UNIT-2

ELABORATION PHASE AND INTERACTIVE DIAGRAMS

UNIT II: Elaboration Phase And Interaction Diagram

Definition of the Elaboration phase, identify system events and system operations, Creating a System Sequence Diagram, Interaction diagrams: Importance of interaction diagrams-Sequence diagrams. Creating a collaboration diagram.

Definition of Elaboration Phase

In software engineering, the elaboration phase is one of the stages in the Rational Unified Process (RUP) and some other software development methodologies, such as the Unified Process and Agile approaches.

It is typically the second phase in these methodologies, following the inception phase, and it focuses on refining the requirements gathered during the inception phase to provide a more detailed understanding of the system to be developed.

The main objectives of the elaboration phase are as follows:

- **Requirements Refinement:** During the inception phase, the project's vision and requirements are identified at a high level. In the elaboration phase, these requirements are further analyzed and refined to ensure they are well-defined, complete, and unambiguous.

- **Architectural Planning:** The elaboration phase is crucial for establishing the system's architecture. It involves defining the overall structure of the software, identifying key components, modules, and their interactions. The architectural design must address various quality attributes such as performance, scalability, maintainability, and security.

- **Risk Identification and Mitigation:** Potential risks associated with the project are identified during the elaboration phase. These risks may be related to technology, complexity, resources, etc. Once identified, appropriate strategies are formulated to mitigate these risks.

- **Prototyping:** In some cases, the elaboration phase may involve creating prototypes of critical parts of the system. These prototypes help in validating design decisions, uncovering potential issues early, and gaining a better understanding of the system's behavior.

- **Estimation and Planning:** The elaboration phase contributes to more accurate project planning and scheduling. As the requirements and architecture become clearer, the development team can provide more detailed estimates for the work involved.

- **Baseline for Iterative Development:** Many software development methodologies, such as Agile and RUP, emphasize iterative development. The elaboration phase sets the initial baseline for subsequent iterations or increments of the software.

- **Initial User Involvement:** During the elaboration phase, user involvement is essential for gathering feedback on the refined requirements and the proposed architecture. This feedback helps in ensuring that the software aligns with user needs.

By the end of the elaboration phase, **the software development team should have a well-defined, feasible plan for the project, a detailed and validated set of requirements, a clear architecture to guide development, and a strategy to manage identified risks**. The project is then ready to move into the construction phase, where the actual development work begins.

System events and System operations

In software engineering, 1. system events and 2. system operations are fundamental concepts related to the behavior and functioning of a software system.

These concepts play a crucial role in understanding and modeling how a software system interacts with its environment and performs its intended tasks.

1. **System Events:**

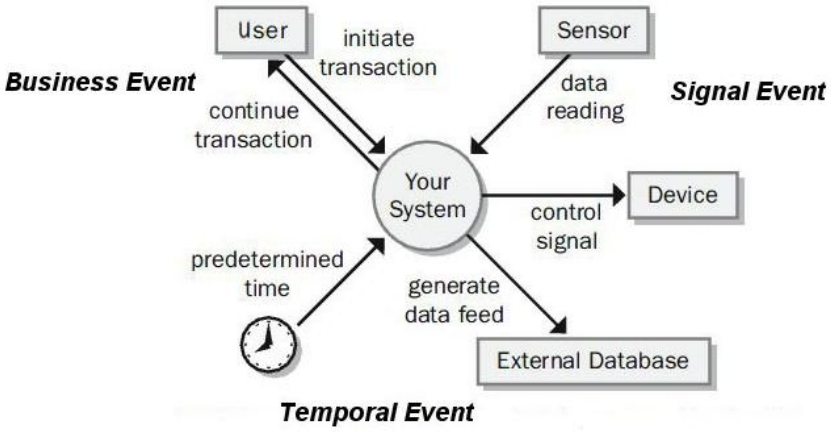
System events are occurrences or incidents that take place within or around the software system, which trigger the execution of certain actions or behavior.

Events can be internal or external to the system and can represent various types of stimuli that the system needs to respond to.

Examples of system events include user interactions, hardware or software interrupts, time-based triggers, communication events, and changes in the system's state.

In event-driven systems, the software responds to these events by executing specific event handlers or event-driven routines. Events can be asynchronous, meaning they happen independently of the software's main flow, and the system needs to be designed to handle them appropriately.

Event handling is commonly used in graphical user interfaces (GUIs), real-time systems, and event-based architectures.



2. **System Operations:**

System operations, also known as **functions or methods**, are the actions or tasks that a software system can perform to achieve specific functionalities or goals. These operations represent the behavior and capabilities of the system.

They encapsulate a set of instructions and algorithms that process data, manipulate state, or produce specific outcomes based on input parameters.

System operations are defined as part of the software's design and are essential for implementing the system's requirements. They can be invoked in response to system events, user interactions, or as part of the system's internal processing. The operations are typically organized into modules, classes, or components based on their related functionality, which promotes modularity and maintainability in the software.

