⋆˚₀⋆✧˚ Chapter 7: Behavioral Modeling: Sequence Diagrams ˚✧⋆₀˚⋆

In the realm of software engineering, capturing the static structure of a system is only half the battle. While a map shows us the layout of a city, it tells us nothing of the traffic flowing through its streets or the rhythm of its daily life. To truly understand a system, we must look beyond its components and witness its **behavior**. Behavioral modeling serves as the pulse of the design process, shifting our focus from what a system *is* to how it *acts*.

Sequence diagrams are the definitive tool for this exploration, providing a chronological narrative of communication. By mapping the exchange of messages between objects over time, these models transform abstract logic into a clear, visual choreography. In this chapter, we delve into the mechanics of the **Sequence Diagram**, mastering the art of documenting interaction to ensure that every process—from the simplest login to the most complex transaction—is executed with precision and clarity.

1. Definition

The **Sequence Diagram** is the primary tool used to model the execution of a scenario within a system. It is widely considered the most important UML diagram after the Class Diagram because it focuses on the **sequential order** of interactions.

While developers use them to map out code execution, these diagrams are equally valuable for business stakeholders. They provide a clear visual for discussing how a business currently operates by showing how different entities—from users to databases—interact to accomplish a task. Essentially, it answers two questions: **Which messages are sent, and when?**

A sequence diagram is made up of two basic building blocks: the frame and the interaction.

2. Frame

The frame is the space in which the sequence diagram is drawn. It represents the border that surrounds the diagram and consists of a header and a contents area.

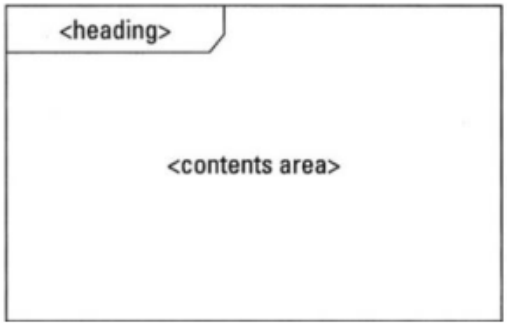

Figure 71. Frame representation in UML

3. Interaction

Simply put, an interaction is a sequence of messages exchanged between objects to accomplish a task. Objects can be created and destroyed. They can ask questions or make requests to other objects by invoking operations.

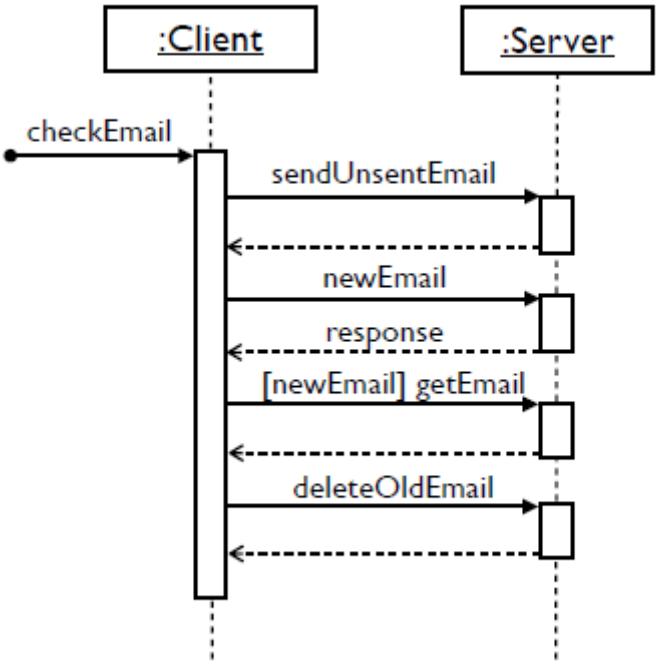

Figure 72. Example of interaction in sequence diagram

To help illustrate how interaction works in a real-world scenario, we can examine a common interaction: **Checking Email**. This process involves a series of messages between a Client (the user's device) and a Server. The interaction follows a specific chronological order to ensure data integrity and synchronization:

- The user clicks the "Check Email" button.
- First, the client sends all the emails that have not yet been sent.
- After receiving an acknowledgment of receipt, the client asks the server if there are any new emails.
- If there are, it downloads the new email.
- Then, it deletes the old overwritten emails from the server.

To draw a sequence diagram, we need to master three main elements: **who** is involved, **when** they are working, and **what** they are saying.

