Sequence Diagram

Sequence diagrams, commonly used by developers, model the interactions between objects in a single use case. They illustrate how the different parts of a system interact with each other to carry out a function, and the order in which the interactions occur when a particular use case is executed.

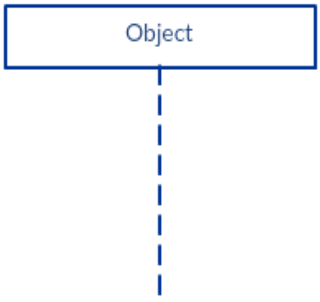
In simpler words, a sequence diagram shows different parts of a system work in a ‘sequence’ to get something done.

Sequence Diagram Notations

A sequence diagram is structured in such a way that it represents a timeline which begins at the top and descends gradually to mark the sequence of interactions. Each object has a column and the messages exchanged between them are represented by arrows.

A Quick Overview of the Various Parts of a Sequence Diagram

Lifeline Notation



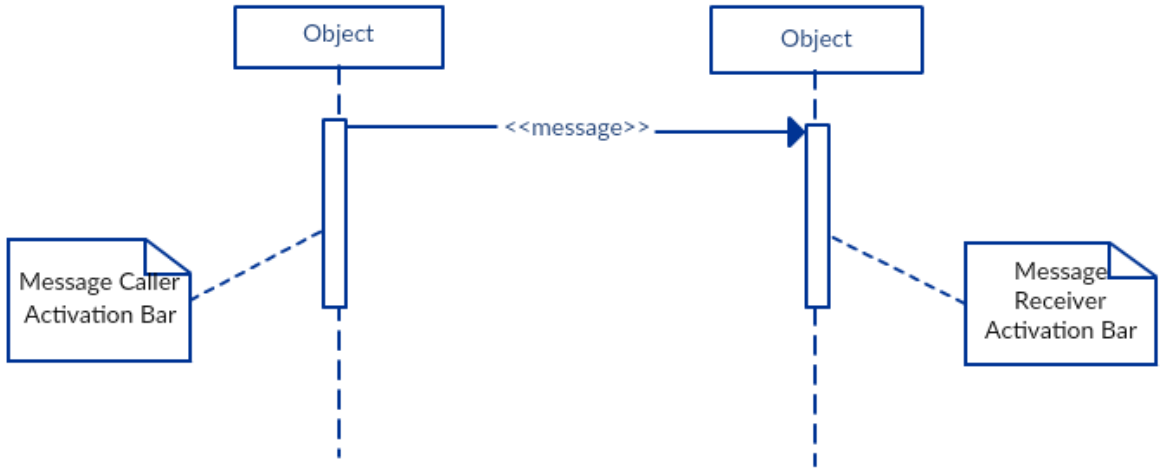
A sequence diagram is made up of several of these lifeline notations that should be arranged horizontally across the top of the diagram. No two lifeline notations should overlap each other. They represent the different objects or parts that interact with each other in the system during the sequence.

A lifeline notation with an actor element symbol is used when the particular sequence diagram is owned by a use case.

Activation Bars

Activation bar is the box placed on the lifeline. It is used to indicate that an object is active (or instantiated) during an interaction between two objects. The length of the rectangle indicates the duration of the objects staying active.

In a sequence diagram, an interaction between two objects occurs when one object sends a message to another. The use of the activation bar on the lifelines of the Message Caller (the object that sends the message) and the Message Receiver (the object that receives the message) indicates that both are active/is instantiated during the exchange of the message.



Message Arrows

An arrow from the Message Caller to the Message Receiver specifies a message in a sequence diagram. A message can flow in any direction; from left to right, right to left or back to the Message Caller itself. While you can describe the message being sent from one object to the other on the arrow, with different arrowheads you can indicate the type of message being sent or received.

The message arrow comes with a description, which is known as a message signature, on it. The format for this message signature is below. All parts except the message_name are optional.

