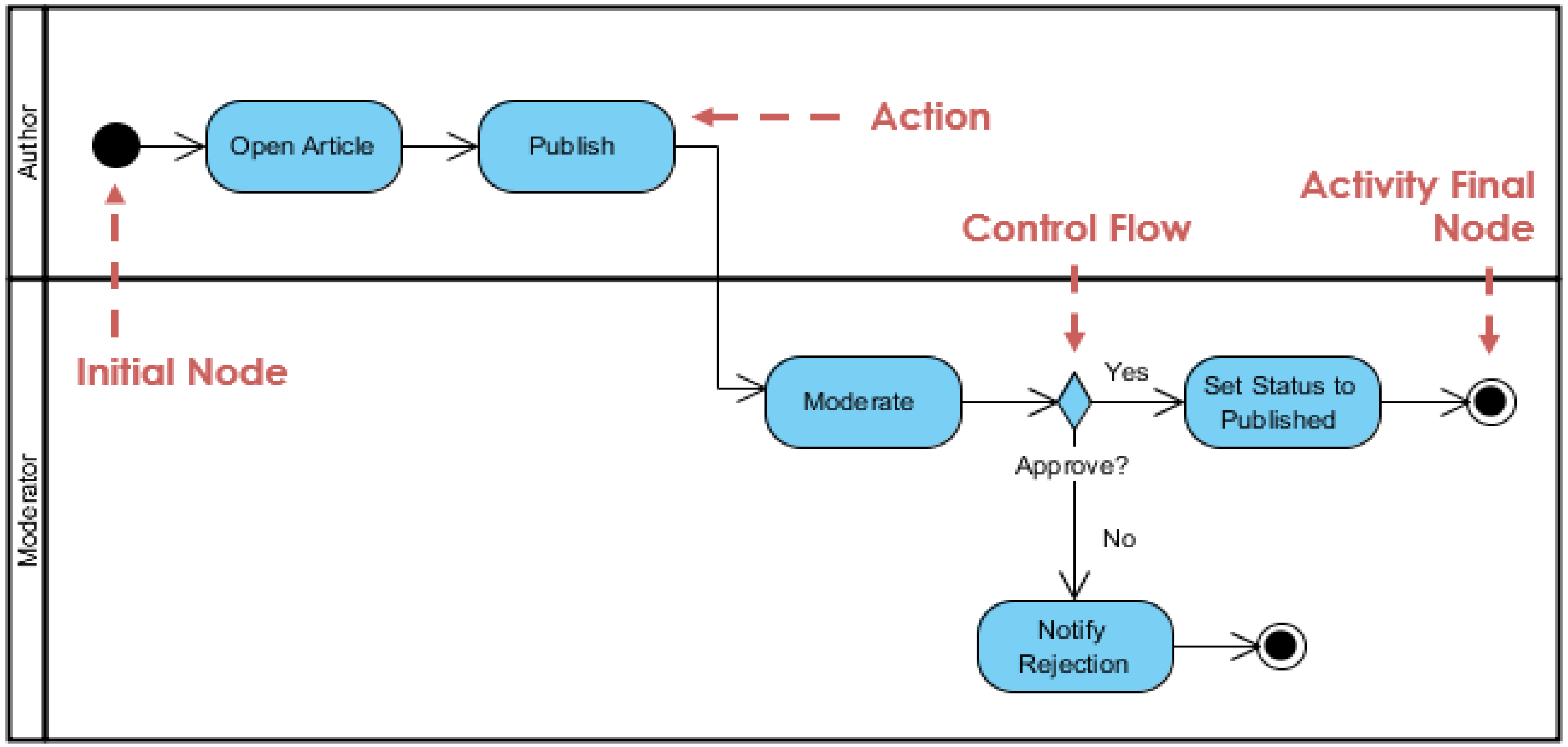
•When a use case is depicted as using the functionality of another use case, the relationship between the use cases is named as include or uses relationship.


Store Patient Records(Paper File)
Store Patient Records (Computerized File)