- **Lifeline (Who)**

The Lifeline represents an object's existence over time. It is represented by a box with the object's name at the top and a vertical dashed line stretching downward.

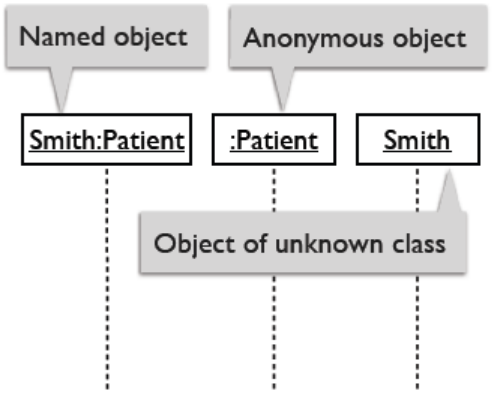

Figure 73. Example of representation of lifeline in a sequence diagram

In sequence diagrams, timing is everything. One of the most important concepts to visualize is the **creation of an object** during the system's execution. There is a critical difference between **object creation** and a **standard message call**. When the message arrow points directly to the **object box**, it indicates a **Constructor Message** (for example creation of the `:Performance`)**.** When the message arrow points to the **dashed lifeline**, it signify that the object already exists and is simply being invoked to execute a method.

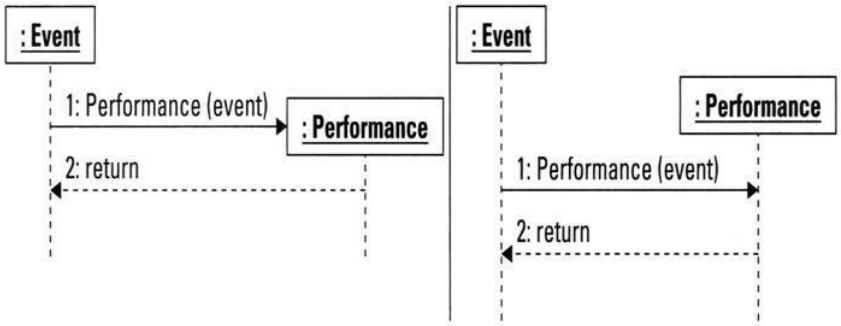

Figure 74. Example of difference between constructor and method calling in sequence diagram

- **Control focus (When)**

The control focus describes the period during which an object performs an action, either directly or through a subordinate procedure. The activation of an object is represented by the start and end of the control focus.

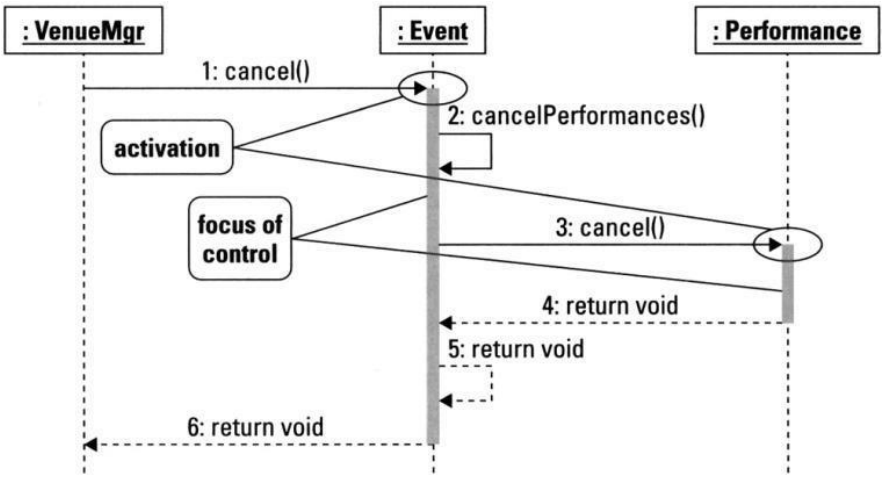

Figure 75. Example of control focus in sequence diagram

- **Message**

A message is the basic form of communication in a sequence diagram. It defines a specific type of communication. The communication may invoke an operation, and create or destroy an instance. The message specifies not only the type of communication but also the sender and the receiver. The message is represented by horizontal arrows. We can distinguish several types of message:

➢ Synchronous message

A synchronous message calls a method of an object and waits for a response. The call parameters must match the parameters of the invoked method's prototype.

➢ Asynchronous message

An asynchronous message calls a method of an object and does not wait for a response. The call parameters must match the parameters of the invoked method's prototype.