In object-oriented programming, system operations are represented by methods, while in procedural programming, they are implemented as functions. In both cases, they represent the verbs/actions that the software system can perform.

To summarize, system events represent external and internal triggers that lead to the execution of certain actions or behaviors within the software system, while system operations are the functions or methods that define the system's capabilities and functionalities. Together, they form the basis of a software system's behavior and interaction with its environment.

Creating System Sequence Diagram:

Creating a System Sequence Diagram (SSD) is a valuable technique in software engineering for visualizing the interactions between external actors and the system under development. SSDs help to capture the sequence of events and system responses during use case scenarios.

Below is a step-by-step guide on how to create a System Sequence Diagram:

Step 1: Identify Use Cases:

Identify the specific use cases that you want to represent in the SSD. Use cases are descriptions of interactions between actors and the system that lead to a specific outcome.

Step 2: Identify External Actors:

Identify all the external actors (users, other systems, or entities) that interact with the system during the identified use cases.

Step 3: Define System Operations:

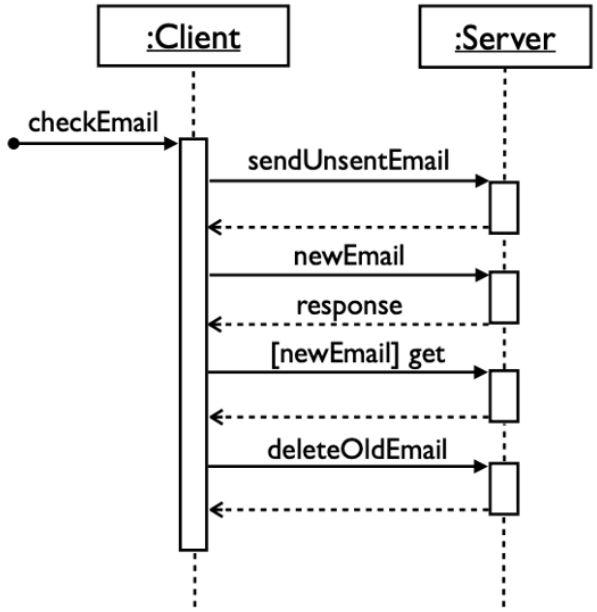
For each use case, determine the system operations (actions or methods) that the system performs in response to the actors' requests. These operations represent the system's behavior.

Step 4: Create the System Sequence Diagram:

Now, let's start creating the SSD for a specific use case:

- Draw a box in the center of the diagram to represent the system boundary. Write the name of the system inside the box.
- Draw vertical lines (lifelines) extending from the system boundary for each external actor involved in the use case. Label these lifelines with the names of the actors.
- Place the use case at the top of the diagram and connect it with the corresponding actor's lifeline using a solid horizontal line. This indicates the initiation of the use case.
- From the use case, draw a vertical dashed line downward, representing the start of the system sequence.
- Below the dashed line, represent the system operations that the system performs in response to the use case. These are represented as messages (labeled with the names of the system operations) from the actor to the system. Optionally, you can include parameters passed in the messages.
- Continue the sequence until the system's response to the actor's request is complete.
- If there are other use cases or alternative paths, create separate diagrams or add additional parts to the existing diagram to represent those interactions.

Step 5: Validate and Review:
Review the SSD with stakeholders, domain experts, and the development team to ensure it accurately represents the intended behavior of the system and the interactions with external actors. Make any necessary adjustments based on feedback. Creating System Sequence Diagrams can help in understanding and communicating the system's interactions and behavior, making it a valuable tool during the analysis and design phases of software development.



Interactive Diagram:

Interactive Diagrams are a valuable tool for visualizing complex systems, architectures, processes, and data interactions. They enable developers, architects, and stakeholders to actively explore and understand the software components, their relationships, and how they interact with each other.

From the term Interaction, it is clear that the diagram is used to describe some type of interactions among the different elements in the model. This interaction is a part of dynamic behavior of the system.

This interactive behavior is represented in UML by two diagrams known **as Sequence diagram and Collaboration diagram**

Interactive diagrams in software engineering can be used for various purposes, including system design, documentation, code analysis, and communication.

Sequence diagram emphasizes on time sequence of messages and **collaboration diagram** emphasizes on the structural organization of the objects that send and receive messages.

Purpose of Interaction Diagrams

The purpose of interaction diagrams is to visualize the interactive behavior of the system. Visualizing the interaction is a difficult task. Hence, the solution is to use different types of models to capture the different aspects of the interaction.

Sequence and collaboration diagrams are used to capture the dynamic nature but from a different angle.

The purpose of interaction diagram is −

- ✓ To capture the dynamic behaviour of a system.
- ✓ To describe the message flow in the system.
- ✓ To describe the structural organization of the objects.
- ✓ To describe the interaction among objects.

