Sequence Diagrams – Unified Modeling Language (UML)

A Sequence Diagram is a key component of Unified Modeling Language (UML) used to visualize the interaction between objects in a sequential order. It focuses on how objects communicate with each other over time, making it an essential tool for modeling dynamic behavior in a system. Sequence diagrams illustrate object interactions, message flows, and the sequence of operations, making them valuable for understanding use cases, designing system architecture, and documenting complex processes.

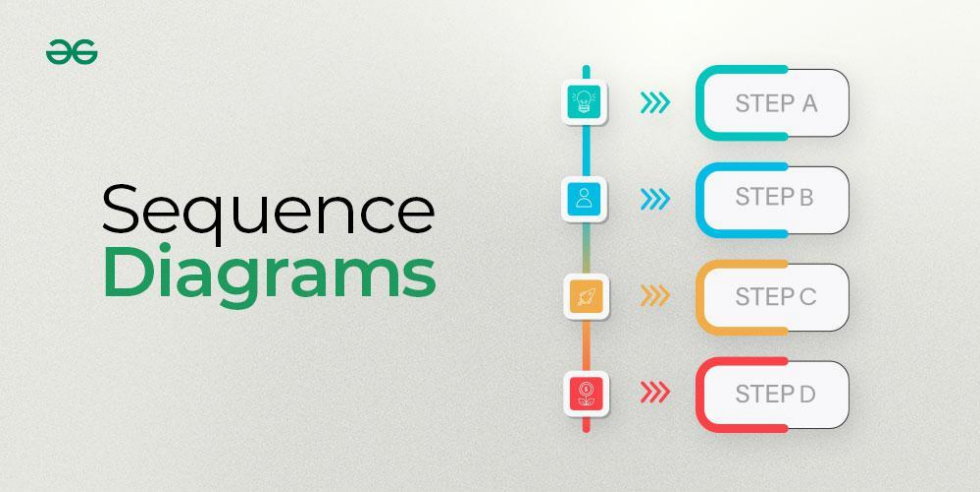


Table of Content

- How to create Sequence Diagrams?
- Use cases of Sequence Diagrams
- Challenges of using Sequence Diagrams

For those looking to sharpen their skills in UML and sequence diagramming, the System Design Course provides detailed explanations and practical exercises to help you effectively model systems.

What are Sequence Diagrams?

Sequence diagrams are a type of UML (Unified Modeling Language) diagram that visually represent the interactions between objects or components in a system over time. They focus on the order and timing of messages or events exchanged between different system elements. The diagram captures how objects communicate with each other through a series of messages, providing a clear view of the sequence of operations or processes.

Why use Sequence Diagrams?

Sequence diagrams are used because they offer a clear and detailed visualization of the interactions between objects or components in a system, focusing on the order and timing of these interactions. Here are some key reasons for using sequence diagrams:

- **Visualizing Dynamic Behavior**: Sequence diagrams depict how objects or systems interact with each other in a sequential manner, making it easier to understand dynamic processes and workflows.
- **Clear Communication**: They provide an intuitive way to convey system behavior, helping teams understand complex interactions without diving into code.
- **Use Case Analysis**: Sequence diagrams are useful for analyzing and representing use cases, making it clear how specific processes are executed within a system.
- **Designing System Architecture**: They assist in defining how various components or services in a system communicate, which is essential for designing complex, distributed systems or service-oriented architectures.
- **Documenting System Behavior**: Sequence diagrams provide an effective way to document how different parts of a system work together, which can be useful for both developers and maintenance teams.
- **Debugging and Troubleshooting**: By modeling the sequence of interactions, they help identify potential bottlenecks, inefficiencies, or errors in system processes.

Sequence Diagram Notations

1. Actors

An actor in a UML diagram represents a type of role where it interacts with the system and its objects. It is important to note here that an actor is always outside the scope of the system we aim to model using the UML diagram.

Notation symbol for actor

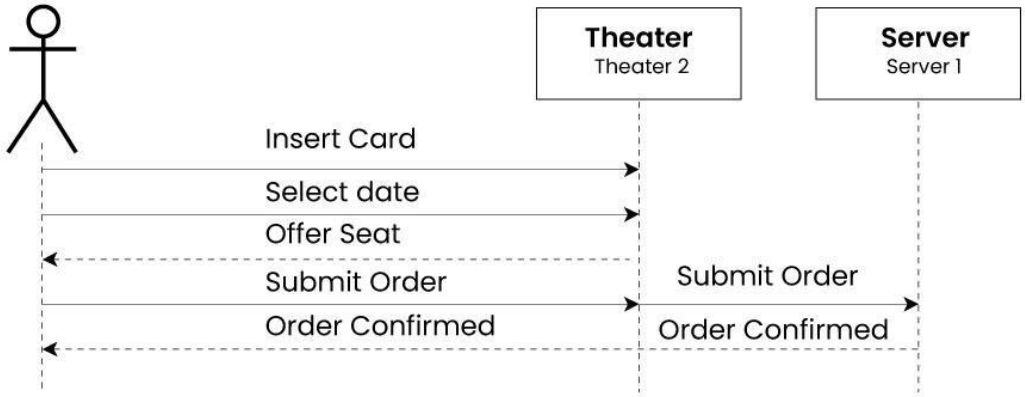
Sequence Diagrams

We use actors to depict various roles including human users and other external subjects. We represent an actor in a UML diagram using a stick person notation. We can have multiple actors in a sequence diagram.