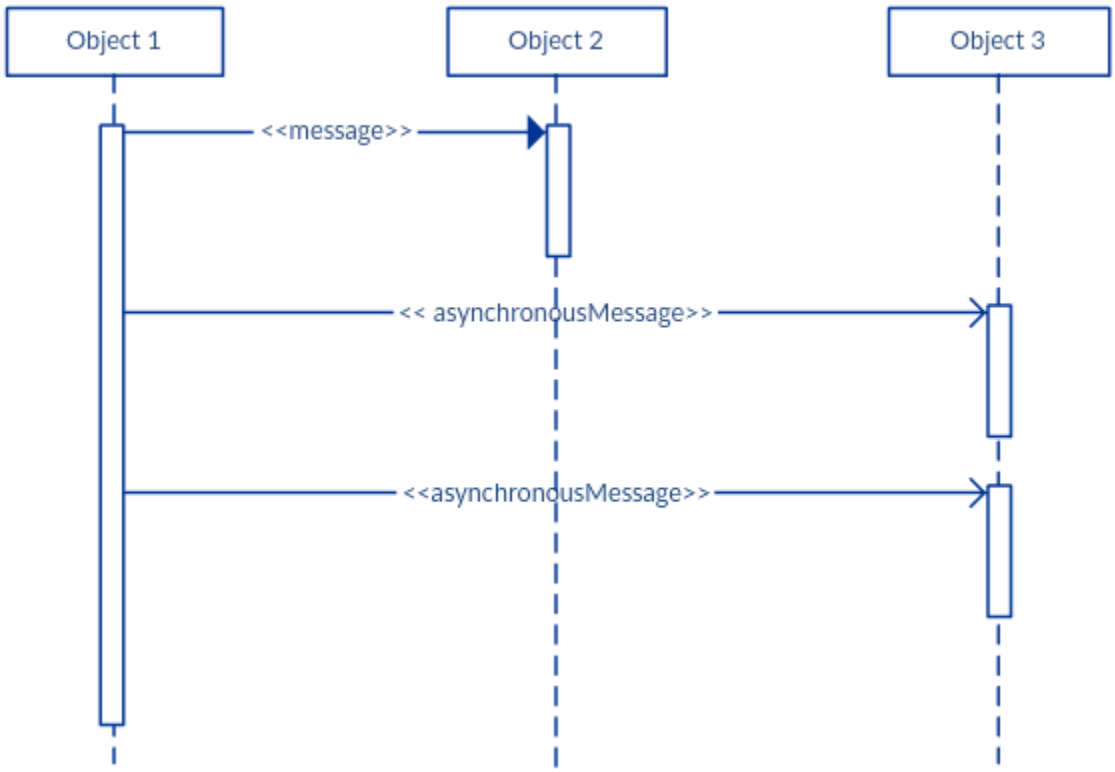
attribute = message_name (arguments): return_type

- *Synchronous message*

As shown in the activation bars example, a synchronous message is used when the sender waits for the receiver to process the message and return before carrying on with another message. The arrowhead used to indicate this type of message is a solid one, like the one below.

- *Asynchronous message*

An asynchronous message is used when the message caller does not wait for the receiver to process the message and return before sending other messages to other objects within the system. The arrowhead used to show this type of message is a line arrow like shown in the example below.

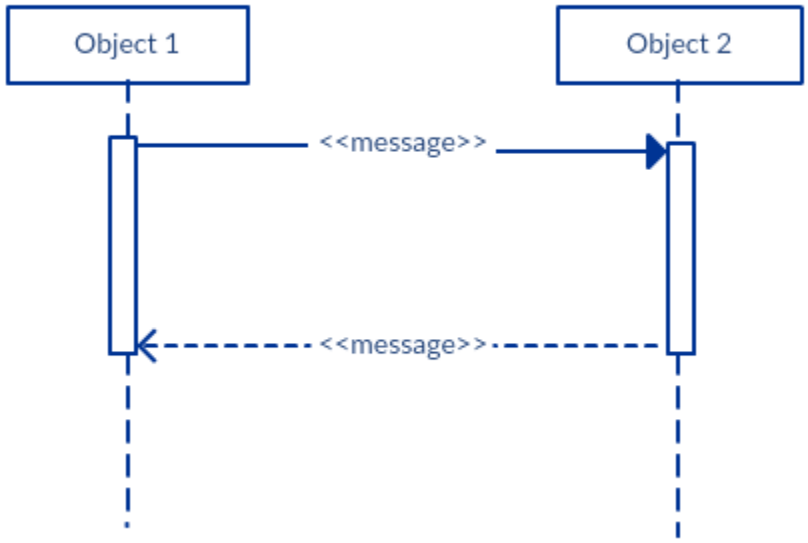


- *Return message*

A return message is used to indicate that the message receiver is done processing the message and is returning control over to the message caller.