Here are some examples of interactive diagrams commonly used in software engineering:

UML Diagrams: Unified Modeling Language (UML) diagrams are widely used in software engineering to model and visualize different aspects of a software system. Interactive UML diagrams allow users to click on elements like **classes, objects, components, and see their attributes, methods, relationships, and associations.**

Interactive Dependency Graphs: Dependency graphs help in understanding the dependencies between software modules, classes, or components. An interactive version allows users to click on nodes to highlight dependencies, explore transitive dependencies, and identify potential code smells.

Sequence Diagrams: Interactive sequence diagrams show the interactions between objects or components over time. Users can click on different steps to view the details of method calls and messages exchanged during the sequence of events.

Interactive Class Hierarchies: For large software systems, class hierarchies can become complex. Interactive class hierarchies enable users to navigate through class inheritance, view method implementations, and understand the overall class structure.

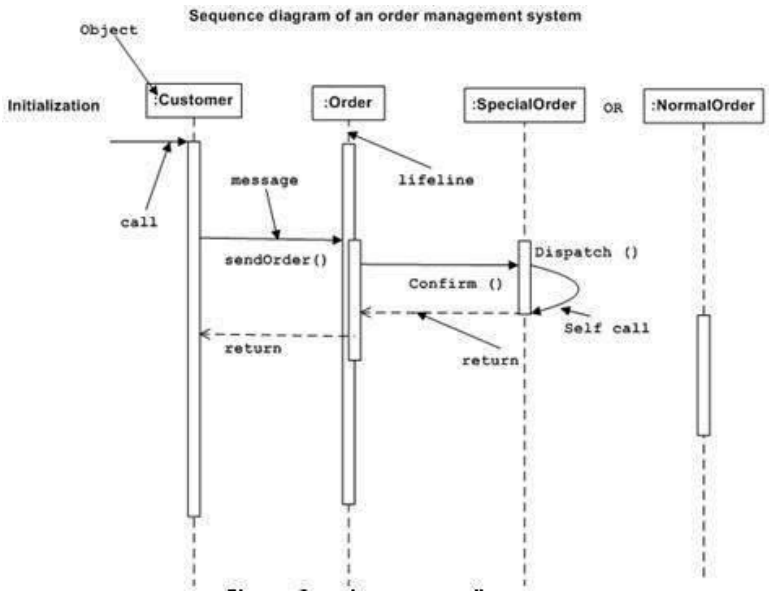
Interactive State Diagrams: State diagrams depict the different states a system or an object can be in and how it transitions between states. An interactive version allows users to explore state transitions and actions associated with each state.

Interactive Data Flow Diagrams: Data flow diagrams illustrate the flow of data through a system or process. Interactivity in these diagrams can allow users to trace data paths, explore data transformations, and identify potential bottlenecks or data-related issues.

Interactive Component Diagrams: Component diagrams represent the physical and logical components of a system and their relationships. Interactivity can help users explore the dependencies between components and the services they offer.

Interactive ER Diagrams: Entity-Relationship (ER) diagrams are used to model the data structure of a database. Interactivity can enable users to view relationships, navigate through entities, and understand the data model.

Creating interactive diagrams in software engineering typically involves using specialized tools or libraries that support interactivity, such as web-based visualization libraries like D3.js or Graphviz. These tools often provide functionalities for handling user interactions, enabling zooming and panning, and linking the diagrams to additional information or documentation.



Importance of Interactive Diagrams:

Interactive diagrams play a crucial role in software engineering due to their numerous benefits and contributions to the software development lifecycle. Here are some of the key reasons why interactive diagrams are important in software engineering:

1. **Visualizing Complexity:** Software systems can be complex, with numerous components, interactions, and dependencies. Interactive diagrams help simplify this complexity by presenting information visually. Developers and stakeholders can interactively explore the architecture, relationships, and data flow, making it easier to understand the system's structure and behavior.

2. **Enhancing Communication:** Interactive diagrams serve as powerful communication tools. They enable developers, architects, and stakeholders to discuss and convey ideas, requirements, and design decisions more effectively. Interactivity allows for real-time exploration, fostering better collaboration and reducing the chances of miscommunication.

3. **Iterative Design and Refinement:** During the software design phase, interactive diagrams allow for iterative refinement. Developers can quickly update and modify the diagrams based on feedback and changing requirements. This agility helps ensure that the design accurately reflects the evolving understanding of the system.

4. **Facilitating Analysis and Troubleshooting:** Interactive diagrams can be used for analyzing codebases and identifying issues. For instance, developers can click on elements to drill down into code details, identify dependencies, and detect potential bottlenecks or design flaws.

5. **Understanding System Behavior:** Interactive diagrams allow developers to simulate and explore the system's behavior in different scenarios. For example, with sequence diagrams, they can step through interactions between objects to understand how the system responds to various inputs.

6. **Enabling User-Focused Design:** Interactive diagrams are beneficial for user-centric design. In user interface design, designers can create wireframes or mockups and allow users to interact with them, gathering feedback on usability and user experience.

7. **Real-Time Data Visualization:** For systems that involve real-time data, interactive diagrams can be connected to live data sources. This provides developers and stakeholders with up-to-date insights into the system's performance and behavior.