For example:

Here the user in seat reservation system is shown as an actor where it exists outside the system and is not a part of the system.

User interacting with seat reservation system

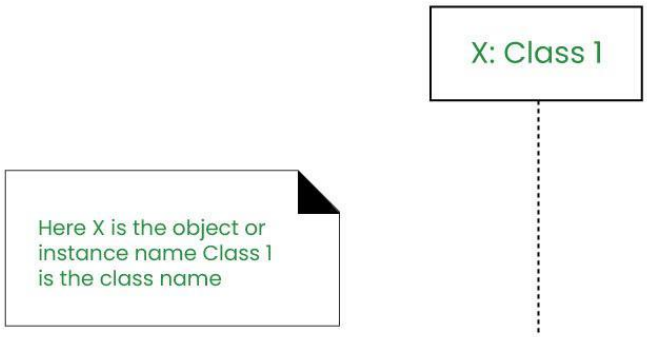


Sequence Diagrams

2. Lifelines

A lifeline is a named element which depicts an individual participant in a sequence diagram. So basically each instance in a sequence diagram is represented by a lifeline. Lifeline elements are located at the top in a sequence diagram. The standard in UML for naming a lifeline follows the following format:
Instance Name : Class Name

Lifeline



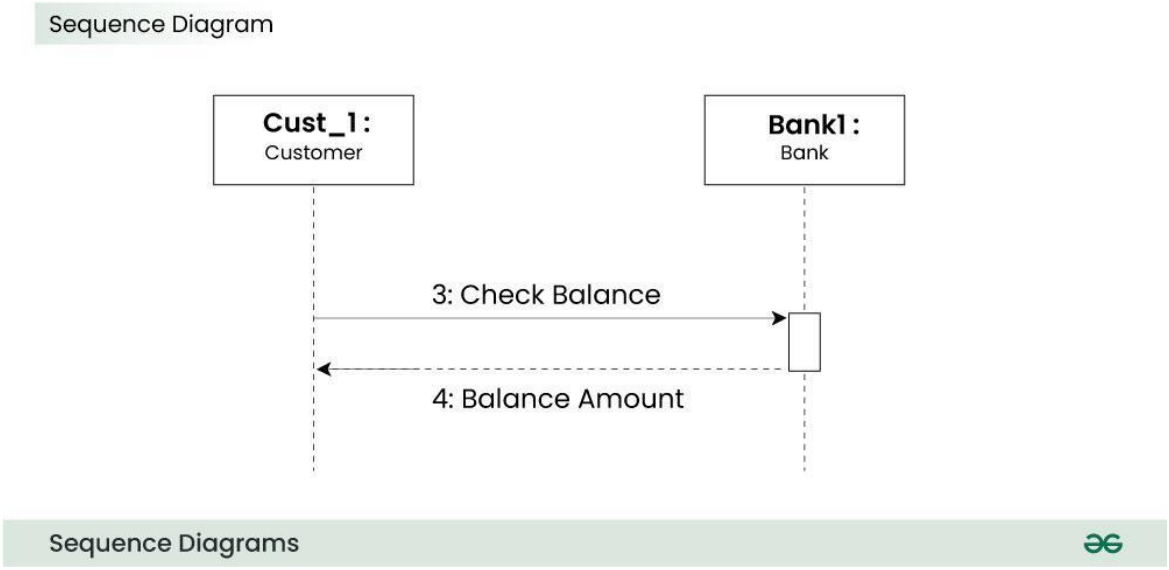
Sequence Diagrams

We display a lifeline in a rectangle called head with its name and type. The head is located on top of a vertical dashed line (referred to as the stem) as shown above.

- If we want to model an unnamed instance, we follow the same pattern except now the portion of lifeline's name is left blank.

- **Difference between a lifeline and an actor**
    - A lifeline always portrays an object internal to the system whereas actors are used to depict objects external to the system.

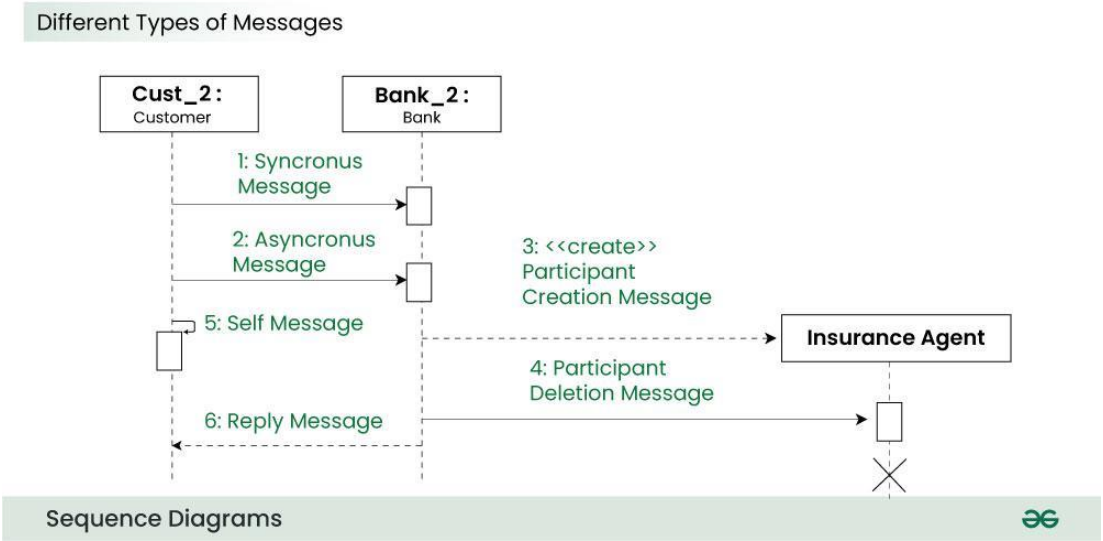
The following is an example of a sequence diagram:



3. Messages

Communication between objects is depicted using messages. The messages appear in a sequential order on the lifeline.