➢ Constructor message

A message that creates an object is called a constructor. It is represented by a dashed line with an open arrowhead.

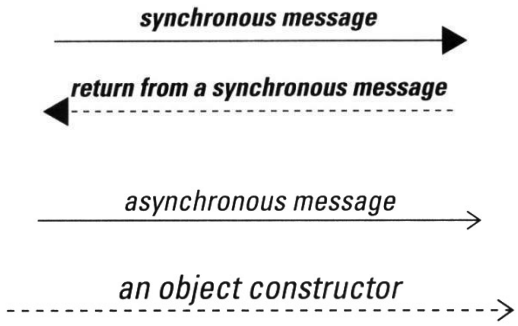

Figure 76. Principle of synchronous, asynchronous and constructor message

➢ Reflexive message

A reflexive message represents a call to an internal method of an object.

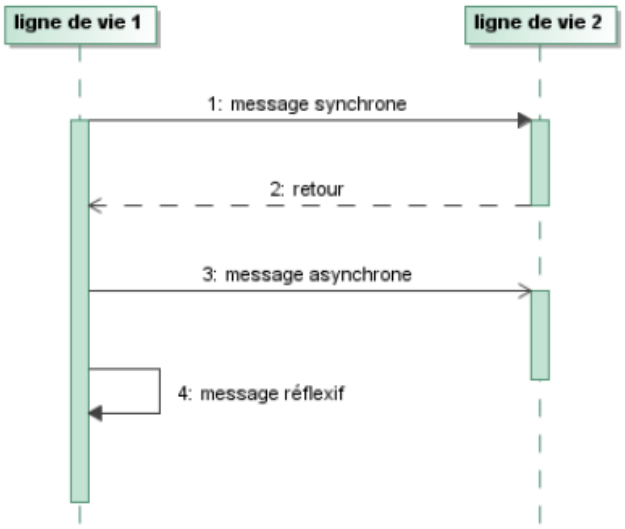

Figure 77. Example of reflexive message in sequence diagram

4. Example of sequence diagrams

In this section, we give a set of examples that resume most frequent blocs found in sequence diagrams.

- **Example of synchronous message**

In this example, labeled **sd Rechercher livre** (Search book), a **Client** initiates a synchronous interaction by invoking the chercher("Tintin") method on the **:Médiathèque** object. This is represented by a solid line with a filled arrowhead, indicating that the client must wait for a result before proceeding. Upon receiving the request, a **control focus** appears on the library's lifeline to show it is actively processing the search. Finally, the library returns the result, nombreLivres, via a dashed response arrow, which signals the end of the operation and allows the client to resume its activity.

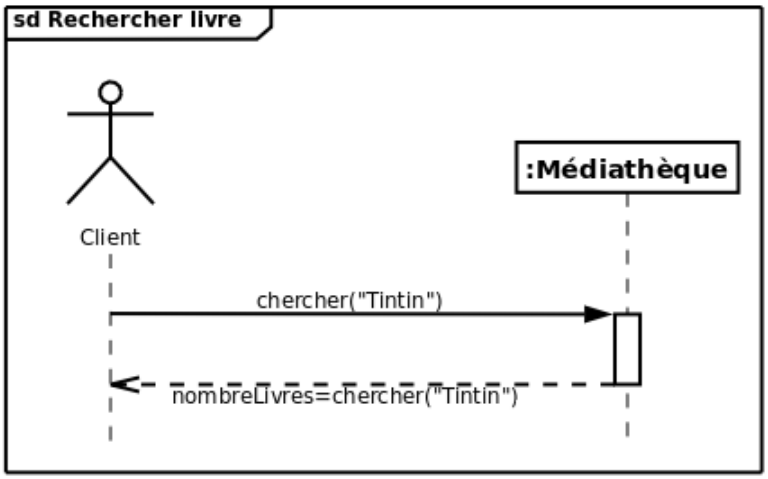

Figure 78. Example of synchronous message in sequence diagram

- **Example of asynchronous message**

In this **sd Retrait argent** (Cash withdrawal) example, a **Client** initiates an asynchronous interaction by sending the introduireCarte method to the **:Distributeur** object. This is represented by a solid line with an **open arrowhead**, signifying that the client triggers the action and does not wait for a response before continuing. The diagram highlights the message lifecycle through four key events: the **sending event** at the client's lifeline, the **reception event** at the dispenser, and the subsequent **start and end of execution** events. The control focus on the **:Distributeur** lifeline illustrates the period during which the

machine processes the card independently, allowing the system to handle multiple parallel actions without blocking the sender.