8. **Supporting Agile Development:** Interactive diagrams align well with agile methodologies, which emphasize collaboration, iterative development, and rapid feedback. They can be used in sprint planning, retrospectives, and other agile ceremonies to facilitate discussions and decisions.

9. **Learning and Training:** Interactive diagrams are valuable for training and on boarding new team members. They offer an immersive and hands-on learning experience, enabling newcomers to grasp complex concepts and understand the system architecture more effectively.

10. **Documentation and Maintenance:** Interactive diagrams serve as living documentation. They can be embedded in the codebase, wikis, or project documentation, providing an always up-to-date visual representation of the system for future reference and maintenance.

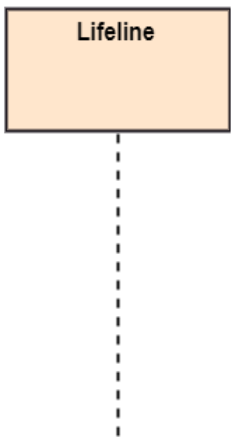
Overall, interactive diagrams bring clarity, engagement, and efficiency to the software engineering process. By promoting better understanding, collaboration, and problem-solving, they significantly contribute to the success of software development projects.

Sequence Diagram:

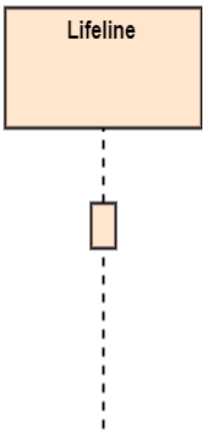
Sequence diagrams are a type of behavioral UML (Unified Modeling Language) diagram used in software engineering to visualize and illustrate the flow of interactions between various objects or components within a system. They showcase the dynamic behavior of a system, representing the sequence of messages exchanged between objects over time. Sequence diagrams are particularly useful for capturing the behavior of a single use case or scenario.

Key Elements of Sequence Diagrams:

1. **Lifeline:** A lifeline represents an individual participant (object or component) in the interaction. It is depicted as a vertical dashed line with the name of the participant written next to it.

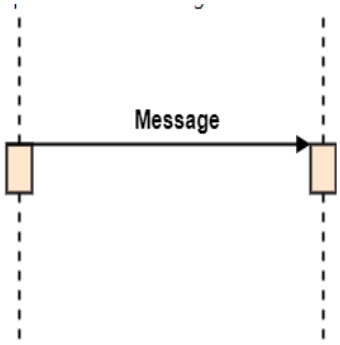


2. **Activation/Execution Occurrence:** An activation or execution occurrence shows when an object is actively involved in the interaction. It is depicted as a narrow rectangle placed on top of the lifeline and indicates the duration of time the object is executing a specific action or method.

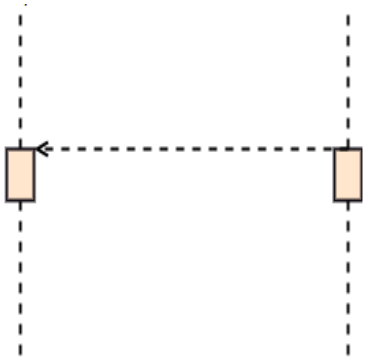


3. **Message:** A message represents a communication between two participants in the sequence diagram. Messages can be 1. synchronous (represented by a solid arrow), 2. asynchronous (represented by a dashed arrow), or 3. create (a message that indicates an object's creation). Messages may also include parameters or return values.

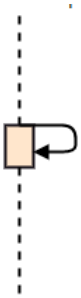
Call Message: It defines a particular communication between the lifelines of an interaction, which represents that the target lifeline has invoked an operation.



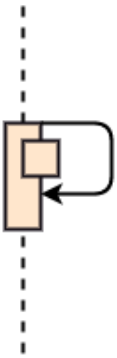
Return Message: A return message represents the response from the called object back to the calling object. It is shown with a dashed arrow and labeled with the value returned, if applicable.



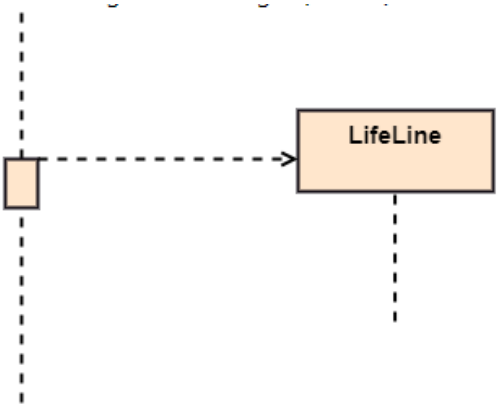
Self-Message: A self-message represents a message that an object sends to itself, indicating an internal action.