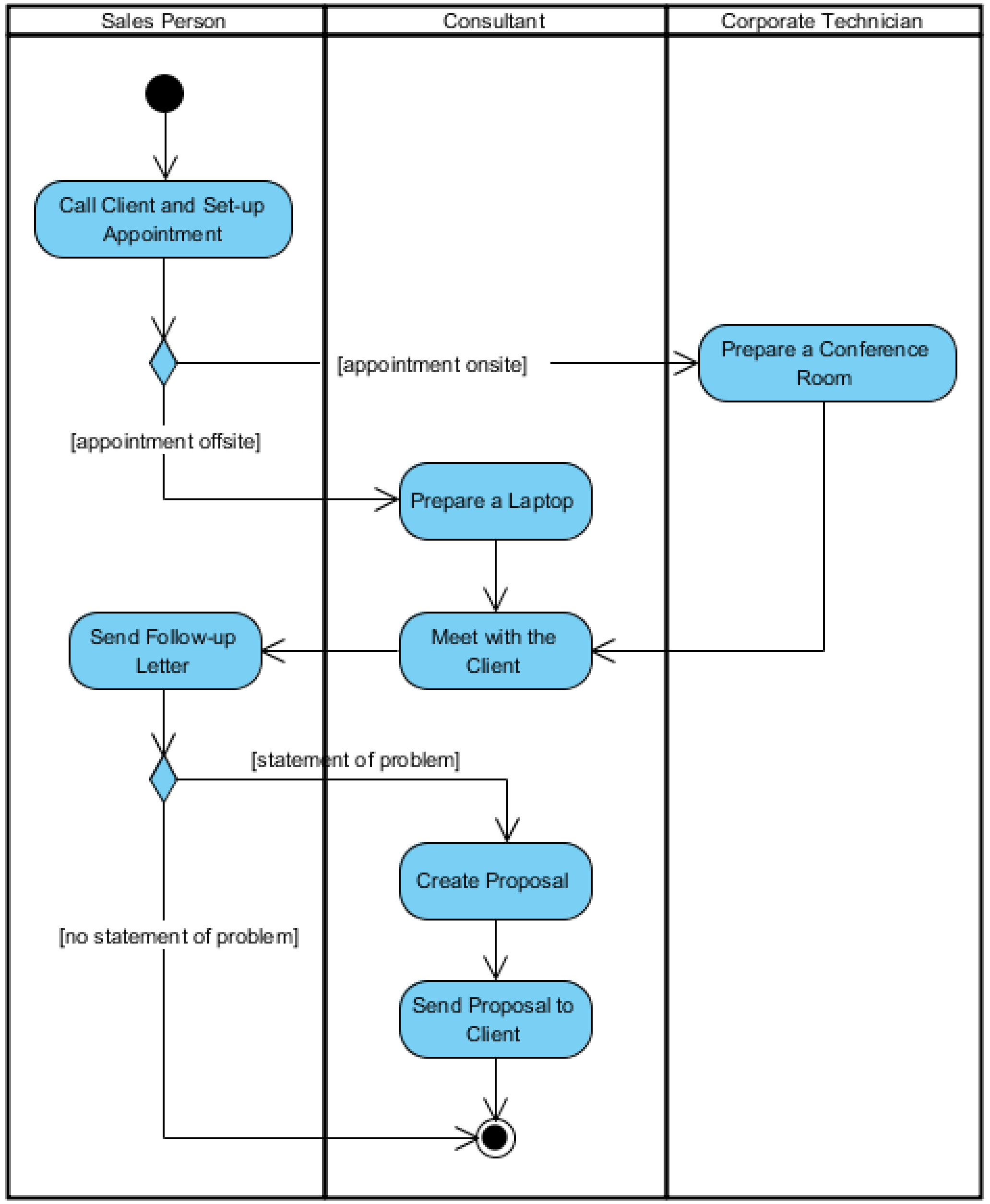
•A generalization relationship is a parent-child relationship between use cases.
•The child use case is an enhancement of the parent use case.
•Generalization is shown as a directed arrow with a triangle arrowhead.
•The child use case is connected at the base of the arrow. The tip of the arrow is connected to the parent use case.

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.