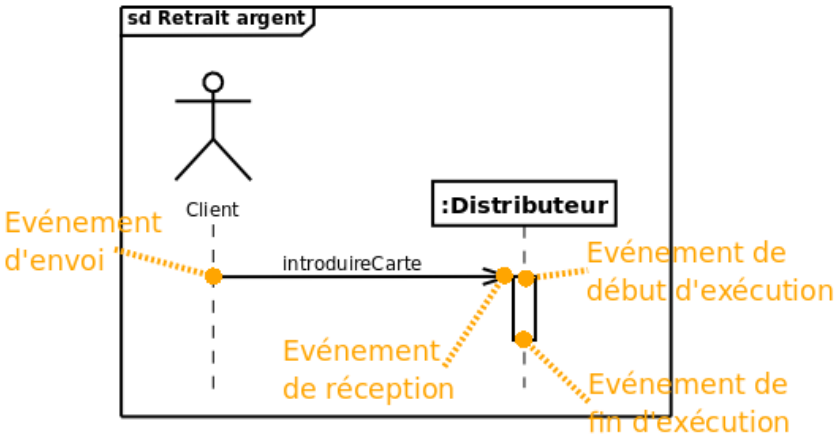

Figure 79. Example of asynchronous message in Cash withdrawal sequence diagram

Another example is the common interpersonal interaction, where the person **P1** sends the message Envoyer SMS (Send SMS) to the person **P2**. This is depicted as an asynchronous interaction because the sender, **P1**, does not wait for a response or a "read receipt" before moving on with their own lifeline. Once the message reaches **P2**, a brief control focus appears on their lifeline, representing the moment the message is received and processed, while **P1** remains free to initiate other tasks or complete their own execution independently.

Figure 80. Example of synchronous message in SMS delivery sequence diagram

- **Example of constructor and destructor messages**

In these examples, the system manages the lifecycle of an account within an online library environment. The object **:LibrairieEnLigne** creates a new instance **c: Compte** by invoking the Compte() constructor. This is represented by a dashed line with an open arrowhead pointing directly to the object's box, which is vertically positioned to mark the exact moment of its creation. The same system terminates the object's existence by sending a <<destroy>> message. This is visually confirmed by the large **red "X"** at the end

of the **c: Compte** lifeline, indicating that the object has been removed from memory and its lifecycle has officially ended.

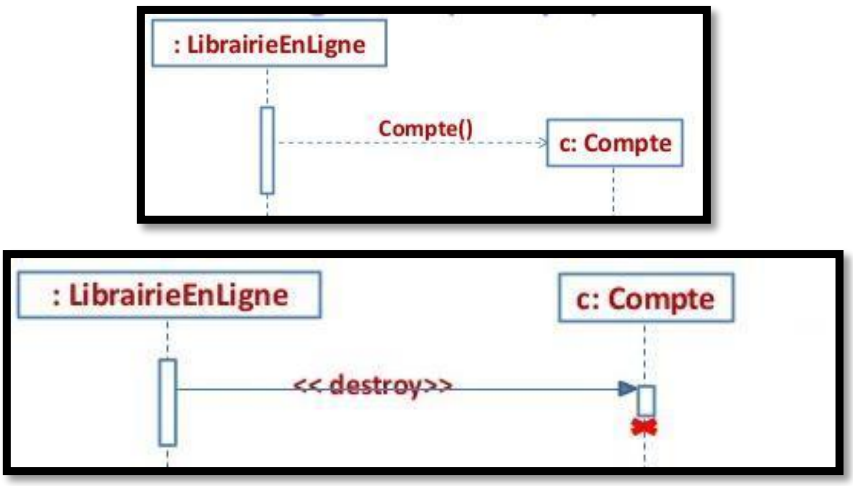

Figure 81. Example of object creation and destruction in a sequence diagram

- **Example of lost message and unknown source**

In this example, the diagram illustrates how a system handles communication failures and external inputs. The interaction begins with a small circle, representing a **found message** from an **unknown source**, where the sender exists outside the scope of the current diagram. The message introduireCarte (insertCard) is dispatched toward the **:Distributeur** object but is intercepted by a **red "X"**, which is the standard notation for a **lost message**. This signifies that the transmission failed and never reached its destination. As a result, the expected **start of execution event** on the dispenser's lifeline is never triggered, visually demonstrating a scenario where the receiver remains inactive because the signal was lost in transit.