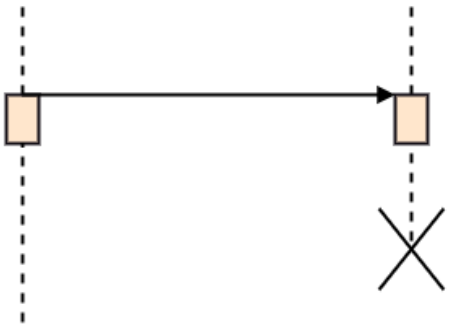
Recursive Message: A self message sent for recursive purpose is called a recursive message. In other words, it can be said that the recursive message is a special case of the self message as it represents the recursive calls.



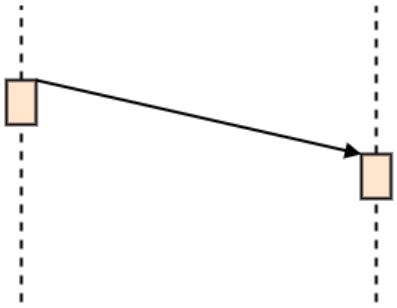
Create Message: It describes a communication, particularly between the lifelines of an interaction describing that the target (lifeline) has been instantiated.



Destroy Message: It describes a communication, particularly between the lifelines of an interaction that depicts a request to destroy the lifecycle of the target.



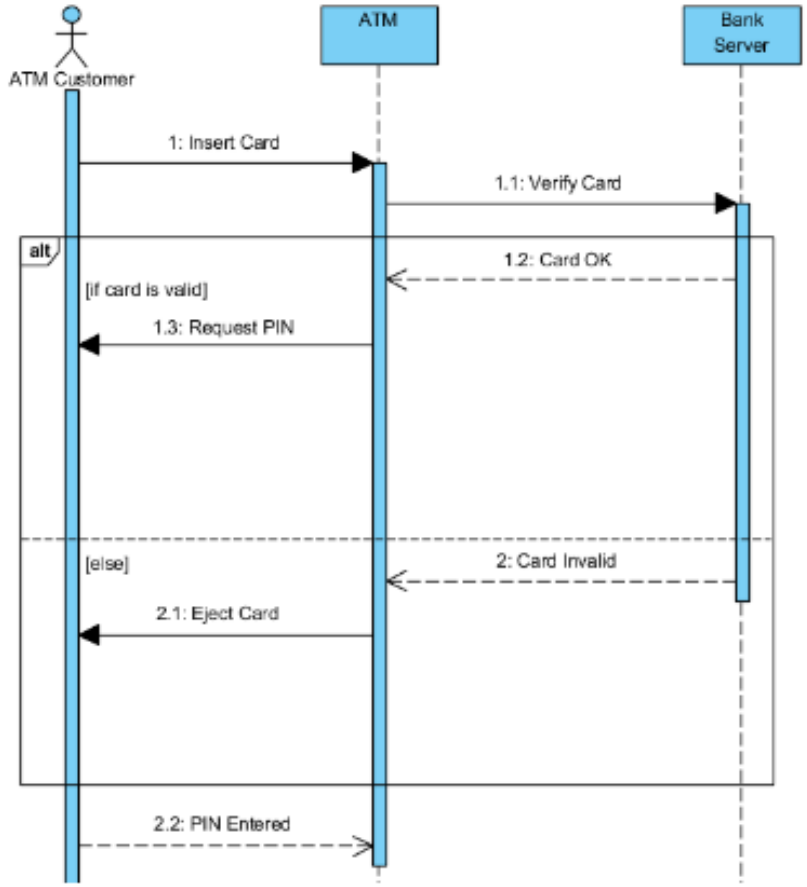
Duration Message: It describes a communication particularly between the lifelines of an interaction, which portrays the time passage of the message while modeling a system.



Advantages of Sequence Diagrams:

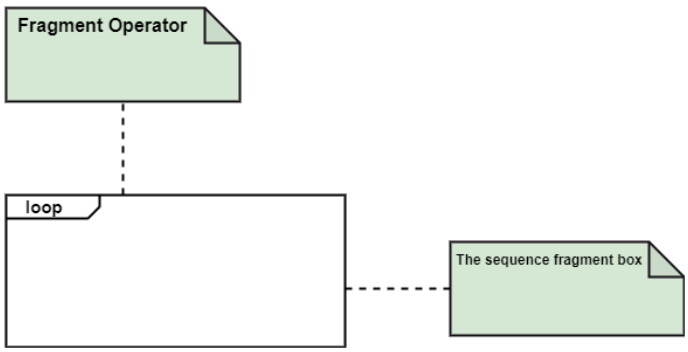
1. **Clarity:** Sequence diagrams provide a clear and visual representation of the interactions between objects, making it easier to understand the behavior of a system during specific use cases.
2. **Communication:** Sequence diagrams serve as an effective communication tool between stakeholders, designers, and developers, allowing them to discuss and refine system behavior.
3. **Testing and Validation:** Sequence diagrams can be used to validate the correctness of a system's behavior and assist in designing test cases for functional testing.
4. **Design and Refactoring:** Sequence diagrams help in designing and refactoring software systems by identifying potential bottlenecks, inefficiencies, or opportunities for improvement.
5. **Documentation:** Sequence diagrams document the dynamic behavior of a system, which can aid in maintaining and extending the software in the future.