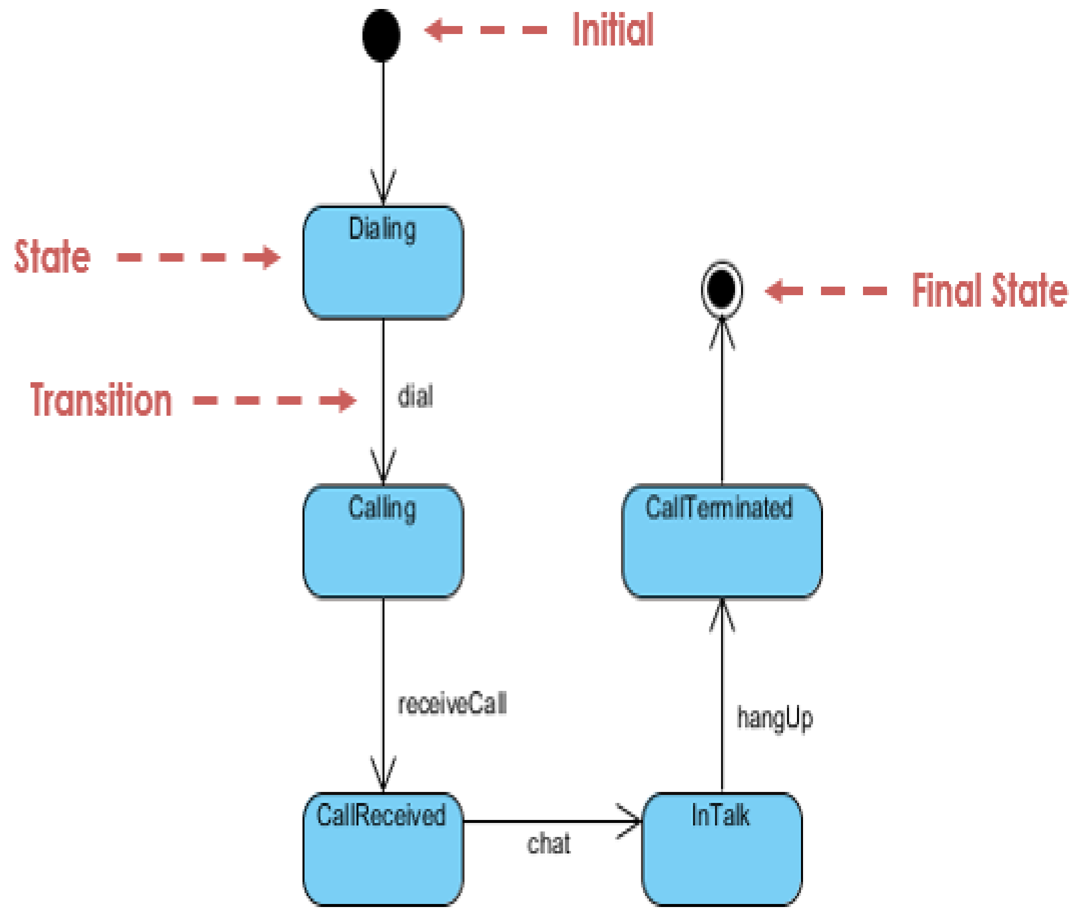


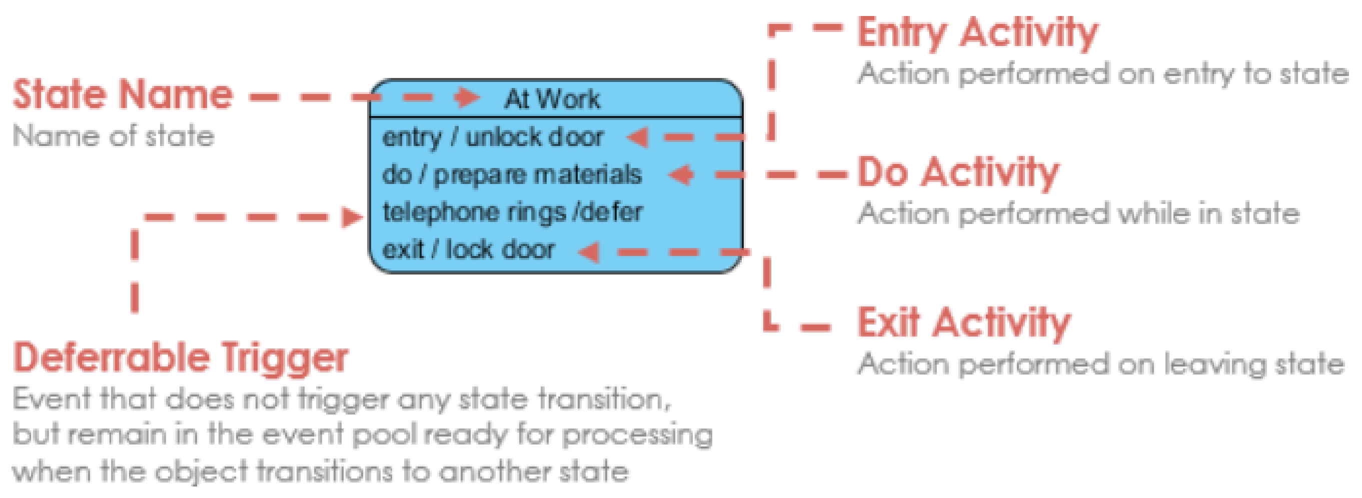
This figure describes the business process for meeting a new client using an activity Diagram with Swimlane.



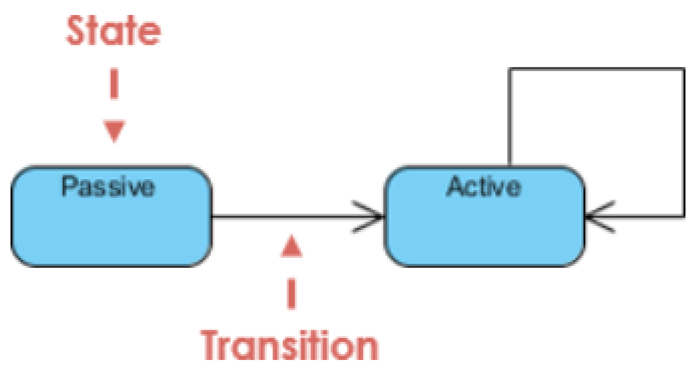
State Machine Diagram

- UML State Machine Diagrams (or sometimes referred to as state diagram, state machine or state chart) show the different states of an entity.
- State machine diagrams can also show how an entity responds to various events by changing from one state to another.
- State machine diagram is a UML diagram used to model the dynamic nature of a system.