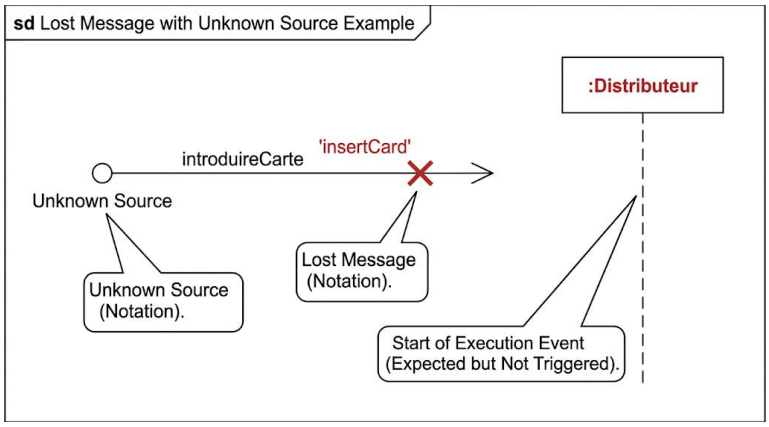

Figure 82. Example of lost message and unknown source in a sequence diagram

5. Advanced interaction fragments

In addition to standard message flows, UML sequence diagrams use **Interaction Fragments** (combined fragments) to model complex logic like conditions, loops, and error states. These are represented by frames with a specific operator in the top-left corner.

- **Option fragment (OPT)**

The **Opt** fragment models an "if-then" scenario. It contains a single interaction that only occurs if a specific **guard condition** is met. If [condition] is **True**, the Action() is performed. If **False**, the entire block is ignored.

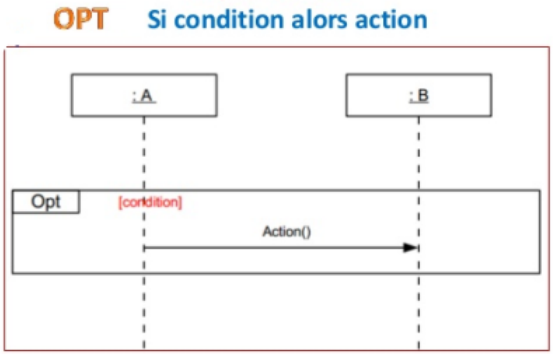

Figure 83. Representation of option fragment in a sequence diagram

- **Iterative fragment (LOOP)**

The Loop fragment represents repetitive behavior. The interaction inside the frame is executed multiple times based on a condition or a specific range. The bloc continue its execution as long as the [condition] remains True. It can also be noted as LOOP (min, max) to define exactly how many times the sequence repeats.

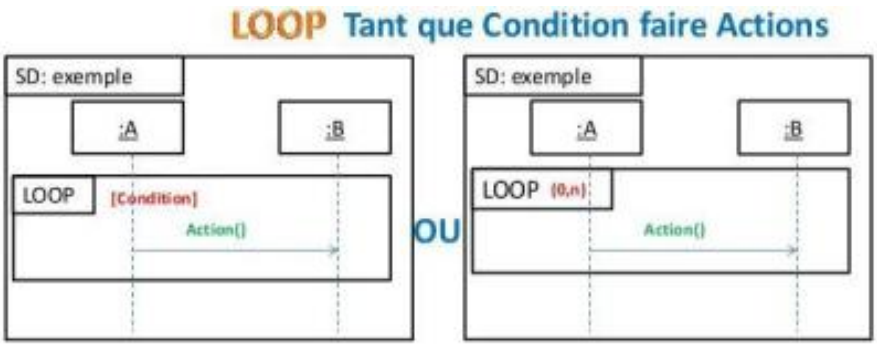

Figure 84. Representation of iterative fragment in a sequence diagram

- **Alternative fragment**

The Alt fragment models an "if-then-else" structure. The frame is split into two or more horizontal operands separated by a dashed line. If the first [condition] is met, ActionA() is executed. If that condition is false, the [else] path is taken, and ActionB() is performed instead. Only one of the paths can ever be executed.