- We represent messages using arrows.
- Lifelines and messages form the core of a sequence diagram.

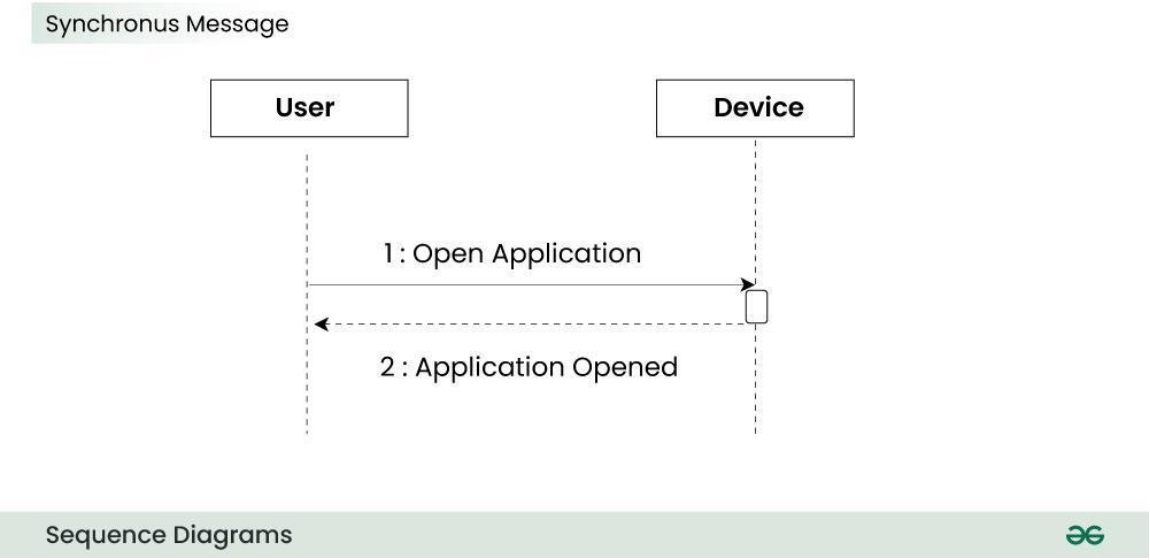


Messages can be broadly classified into the following categories:

1. Synchronous messages

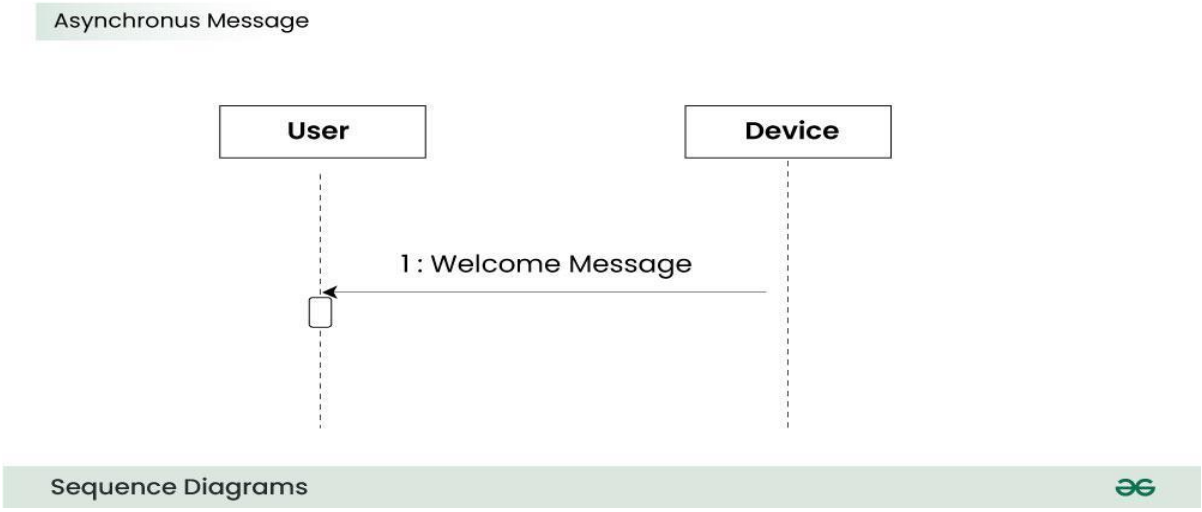
A synchronous message waits for a reply before the interaction can move forward. The sender waits until the receiver has completed the processing of the message. The caller continues only when it knows that the receiver has processed the previous message i.e. it receives a reply message.

- A large number of calls in object oriented programming are synchronous.
- We use a **solid arrow head** to represent a synchronous message.



2. Asynchronous Messages

An asynchronous message does not wait for a reply from the receiver. The interaction moves forward irrespective of the receiver processing the previous message or not. We use a lined arrow head to represent an asynchronous message.