Return messages are optional notation pieces, for an activation bar that is triggered by a synchronous message always implies a return message.

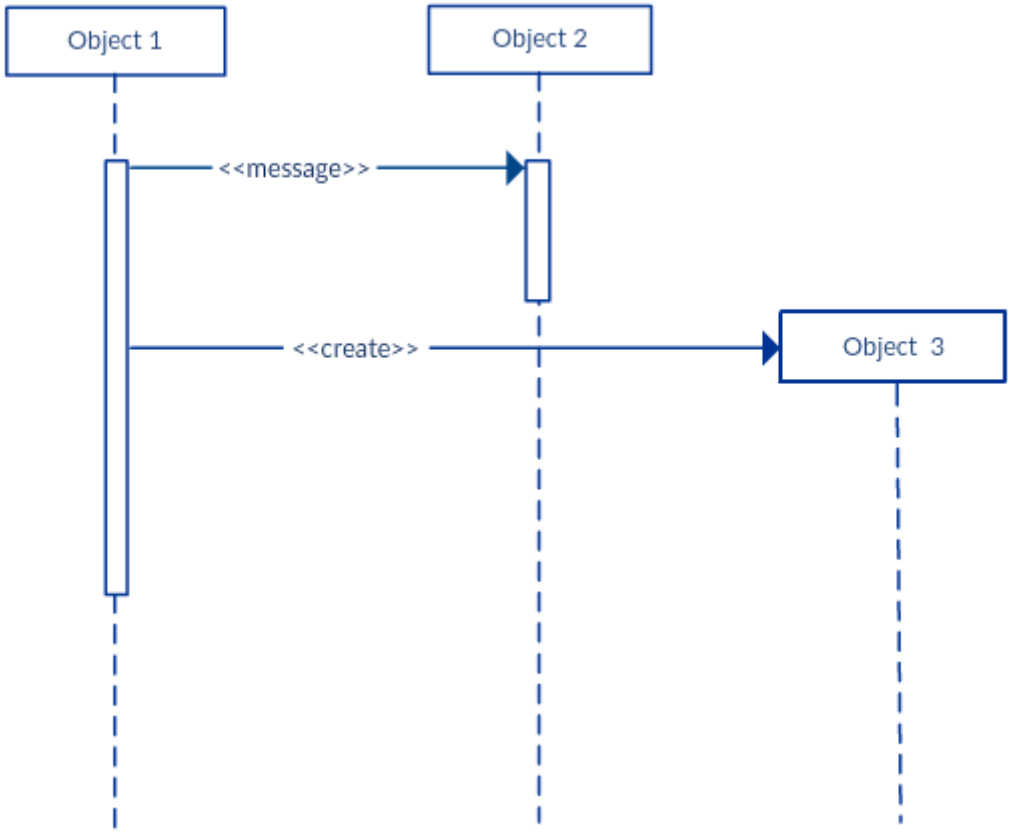
Tip: You can avoid cluttering up your diagrams by minimizing the use of return messages since the return value can be specified in the initial message arrow itself.



- *Participant creation message*

Objects do not necessarily live for the entire duration of the sequence of events. Objects or participants can be created according to the message that is being sent.