Example 1:



Sequence Fragments

1. Sequence fragments have been introduced by UML 2.0, which makes it quite easy for the creation and maintenance of an accurate sequence diagram.
2. It is represented by a box called a combined fragment, encloses a part of interaction inside a sequence diagram.
3. The type of fragment is shown by a fragment operator.

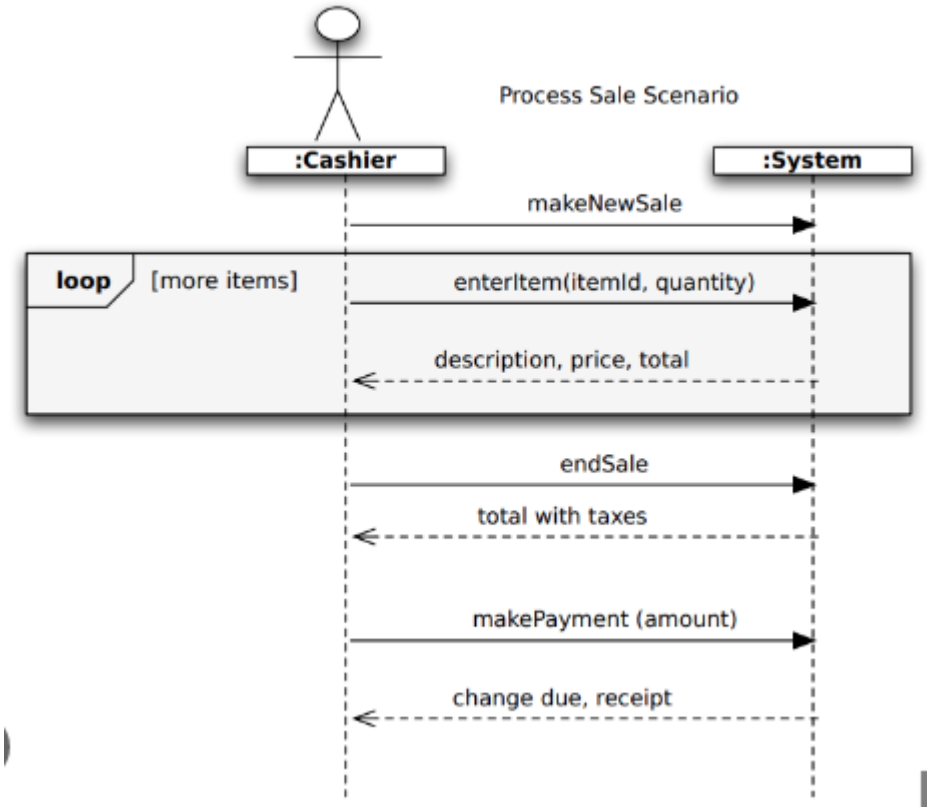


Types of fragments

Following are the types of fragments enlisted below;

Operator	Fragment Type
alt	Alternative multiple fragments: The only fragment for which the condition is true, will execute.
opt	Optional: If the supplied condition is true, only then the fragments will execute. It is similar to alt with only one trace.
par	Parallel: Parallel executes fragments.
loop	Loop: Fragments are run multiple times, and the basis of interaction is shown by the guard.
region	Critical region: Only one thread can execute a fragment at once.
neg	Negative: A worthless communication is shown by the fragment.
ref	Reference: An interaction portrayed in another diagram. In this, a frame is drawn so as to cover the lifelines involved in the communication. The parameter and return value can be explained.
sd	Sequence Diagram: It is used to surround the whole sequence diagram.

Example 2:



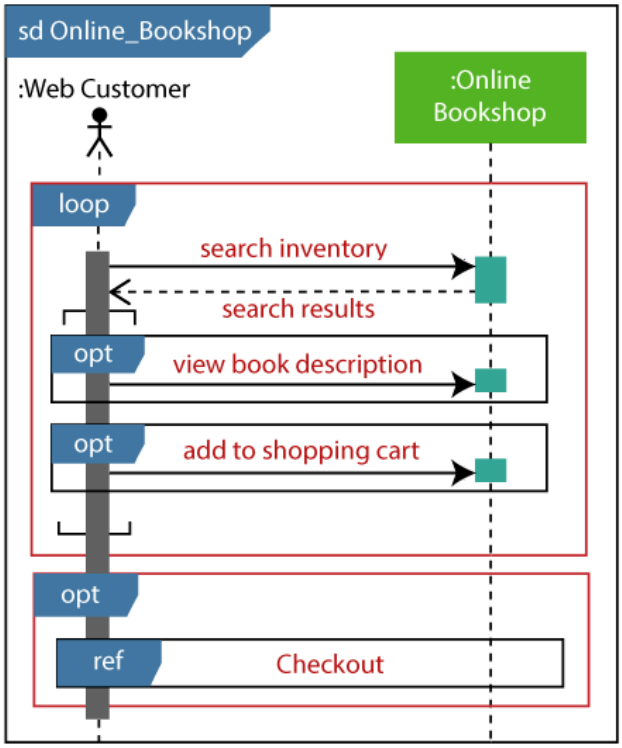
Example 3:

Below is a simple example of a sequence diagram for a basic **online shopping scenario:**

- ✓ Customer->Order: createOrder()
- ✓ Order->Payment: makePayment()
- ✓ Payment->Bank: processPayment()
- ✓ Bank-->Payment: paymentResult(success)
- ✓ Payment-->Order: paymentStatus(success)
- ✓ Order->Warehouse: shipOrder()
- ✓ Warehouse-->Order: orderShipped()

In this example, the customer interacts with the system to create an order. The order is then processed, payment is made, and the order is shipped after successful payment processing.