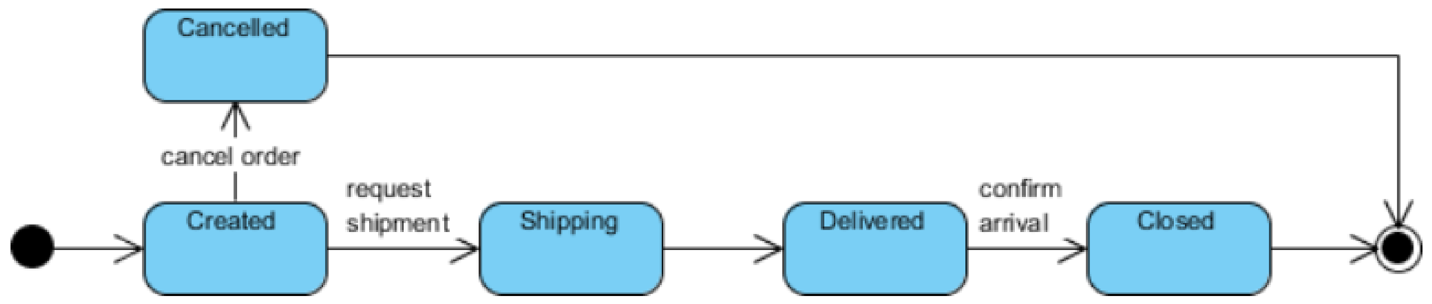



State Name
Name of state
At Work
entry / unlock door
do / prepare materials
telephone rings /defer
exit / lock door
Entry Activity
Action performed on entry to state
Do Activity
Action performed while in state
Exit Activity
Action performed on leaving state
Deferrable Trigger
Event that does not trigger any state transition, but remain in the event pool ready for processing when the object transitions to another state

