Sequence Diagram

Understand how sequence diagrams represent interactions between actors and objects.

We'll cover the following

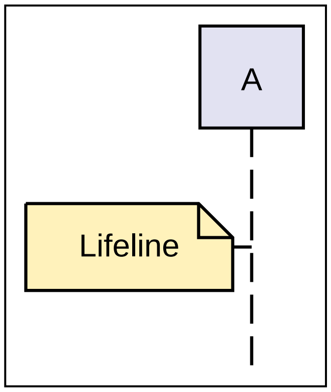
A **sequence diagram** is a form of communication diagram that illustrates how different actors and objects interact with each other or between themselves. The diagram represents these interactions as an exchange of messages between various entities and the type of exchange. Sequence diagrams also demonstrate the sequence of events that occur in a specific use case and the logic behind different operations and functions.

Elements of a sequence diagram

Various elements make up a sequence diagram. Let's discuss some of the essential elements of this diagram that appear most often.

Lifeline

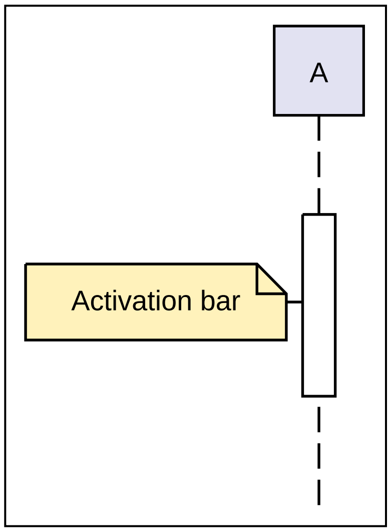
In sequence diagrams, we write all entities horizontally. Each entity has a **lifeline** that represents its existence, i.e., when the entity is activated or deactivated. We illustrate this using a dotted line below the entity. We represent different objects, actors, entities, or boundaries in a system using lifelines, and they never overlap each other. The illustration below shows how to visualize a lifeline in UML. The horizontal boxes are used to denote the objects involved in the interaction.



Lifeline in a sequence diagram

Activation bars

Activation bars indicate the **active period** of an object, that is, the time when an object sends or receives messages. We draw these using simple vertical boxes on the lifeline. Here's an example of an activation bar:

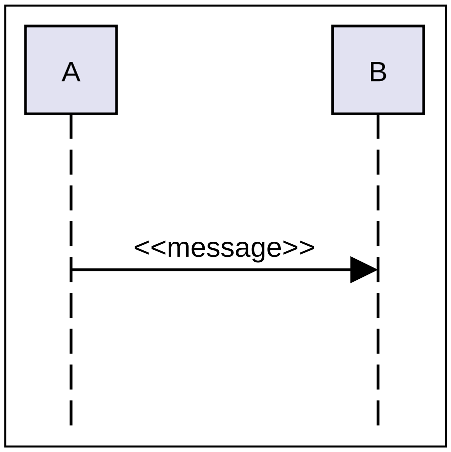


Activation bar in a sequence diagram

Messages

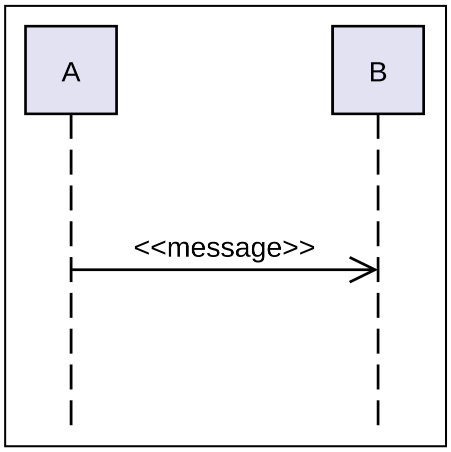
In sequence diagrams, a message is an interaction between two objects. It can be in the form of sending and receiving messages, invoking an operation, or creating a new object or entity. Messages are drawn horizontally in any direction: left to right, right to left, or back to themselves. Different kinds of messages are represented using different arrows. Let's look at the types of messages:

1. A **synchronous message** is a type of message where the sender has to wait for the receiver to return a response before it can perform another operation. We can draw synchronous messages using a solid line with a filled arrowhead.