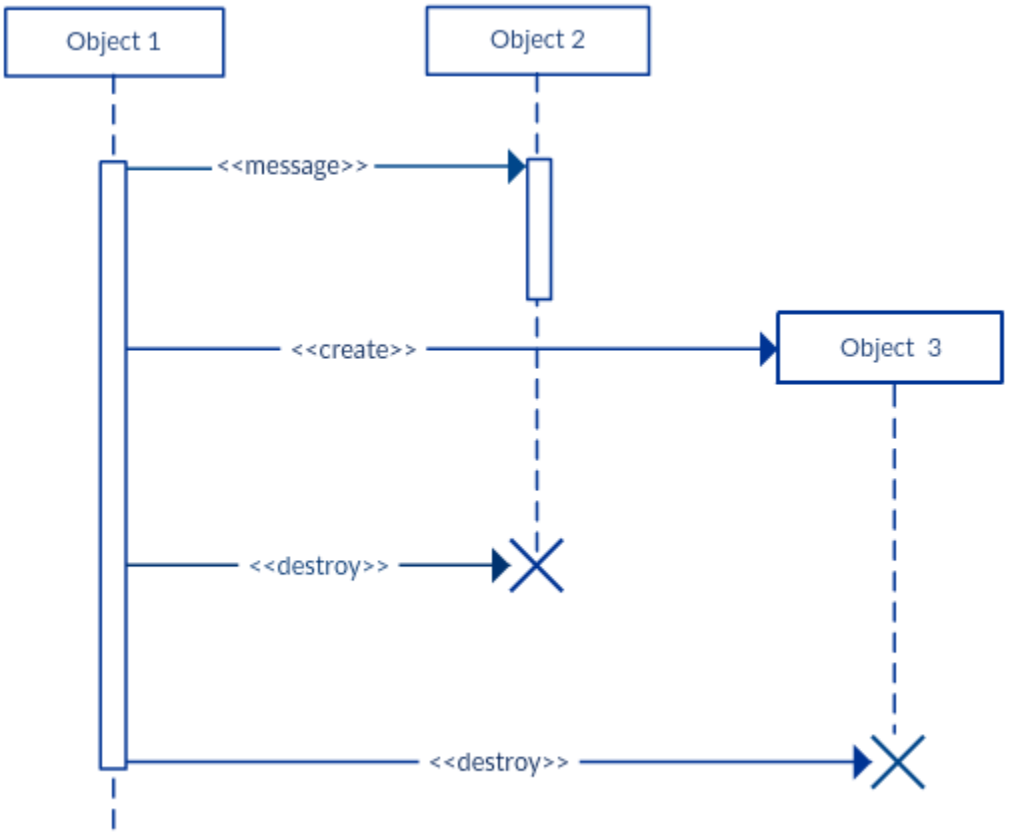
The dropped participant box notation can be used when you need to show that the particular participant did not exist until the create call was sent. If the created participant does something immediately after its creation, you should add an activation box right below the participant box.



- *Participant destruction message*

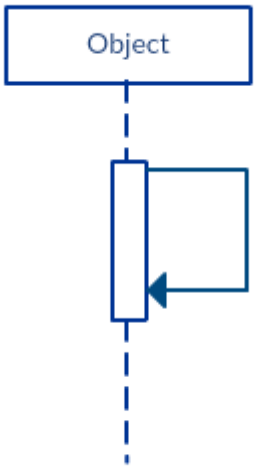
Likewise, participants when no longer needed can also be deleted from a sequence diagram. This is done by adding an ‘X’ at the end of the lifeline of

the said participant.



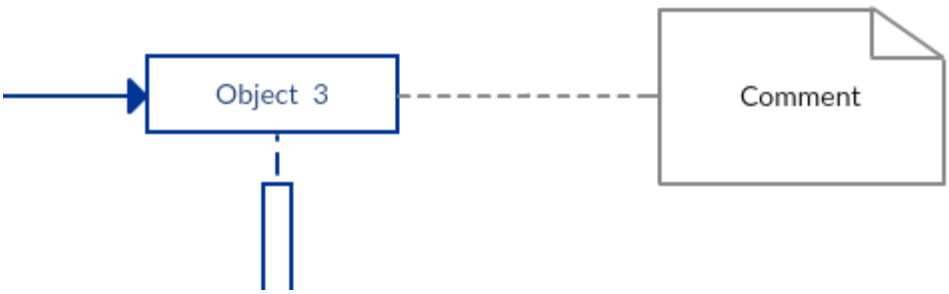
- *Reflexive message*

When an object sends a message to itself, it is called a reflexive message. It is indicated with a message arrow that starts and ends at the same lifeline as shown in the example below.