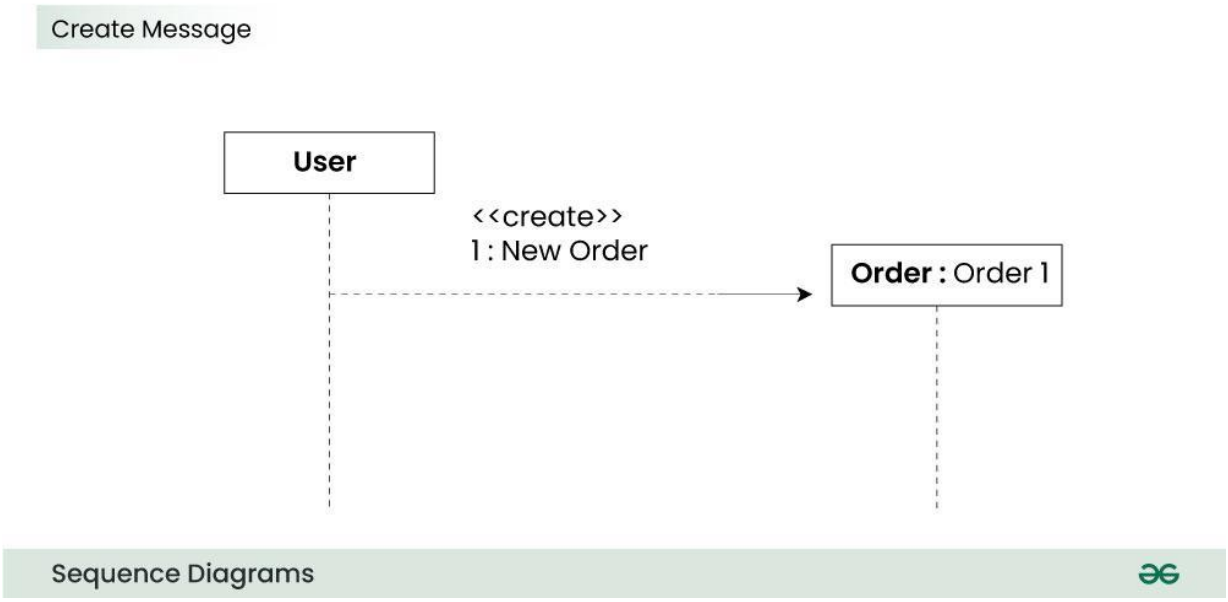


4. Create message

We use a Create message to instantiate a new object in the sequence diagram. There are situations when a particular message call requires the creation of an object. It is represented with a dotted arrow and create word labelled on it to specify that it is the create Message symbol.

For example:

The creation of a new order on a e-commerce website would require a new object of Order class to be created.



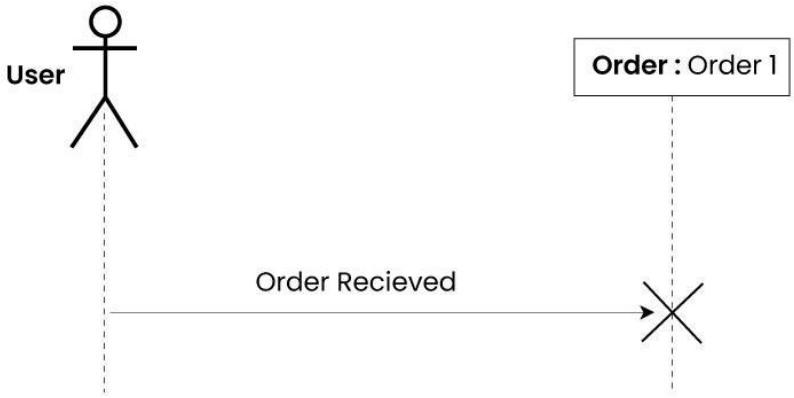
5. Delete Message

We use a Delete Message to delete an object. When an object is deallocated memory or is destroyed within the system we use the Delete Message symbol. It destroys the occurrence of the object in the system.It is represented by an arrow terminating with a x.

For example:

In the scenario below when the order is received by the user, the object of order class can be destroyed.

Delete Image

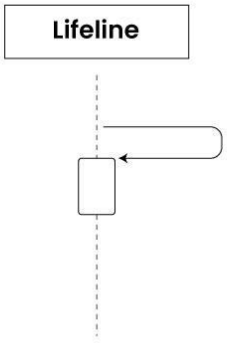


Sequence Diagrams

6. Self Message

Certain scenarios might arise where the object needs to send a message to itself. Such messages are called Self Messages and are represented with a **U shaped arrow**.

Self Image

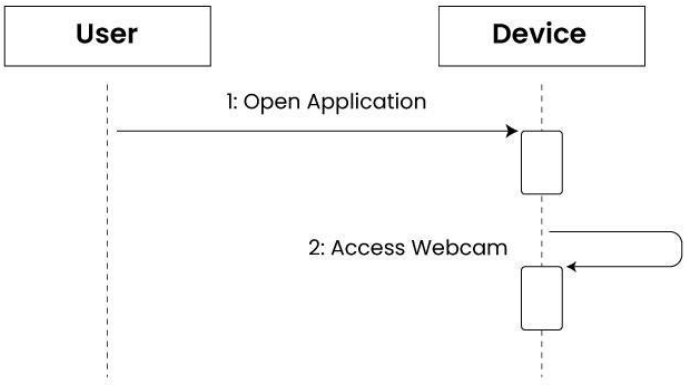


Sequence Diagrams

For example:

Consider a scenario where the device wants to access its webcam. Such a scenario is represented using a self message.