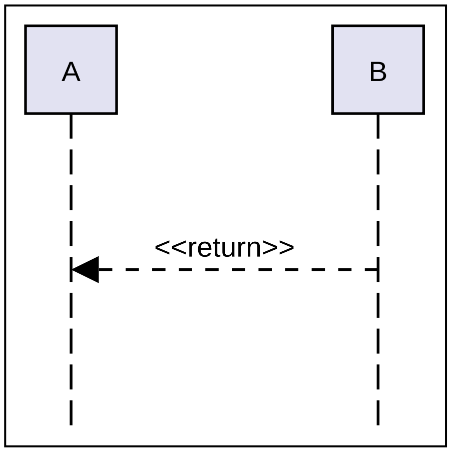
A synchronous message

2. An **asynchronous message** is a type of message where the sender does not have to wait for a response from the receiver. The sender can continue sending messages to other objects. We can draw the asynchronous messages using a solid line with an open arrowhead.



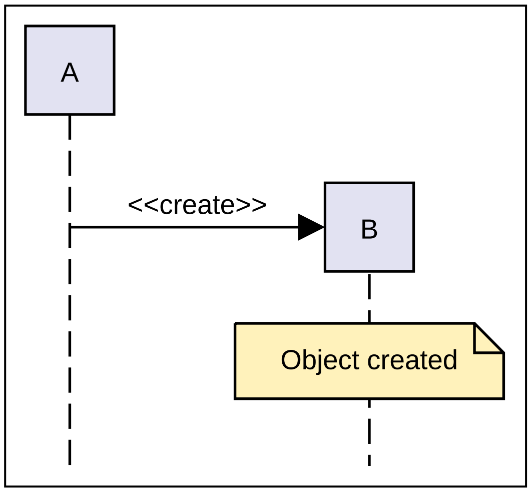
An asynchronous message

3. A **synchronous return** is a type of message generated in response to a synchronous call. A synchronous message has to be paired with a synchronous return message. It means that the receiver has processed the message and sent a response. We can draw synchronous return messages using a dotted line with a filled arrowhead.



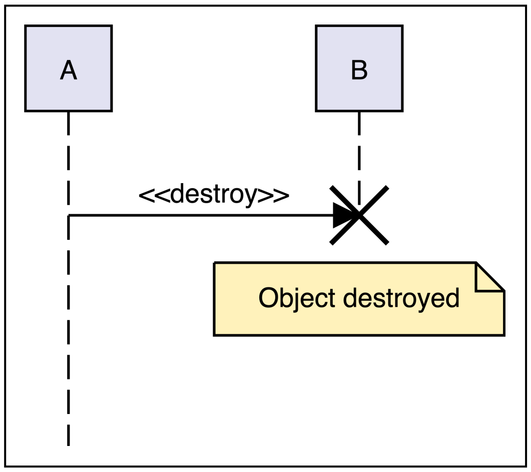
A synchronous return message

4. A **create message** indicates that a new object is created during an interaction. New objects can be created as a result of some message or operation. We can draw it like this:



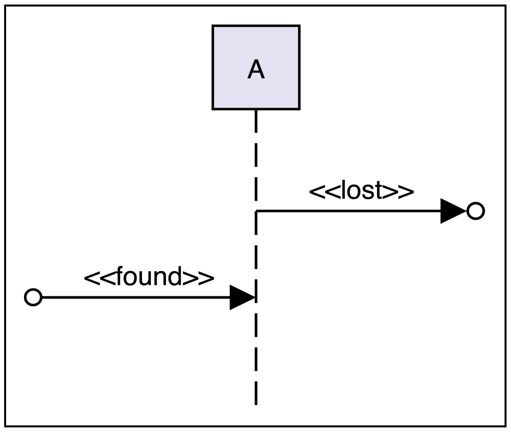
A create message

5. A **destroy message** indicates that an object is destroyed during a sequence of events. Objects can be destroyed as a result of some message or operation, and their lifeline ends. We can draw it like this:



A destroy message

6. A **lost message** is a message that initiates from an object but does not reach its endpoint. It appears as a message that is terminated. A **found message** is a message that is received, but the sender is unknown. It appears as a message that reaches an endpoint but does not initiate from any object. We draw lost messages as an arrow ending with a circle and found messages with an arrow starting with a circle.