Overall, sequence diagrams are valuable tools for understanding, designing, and documenting the dynamic behavior of a system, making them a fundamental part of software engineering.



Collaboration Diagrams:

In software engineering, a collaboration diagram, also known as a communication diagram, is a type of UML (Unified Modeling Language) diagram used to visualize and describe the interactions and relationships between objects or components within a system.

Collaboration diagrams focus on the structural organization of objects and the messages exchanged between them during a specific use case or scenario.

To create a collaboration diagram, follow these steps:

Step 1: Identify the Use Case or Scenario:

Choose a specific use case or scenario for which you want to create the collaboration diagram. This will help in determining which objects are involved in the interaction.

Step 2: Identify the Objects:

Identify the objects or components that participate in the interaction. These objects represent the entities within the system that communicate and collaborate to achieve the use case's goals.

Step 3: Create the Diagram:

Now, let's start creating the collaboration diagram:

- Draw a box in the center of the diagram to represent the use case or scenario you are modeling. Write the name of the use case inside the box.
- Create lifelines for each object involved in the interaction. Lifelines are represented as vertical dashed lines extending from the top of the diagram down to the bottom. Label each lifeline with the name of the corresponding object.
- Draw arrows (messages) between the lifelines to represent the messages exchanged between objects. These messages indicate the communication or interaction between objects to achieve the use case's objectives.
- Label the messages with the names of the methods or operations being called and any parameters or return values. Optionally, you can include conditions, loops, or branching constructs to illustrate the flow of the interaction.

Step 4: Refine and Organize:

Organize the elements of the collaboration diagram in a way that clearly shows the flow of interactions between objects. Refine the diagram to ensure it is clear and concise.

Step 5: Review and Validate:

Review the collaboration diagram with stakeholders, domain experts, and the development team to ensure it accurately represents the intended interactions and relationships between objects during the chosen use case or scenario.

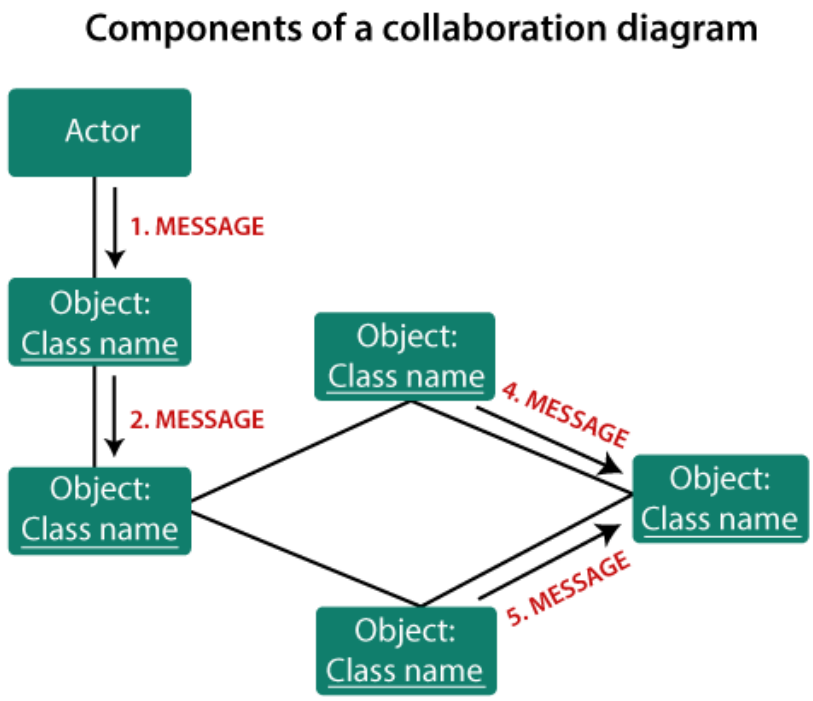
Notations of a Collaboration Diagram

Following are the components of a component diagram that are enlisted below:

1. **Objects:** The representation of an object is done by an object symbol with its name and class underlined, separated by a colon.
   In the collaboration diagram, objects are utilized in the following ways:
   - The object is represented by specifying their name and class.
   - It is not mandatory for every class to appear.
   - A class may constitute more than one object.