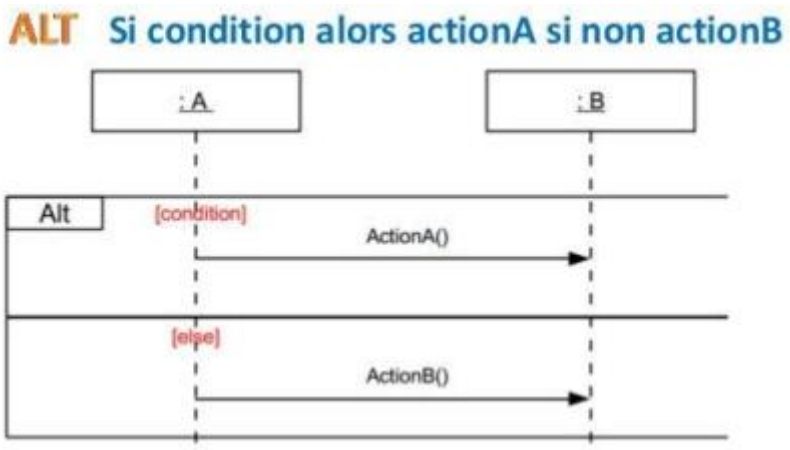

Figure 85. Representation of alternative fragment in a sequence diagram

- **Interruption fragment**

The Break fragment is a specialized scenario used to model an "interruption" or an early exit. If the [condition] is verified, the interactions inside the Break frame are performed, and then the rest of the enclosing fragment is skipped. This is often used to model exception handling or "cancel" operations.

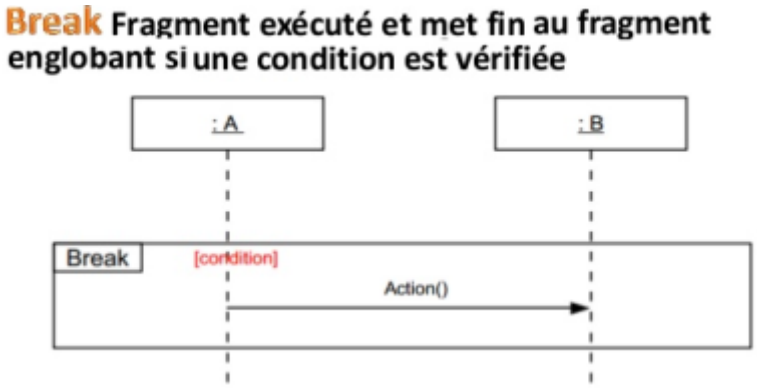

Figure 86. Representation of interruption fragment in a sequence diagram

An example of application of some of this fragment is the **sd Online_Bookshop** example. Here the the system uses a **loop** fragment to allow the **:Web Customer** to repeat searches within a single session. After receiving search results, two independent **opt** fragments

provide the flexibility to either view book description or add to shopping cart based on user preference. This demonstrates how combined fragments model iterative behavior and conditional logic without forcing a strictly linear execution.

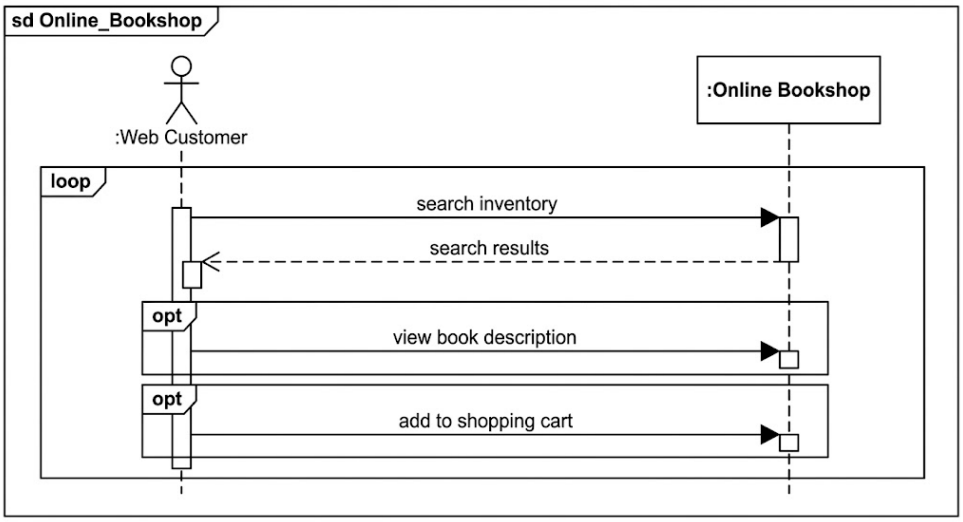

Figure 87. Example of fragment application in a sequence diagram