Comment

UML diagrams generally permit the annotation of comments in all UML diagram types. The comment object is a rectangle with a folded-over corner as shown below. The comment can be linked to the related object with a dashed line.

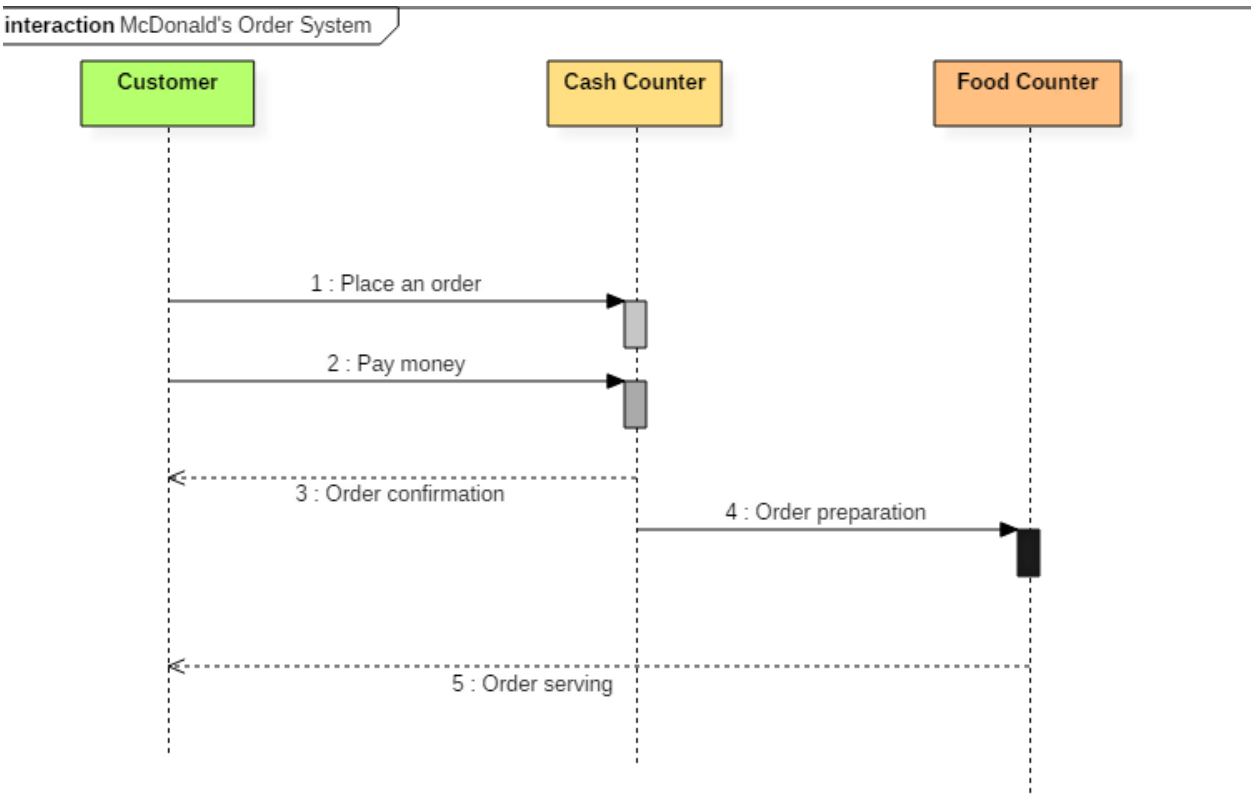


Note: View Sequence Diagram Best Practices to learn about sequence fragments.