The lost and found messages

> **Note:** Some sequence diagrams use filled circles to represent the lost and found messages. However, throughout the course we will apply the empty circle convention.

How to draw a sequence diagram

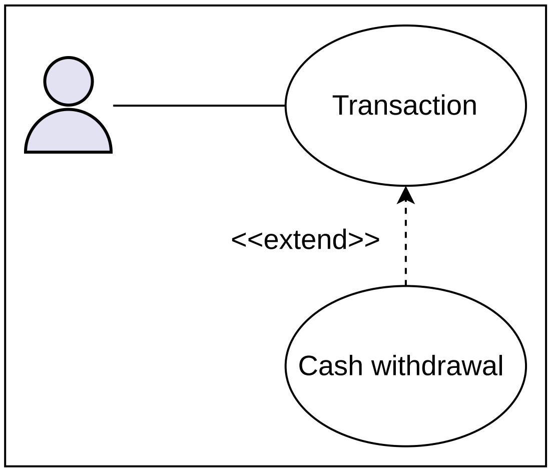
To create a self-explanatory sequence diagram, there are a number of steps that we need to follow. Before we start, it is important to understand that there isn't one correct way of creating a sequence diagram. Some of us may have a different approach to handling these problems.

In this lesson, we introduce a methodology that will help us break down the problem into smaller, achievable tasks. Let's take a look at the steps.

Identify the use case

To get started with the sequence diagram, it is essential to know our use case. The sequence diagram is a means to define the sequential order of events that occur in that use case.

Let's build on a simple scenario where a customer wants to withdraw cash from an ATM. Our use case will be as follows:



Use case for cash withdrawal

Identify the actors and objects

Now, we need to list down all the actors and objects that will be involved in the entire sequence. These would be the following:

- Customer
- ATM
- Transaction

- Account
- Cash dispenser

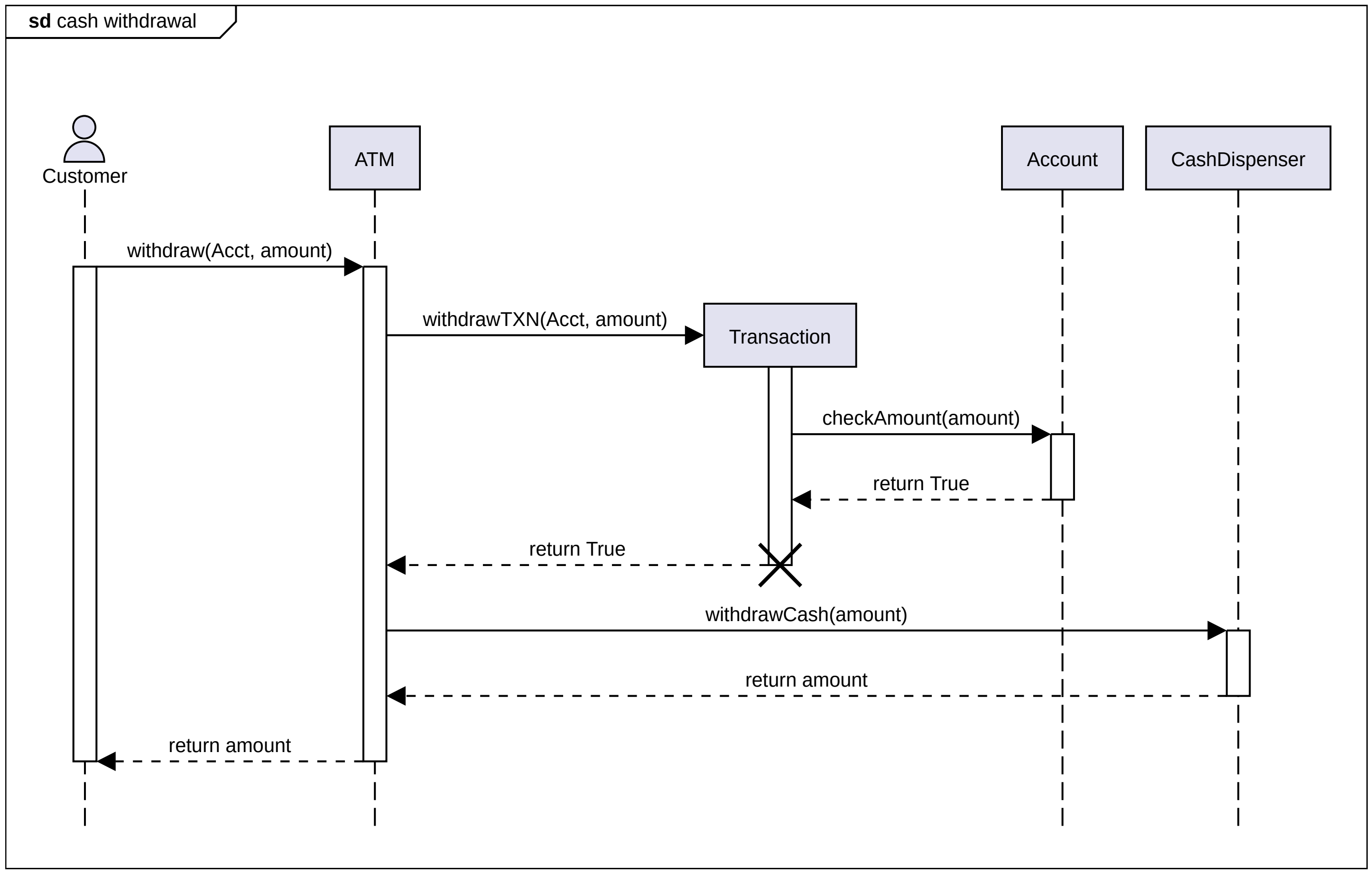
Identify the order of actions

We have identified the entities involved in the cash withdrawal sequence, but we don't know how these entities will interact with each other. It is important that we note down the action and the order in which they will occur. Here's how we can define the interaction in steps:

1. The customer requests a cash withdrawal from the ATM with their account information and the amount.
2. The ATM initiates a cash withdrawal transaction against the account and the given amount.
3. The transaction amount is verified for the given account.
4. In case the amount entered is valid, the account verifies the transaction.
5. The ATM signals the cash dispenser to release the required cash amount.
6. The cash dispenser confirms the release of cash.
7. The ATM prompts the user to collect the cash.

Create the diagram

We have identified all the involved objects and entities in the interaction. We have also listed down the order in which the objects will perform these actions. To finally create the sequence diagram, we need to relate these steps with the type of messages that the objects will exchange with each other. Here's a possible diagram that we can create for the cash withdrawal scenario:



The sequence diagram for cash withdrawal

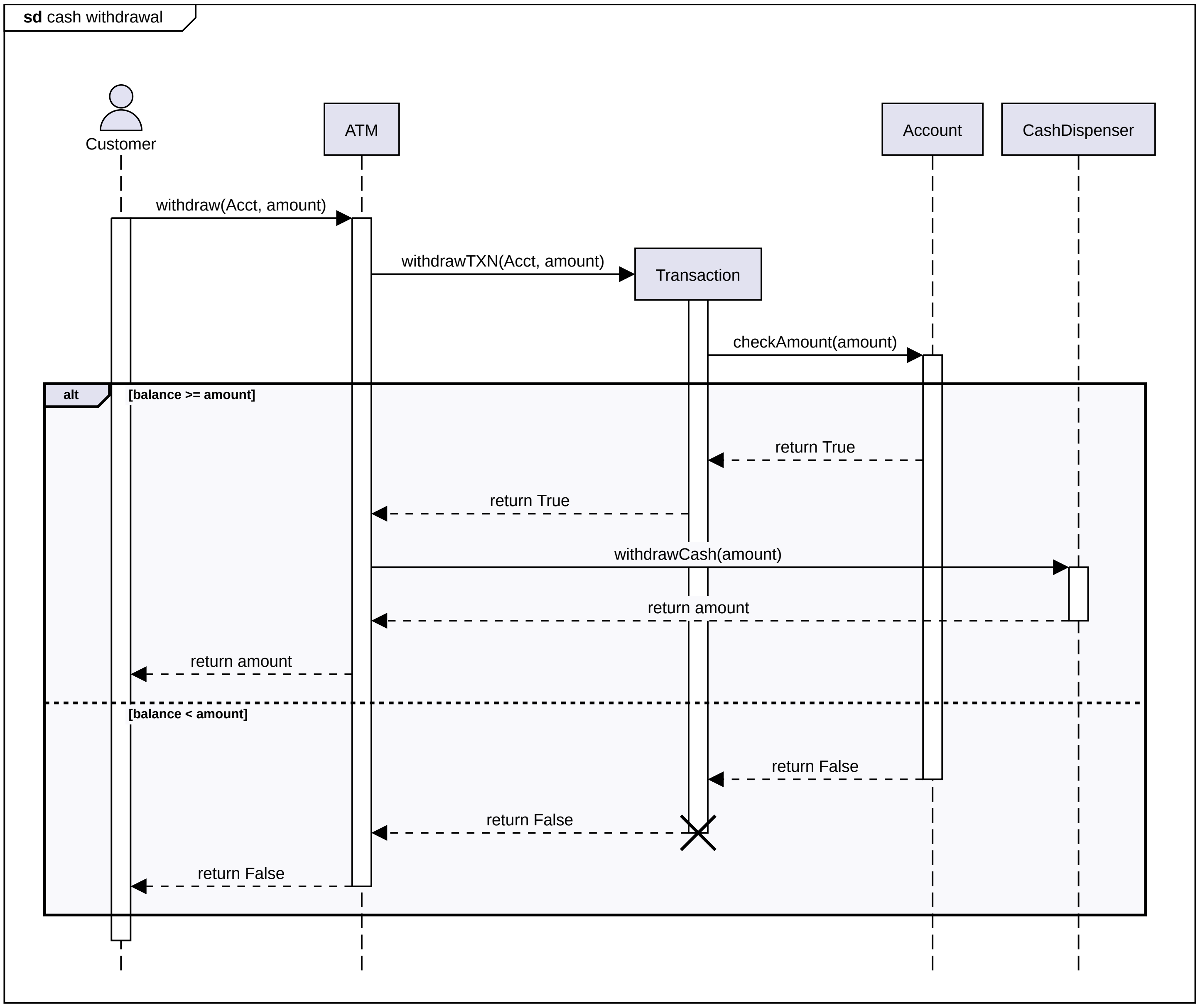
Fragment frame in the sequence diagram

Our goal is to keep our sequence diagram as simple as possible without clutter and information overload. However, sometimes we need to show more complex actions like conditional actions or loop sequences. Sequence diagrams allow us to represent this information using the sequence fragment component. We can identify the operator of a fragment in the top left corner of the fragment frame. Here are a few examples of fragment operators:

- **Alternative (alt):** This operator models the `if-else` condition. It divides the fragment into parts, and either of the parts can take place based on the guard condition.
- **Option (opt):** This operator models the `if` condition. The fragment will only execute if the guard condition is met. Otherwise, the entire fragment is skipped, and the interaction continues.

- **Loop (loop):** This operator represents a repetitive sequence. The fragment will keep repeating until the guard condition is met.
- **Reference (ref):** This operator assists in managing larger diagrams. It allows us to reuse parts of another sequence diagram by providing a reference to it.

Let's expand the sequence diagram we created to see how we can build on it further using a sequence fragment.



The sequence diagram for cash withdrawal with a sequence fragment

In the diagram above, we add the alternatives operator, which divides the sequence into two fragments. The first fragment takes place if the guard condition is met, while in any other case, the second fragment occurs. Here the guard condition is that the balance in the account should be greater than or equal to the amount requested by the customer.