Self Image

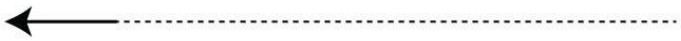


Sequence Diagrams

7. Reply Message

Reply messages are used to show the message being sent from the receiver to the sender. We represent a return/reply message using an **open arrow head with a dotted line**. The interaction moves forward only when a reply message is sent by the receiver.

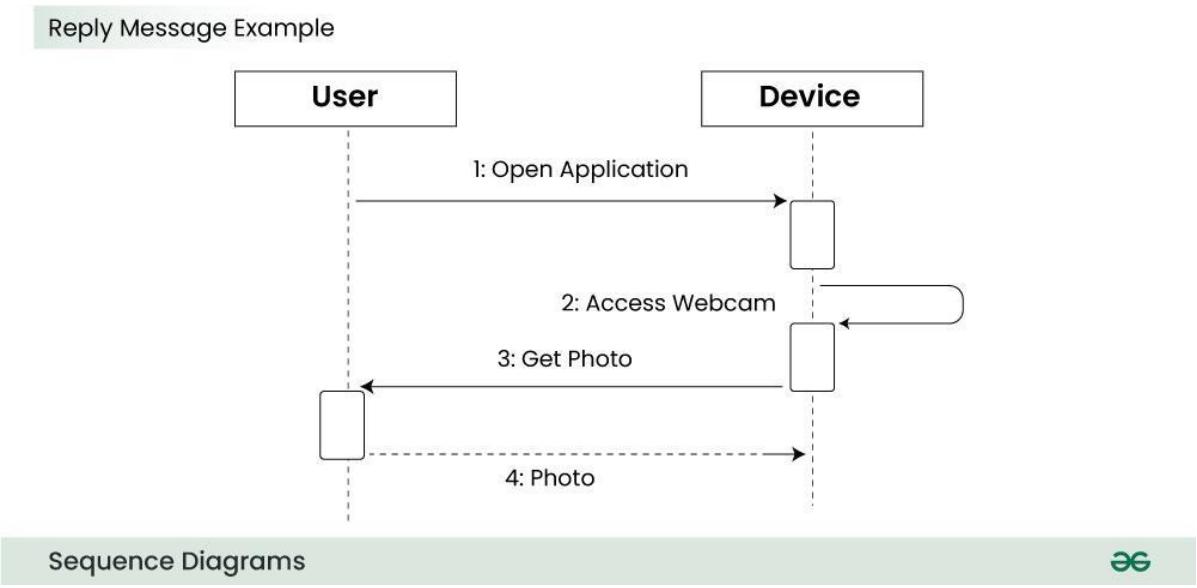
Reply Message



Sequence Diagrams

For example:

Consider the scenario where the device requests a photo from the user. Here the message which shows the photo being sent is a reply message.