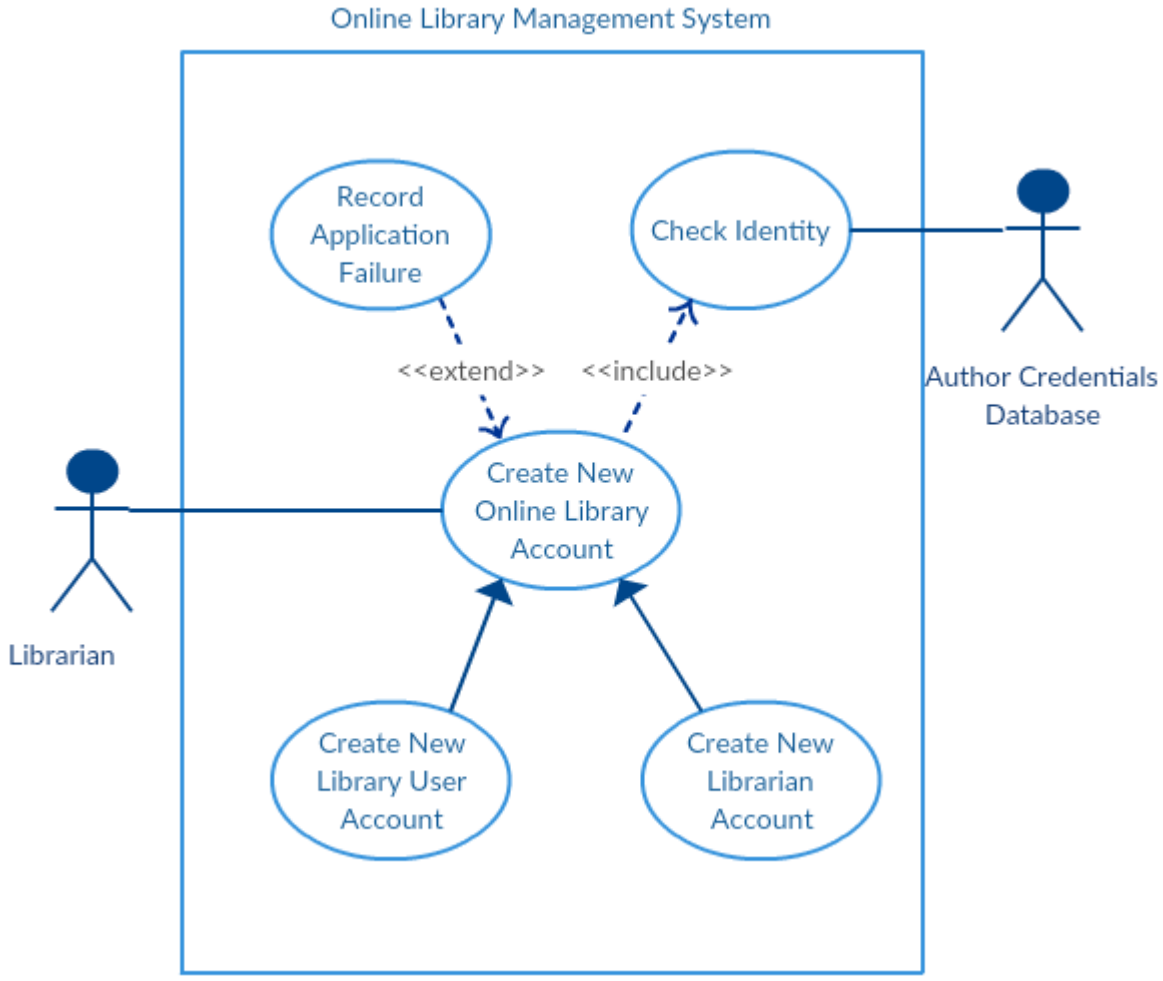
How to Draw a Sequence Diagram

A sequence diagram represents the scenario or flow of events in one single use case. The message flow of the sequence diagram is based on the narrative of the particular use case.

The following sequence diagram example represents McDonald's ordering system:



Before you start drawing the sequence diagram or decide what interactions should be included in it, you need to draw the use case diagram and ready a comprehensive description of what the particular use case does.



From the above use case diagram example of 'Create New Online Library Account', we will focus on the use case named 'Create New User Account' to draw our sequence diagram example.