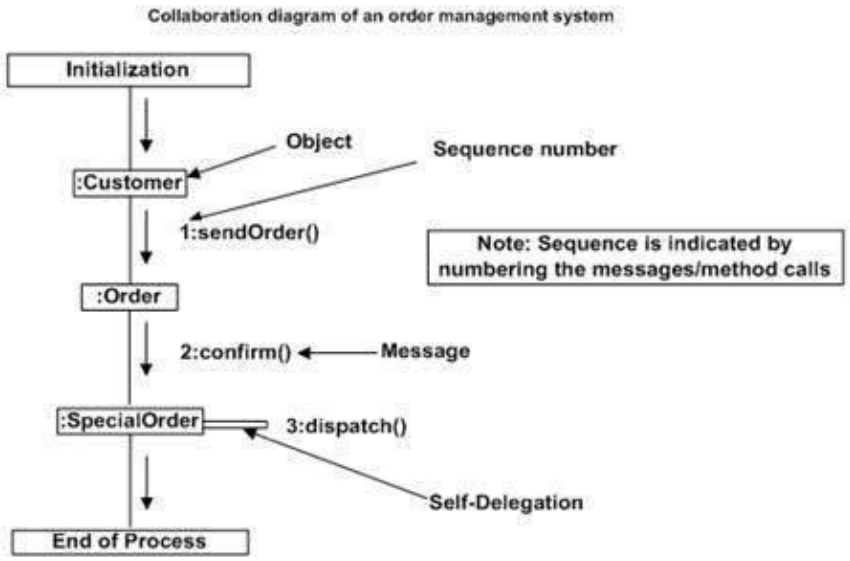
- In the collaboration diagram, firstly, the object is created, and then its class is specified.
- To differentiate one object from another object, it is necessary to name them.

2. **Actors:** In the collaboration diagram, the actor plays the main role as it invokes the interaction. Each actor has its respective role and name. In this, one actor initiates the use case.
3. **Links:** The link is an instance of association, which associates the objects and actors. It portrays a relationship between the objects through which the messages are sent. It is represented by a solid line. The link helps an object to connect with or navigate to another object, such that the message flows are attached to links.
4. **Messages:** It is a communication between objects which carries information and includes a sequence number, so that the activity may take place. It is represented by a labeled arrow, which is placed near a link. The messages are sent from the sender to the receiver, and the direction must be navigable in that particular direction. The receiver must understand the message.

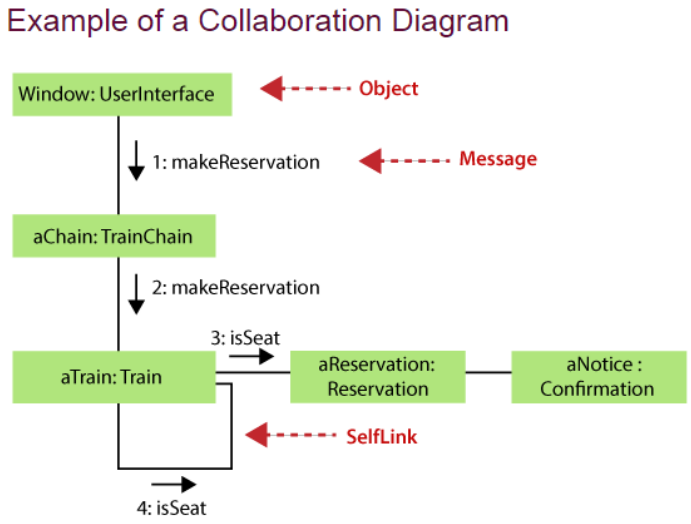


Example:

In this example, the collaboration diagram illustrates the interactions between objects (Customer, Order, Payment, and Warehouse) during the online shopping process. The arrows represent messages and method calls exchanged between the objects to complete the use case.



Collaboration diagrams provide valuable insights into the communication and collaboration patterns among objects, aiding in software design, analysis, and documentation.



Benefits of a Collaboration Diagram

1. The collaboration diagram is also known as **Communication Diagram**.
2. It mainly puts emphasis on the structural aspect of an interaction diagram, i.e., how lifelines are connected.
3. The syntax of a collaboration diagram is similar to the sequence diagram; just the difference is that the lifeline does not consist of tails.
4. The messages transmitted over sequencing are represented by numbering each individual message.
5. The collaboration diagram is semantically weak in comparison to the sequence diagram.
6. The special case of a collaboration diagram is the object diagram.
7. It focuses on the elements and not the message flow, like sequence diagrams.
8. Since the collaboration diagrams are not that expensive, the sequence diagram can be directly converted to the collaboration diagram.

9. There may be a chance of losing some amount of information while implementing a collaboration diagram with respect to the sequence diagram.

The drawback of a Collaboration Diagram

1. Multiple objects residing in the system can make a complex collaboration diagram, as it becomes quite hard to explore the objects.
2. It is a time-consuming diagram.
3. After the program terminates, the object is destroyed.
4. As the object state changes momentarily, it becomes difficult to keep an eye on every single that has occurred inside the object of a system.

****