State Notation



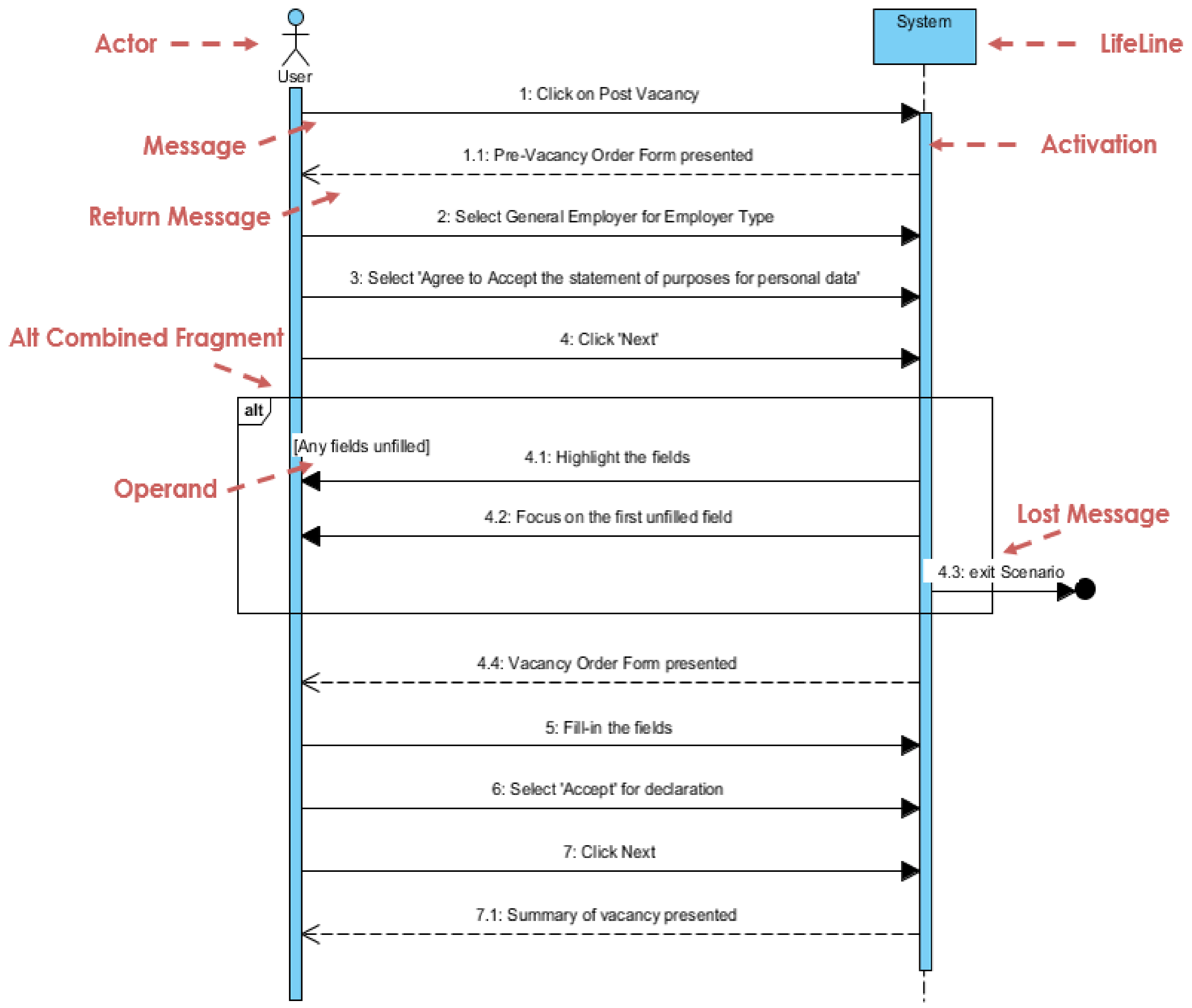
State transitions connecting the states



Initial and Final States

Sequence Diagram

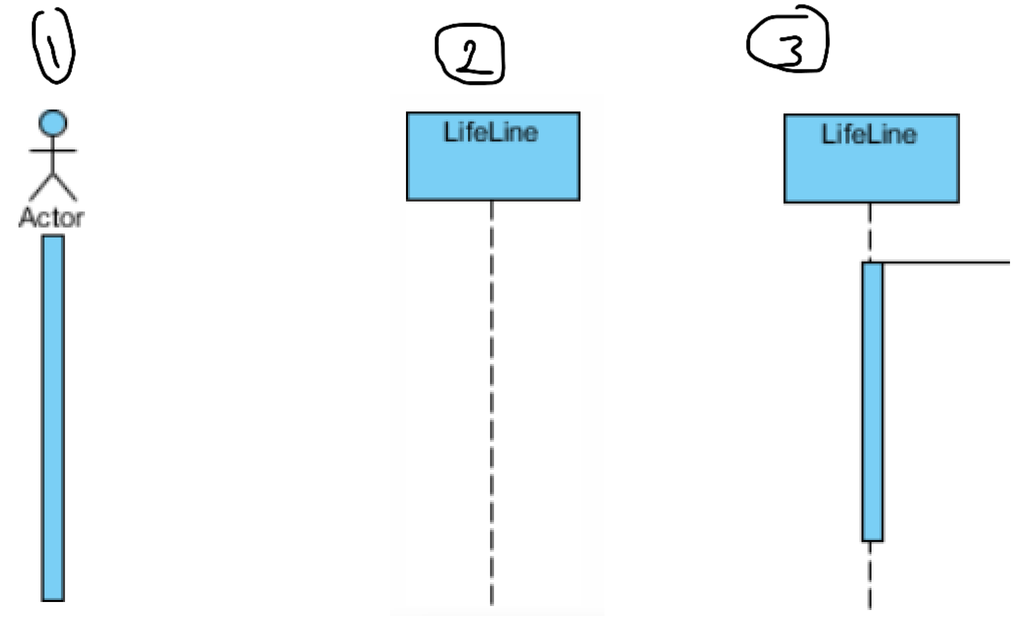
- The Sequence Diagram models the collaboration of objects based on a time sequence.
- It shows how the objects interact with others in a particular scenario of a use case.
- The horizontal axis shows the elements that are involved in the interaction
- The vertical axis represents time proceedings (or progressing) down the page.


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

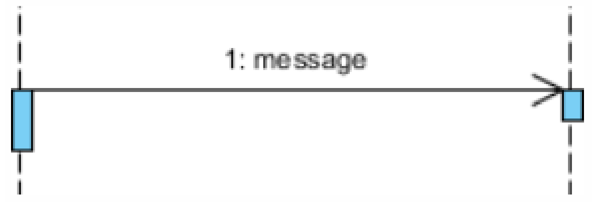
Actor: Represent roles played by human users, external hardware, or other subjects.

Lifeline: A lifeline represents an individual participant in the Interaction.

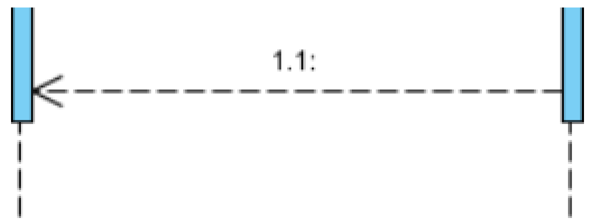
Activations: A thin rectangle on a lifeline) represents the period during which an element is performing an operation.



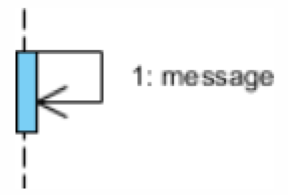
Call Message: A message defines a particular communication between Lifelines of an Interaction.



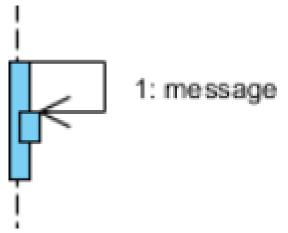
Return Message: Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message.



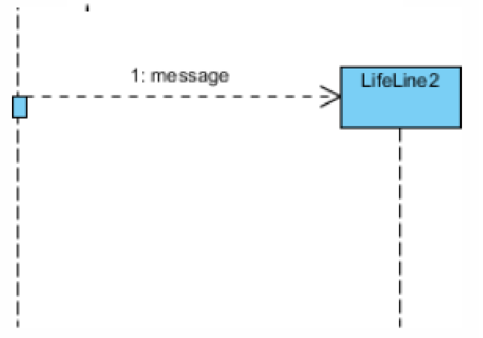
Self Message: Self message is a kind of message that represents the invocation of message of the same lifeline.