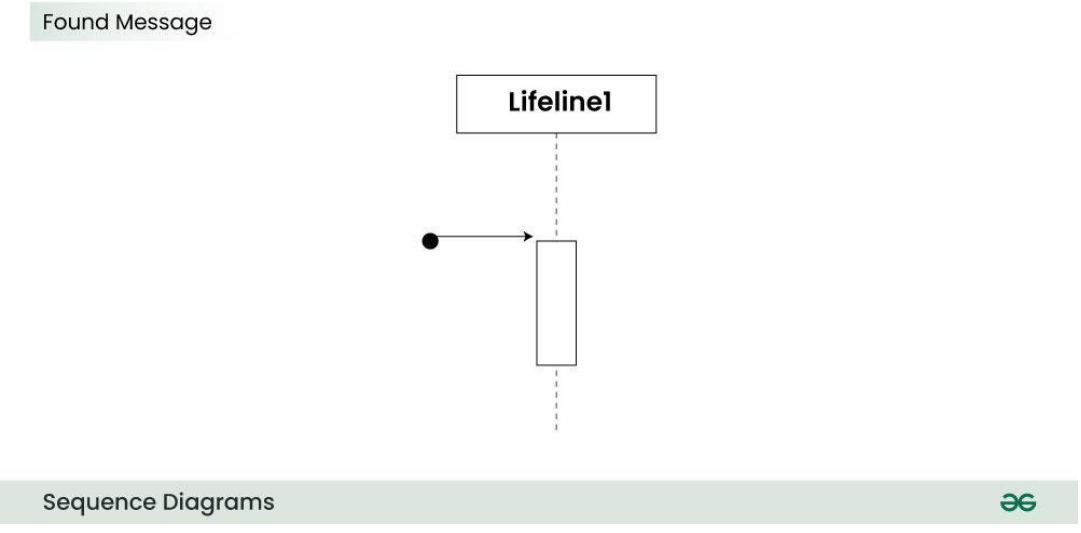


8. Found Message

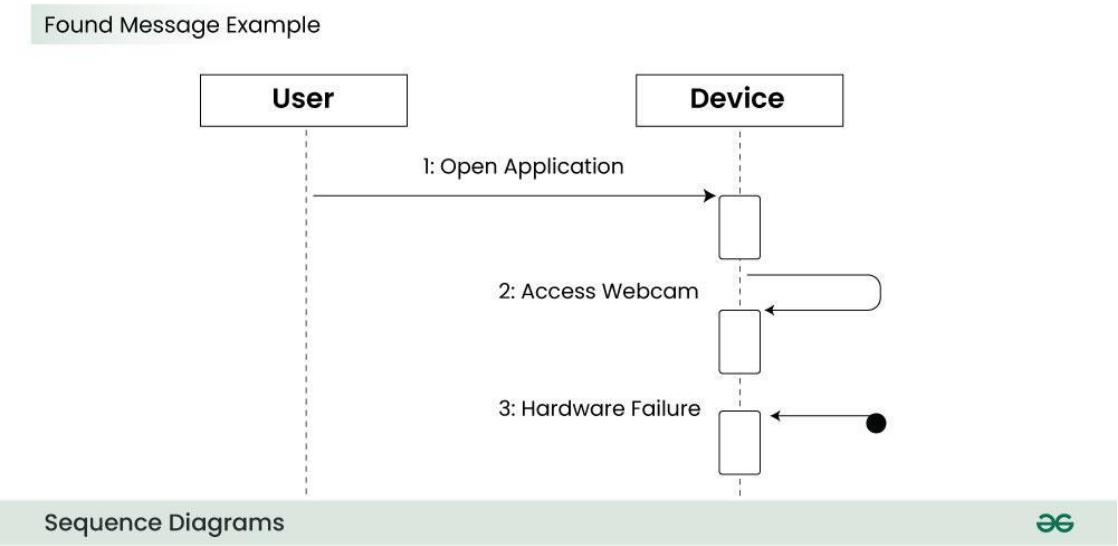
A Found message is used to represent a scenario where an unknown source sends the message. It is represented using an **arrow directed towards a lifeline** from an end point.

For example:

Consider the scenario of a hardware failure.



It can be due to multiple reasons and we are not certain as to what caused the hardware failure.

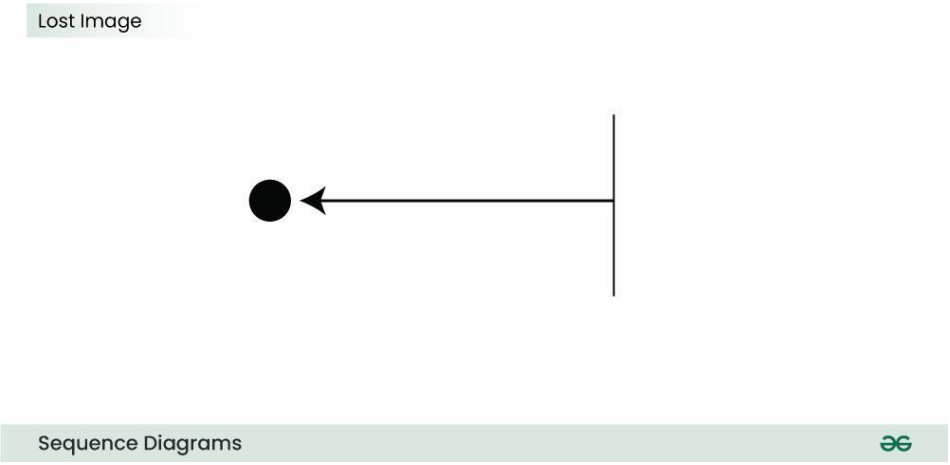


9. Lost Message

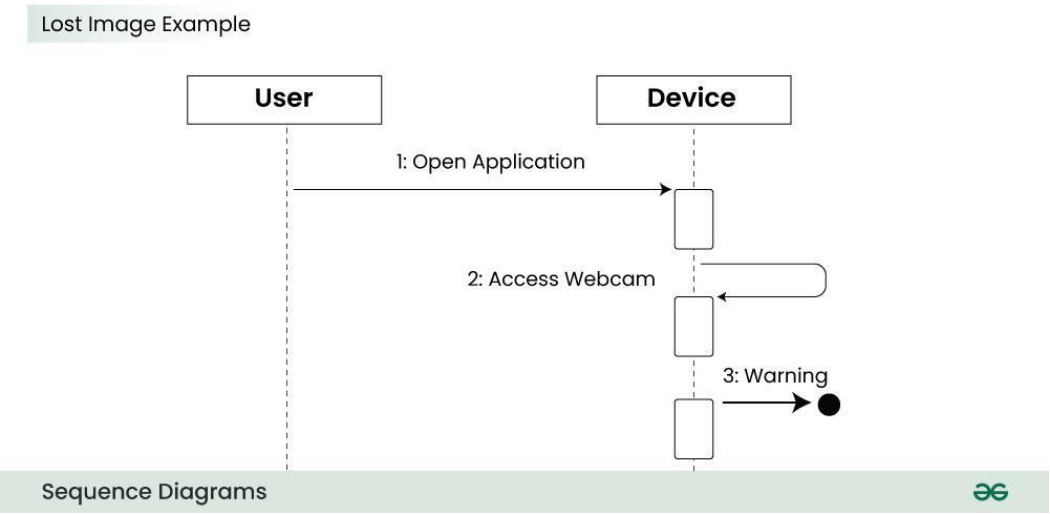
A Lost message is used to represent a scenario where the recipient is not known to the system. It is represented using an arrow directed towards an end point from a lifeline.

For example:

Consider a scenario where a warning is generated.



The warning might be generated for the user or other software/object that the lifeline is interacting with. Since the destination is not known before hand, we use the Lost Message symbol.

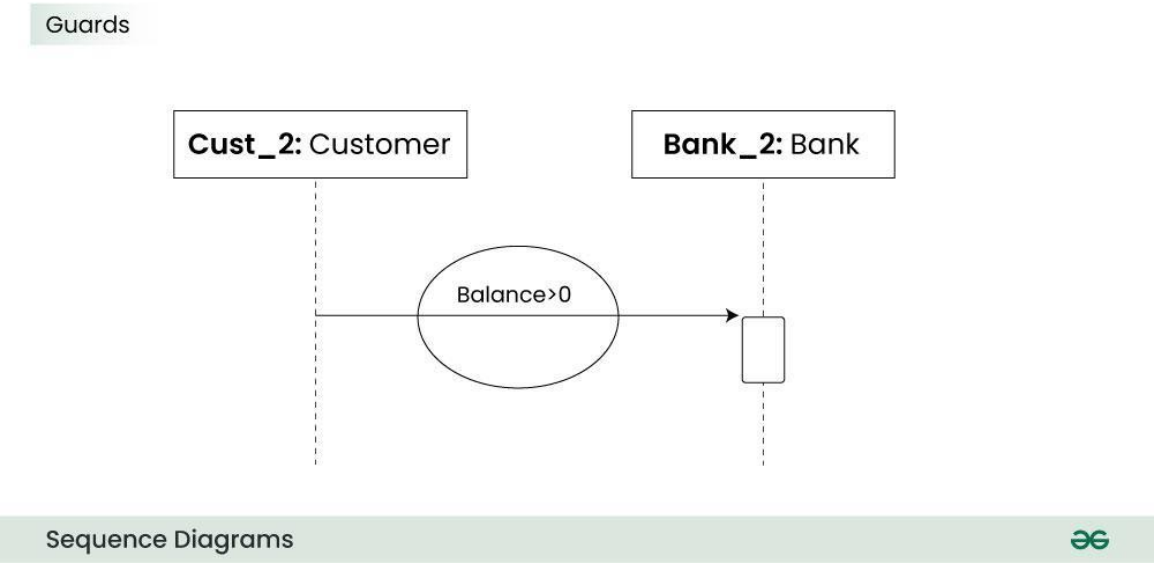


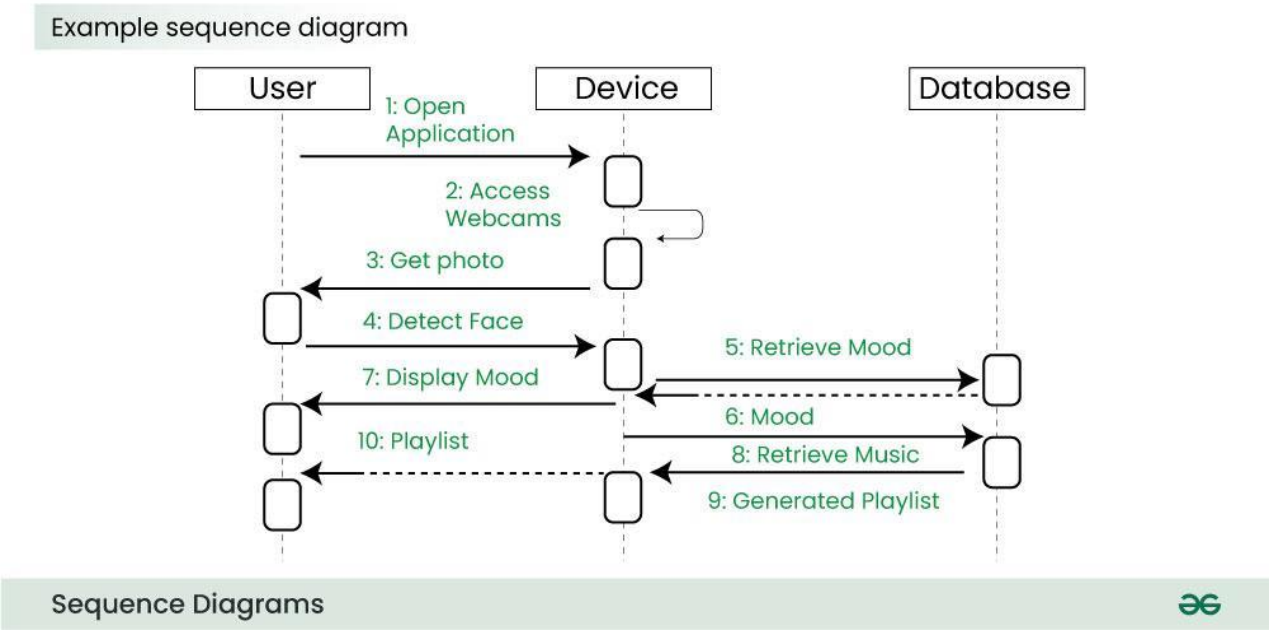
10. Guards

To model conditions we use guards in UML. They are used when we need to restrict the flow of messages on the pretext of a condition being met. Guards play an important role in letting software developers know the constraints attached to a system or a particular process.

For example:

In order to be able to withdraw cash, having a balance greater than zero is a condition that must be met as shown below.