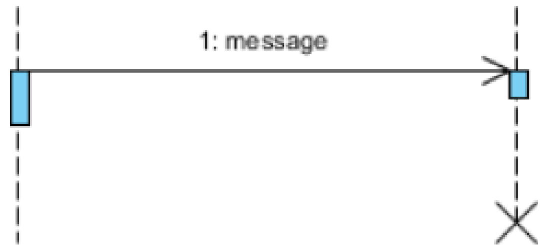
Recursive Message: Recursive message is a kind of message that represents the invocation of message of the same lifeline.



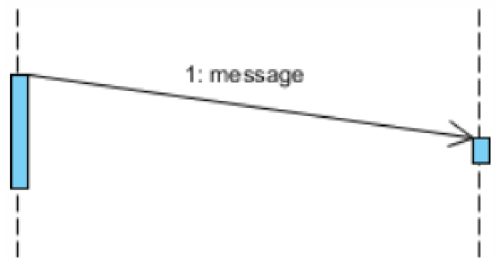
Create Message: Create message is a kind of message that represents the instantiation of (target) lifeline.



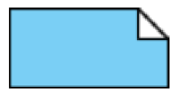
Destroy Message: Destroy message is a kind of message that represents the request of destroying the lifecycle of target lifeline.



Duration Message: Duration message shows the distance between two time instants for a message invocation.

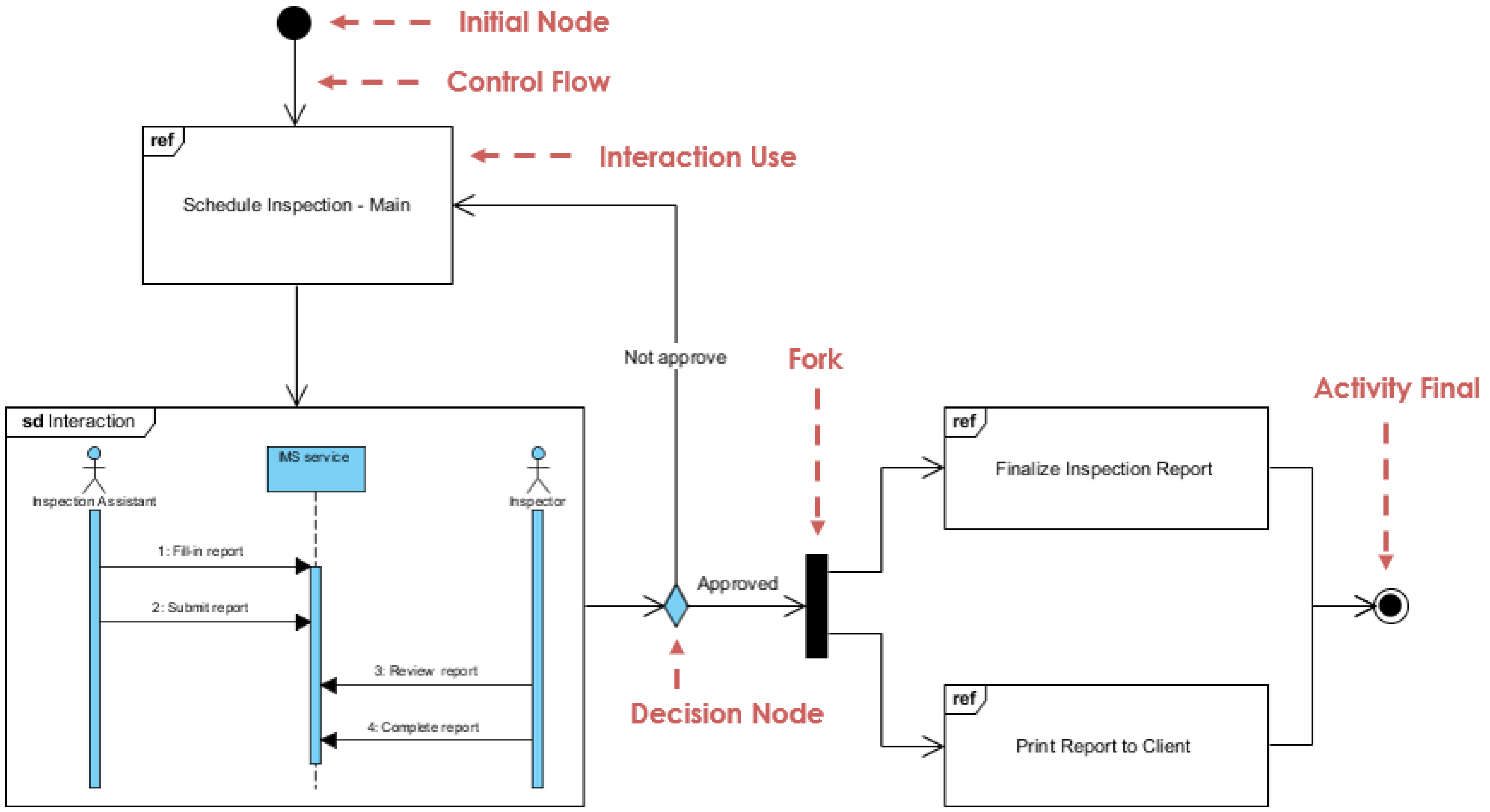


Note: A note (comment) gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler.