The above sequence diagram depicts the sequence diagram for an emotion based music player:

- Firstly the application is opened by the user.
- The device then gets access to the web cam.
- The webcam captures the image of the user.
- The device uses algorithms to detect the face and predict the mood.
- It then requests database for dictionary of possible moods.
- The mood is retrieved from the database.
- The mood is displayed to the user.
- The music is requested from the database.
- The playlist is generated and finally shown to the user.

How to create Sequence Diagrams?

Creating a sequence diagram involves several steps, and it's typically done during the design phase of software development to illustrate how different components or objects interact over time. Here's a step-by-step guide on how to create sequence diagrams:

- **Step 1: Identify the Scenario:**
    - Understand the specific scenario or use case that you want to represent in the sequence diagram. This could be a specific interaction between objects or the flow of messages in a particular process.
- **Step 2: List the Participants:**
    - Identify the participants (objects or actors) involved in the scenario. Participants can be users, systems, or external entities.

- **Step 3: Define Lifelines:**
    - Draw a vertical dashed line for each participant, representing the lifeline of each object over time. The lifeline represents the existence of an object during the interaction.
- **Step 4: Arrange Lifelines:**
    - Position the lifelines horizontally in the order of their involvement in the interaction. This helps in visualizing the flow of messages between participants.
- **Step 5: Add Activation Bars:**
    - For each message, draw an activation bar on the lifeline of the sending participant. The activation bar represents the duration of time during which the participant is actively processing the message.
- **Step 6: Draw Messages:**
    - Use arrows to represent messages between participants. Messages flow horizontally between lifelines, indicating the communication between objects. Different types of messages include synchronous (solid arrow), asynchronous (dashed arrow), and self-messages.
- **Step 7: Include Return Messages:**
    - If a participant sends a response message, draw a dashed arrow returning to the original sender to represent the return message.
- **Step 8: Indicate Timing and Order:**
    - Use numbers to indicate the order of messages in the sequence. You can also use vertical dashed lines to represent occurrences of events or the passage of time.
- **Step 9: Include Conditions and Loops:**
    - Use combined fragments to represent conditions (like if statements) and loops in the interaction. This adds complexity to the sequence diagram and helps in detailing the control flow.
- **Step 10: Consider Parallel Execution:**
    - If there are parallel activities happening, represent them by drawing parallel vertical dashed lines and placing the messages accordingly.
- **Step 11: Review and Refine:**
    - Review the sequence diagram for clarity and correctness. Ensure that it accurately represents the intended interaction. Refine as needed.
- **Step 12: Add Annotations and Comments:**

    - Include any additional information, annotations, or comments that provide context or clarification for elements in the diagram.
- **Step 13: Document Assumptions and Constraints:**
    - If there are any assumptions or constraints related to the interaction, document them alongside the diagram.
- **Step 14: Tools:**
    - Use a UML modeling tool or diagramming software to create a neat and professional-looking sequence diagram. These tools often provide features for easy editing, collaboration, and documentation.

Use cases of Sequence Diagrams

- **System Behavior Visualization:**
    - Sequence diagrams are used to illustrate the dynamic behavior of a system by showing the interactions among various components, objects, or actors over time.
    - They provide a clear and visual representation of the flow of messages and events in a specific scenario.
- **Software Design and Architecture:**
    - During the design phase of software development, sequence diagrams help developers and architects plan and understand how different components and objects will interact to accomplish specific functionalities.
    - They provide a blueprint for the system's behavior.
- **Communication and Collaboration:**
    - Sequence diagrams serve as a communication tool among stakeholders, including developers, designers, project managers, and clients.
    - They help in conveying complex interactions in an easy-to-understand visual format, fostering collaboration and shared understanding.
- **Requirements Clarification:**
    - When refining system requirements, sequence diagrams can be used to clarify and specify the expected interactions between system components or between the system and external entities.
    - They help ensure a common understanding of system behavior among all stakeholders.
- **Debugging and Troubleshooting:**

- Developers use sequence diagrams as a debugging tool to identify and analyze issues related to the order and timing of messages during system interactions.
- It provides a visual representation of the flow of control and helps in locating and resolving problems.

Challenges of using Sequence Diagrams

- **Complexity and Size:** As systems grow in complexity, sequence diagrams can become large and intricate. Managing the size of the diagram while still accurately representing the interactions can be challenging, and overly complex diagrams may become difficult to understand.
- **Abstraction Level:** Striking the right balance in terms of abstraction can be challenging. Sequence diagrams need to be detailed enough to convey the necessary information, but too much detail can overwhelm readers. It's important to focus on the most critical interactions without getting bogged down in minutiae.
- **Dynamic Nature:** Sequence diagrams represent dynamic aspects of a system, and as a result, they may change frequently during the development process. Keeping sequence diagrams up-to-date with the evolving system can be a challenge, especially in rapidly changing or agile development environments.
- **Ambiguity in Messages:** Sometimes, it can be challenging to define the exact nature of messages between objects. Ambiguity in message content or meaning may lead to misunderstandings among stakeholders and impact the accuracy of the sequence diagram.
- **Concurrency and Parallelism:** Representing concurrent and parallel processes can be complex. While sequence diagrams have mechanisms to indicate parallel execution, visualizing multiple interactions happening simultaneously can be challenging and may require additional diagrammatic elements.
- **Real-Time Constraints:** Representing real-time constraints and precise timing requirements can be challenging. While sequence diagrams provide a sequential representation, accurately capturing and communicating real-time aspects might require additional documentation or complementary diagrams.