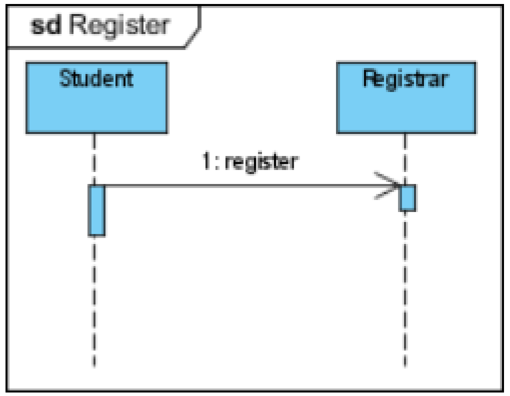


Interaction Diagram

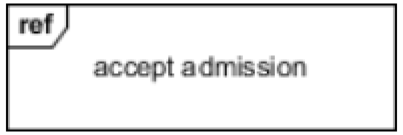
It is a variant of the Activity Diagram where the nodes are the interactions or interaction occurrences. The Interaction Overview Diagram focuses on the overview of the flow of control of the interactions which can also show the flow of activity between diagrams.



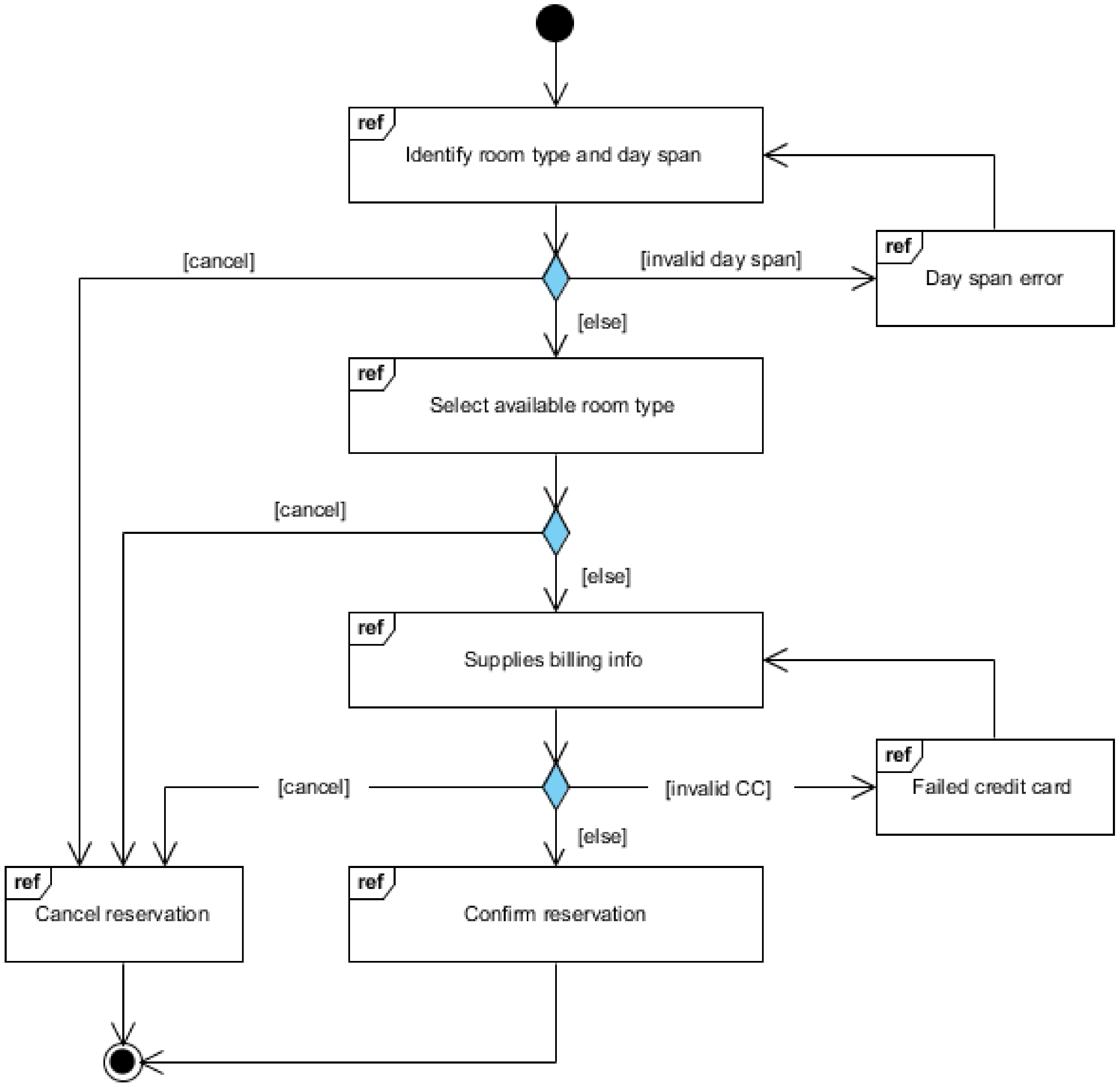
Notation



Interaction
An Interaction diagram of any kind may appear inline as an Activity Invocation.

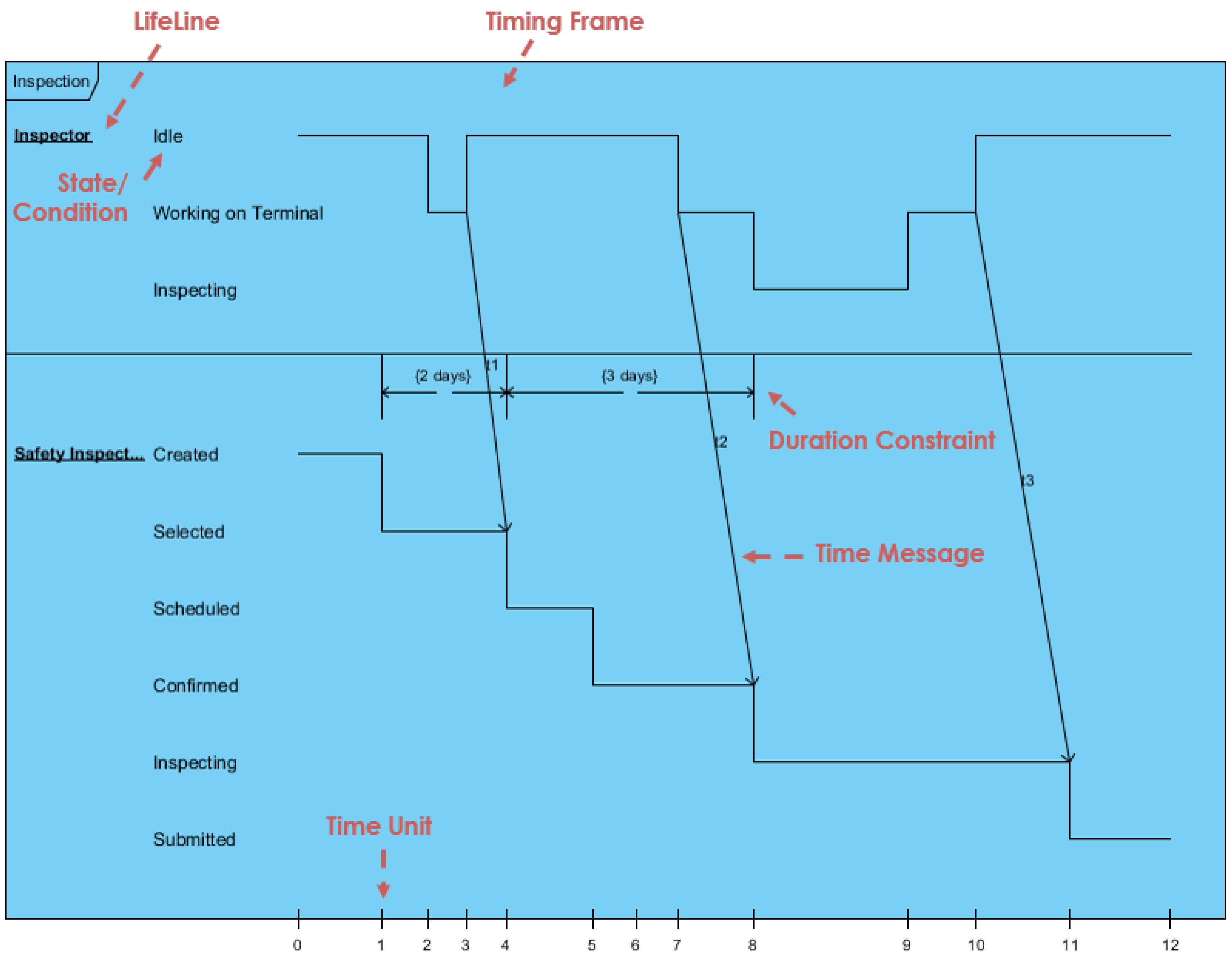


Interaction Use
Large and complex sequence diagrams could be simplified with interaction uses. It is also common to reuse some interaction between several other interactions.



Timing Diagram

- Timing Diagram shows the behavior of the object(s) in a given period of time.
- Timing diagram is a special form of a sequence diagram.
- The differences between timing diagram and sequence diagram are the axes are reversed so that the time increase from left to right and the lifelines are shown in separate compartments arranged vertically.


LifeLine
Timing Frame
Inspection
Inspector
Idle
State/
Condition
Working on Terminal
Inspecting
{2 days}
t1
{3 days}
Duration Constraint
t2
t3
Safety Inspect...
Created
Selected
Time Message
Scheduled
Confirmed
Inspecting
Submitted
Time Unit
0
1
2
3
4
5
6
7
8
9
10
11
12