Before drawing the sequence diagram, it's necessary to identify the objects or actors that would be involved in creating a new user account. These would be;

- Librarian
- Online Library Management system
- User credentials database
- Email system

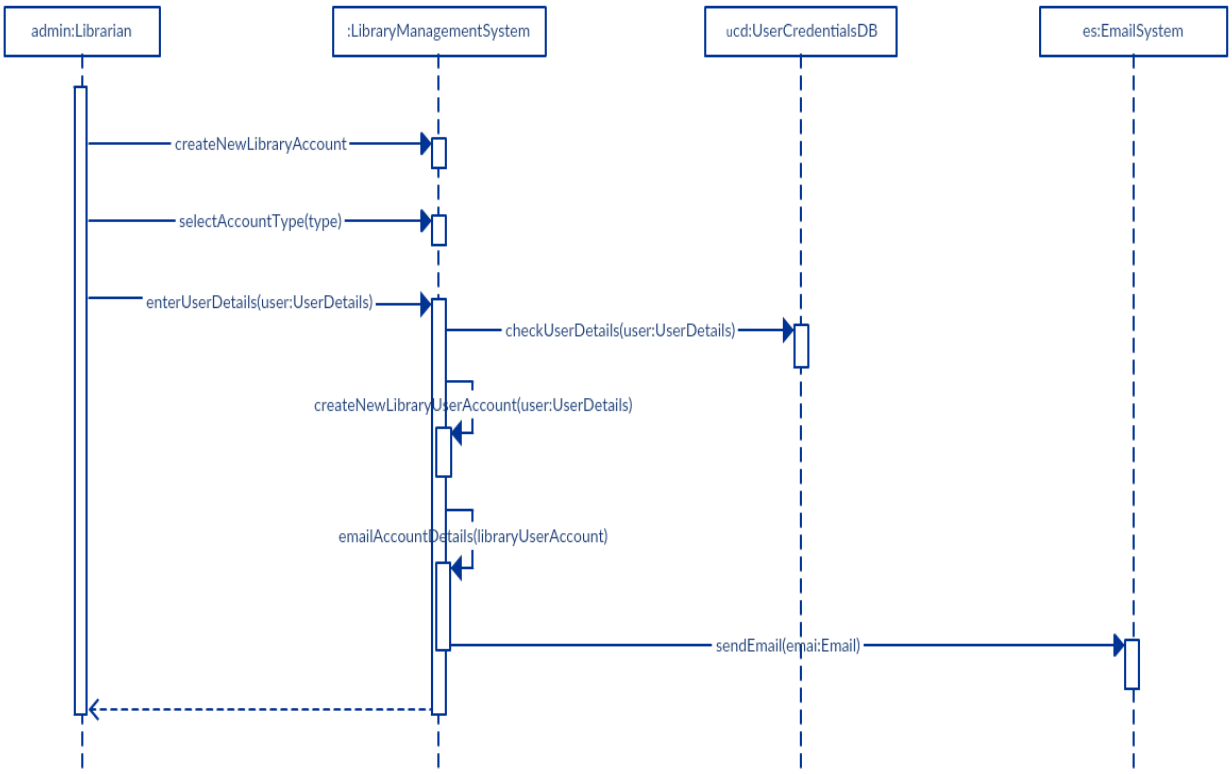
Once you identify the objects, it is then important to write a detailed description on what the use case does. From this description, you can easily figure out the interactions (that should go in the sequence diagram) that would occur between the objects above, once the use case is executed.

Here are the steps that occur in the use case named ‘Create New Library User Account’.

- The librarian requests the system to create a new online library account
- The librarian then selects the library user account type
- The librarian enters the user’s details
- The user’s details are checked using the user Credentials Database
- The new library user account is created
- A summary of the of the new account’s details are then emailed to the user

From each of these steps, you can easily specify what messages should be exchanged between the objects in the sequence diagram. Once it’s clear, you can go ahead and start drawing the sequence diagram.

The sequence diagram below shows how the objects in the online library management system interact with each other to perform the function ‘Create New Library User Account’.



Example: Place Order

The example shows a Sequence diagram with three participating objects: Customer, Order, and the Stock. Without even knowing the notation formally, you can probably get a pretty good idea of what is going on.

Step 1 and 2: Customer creates an order.

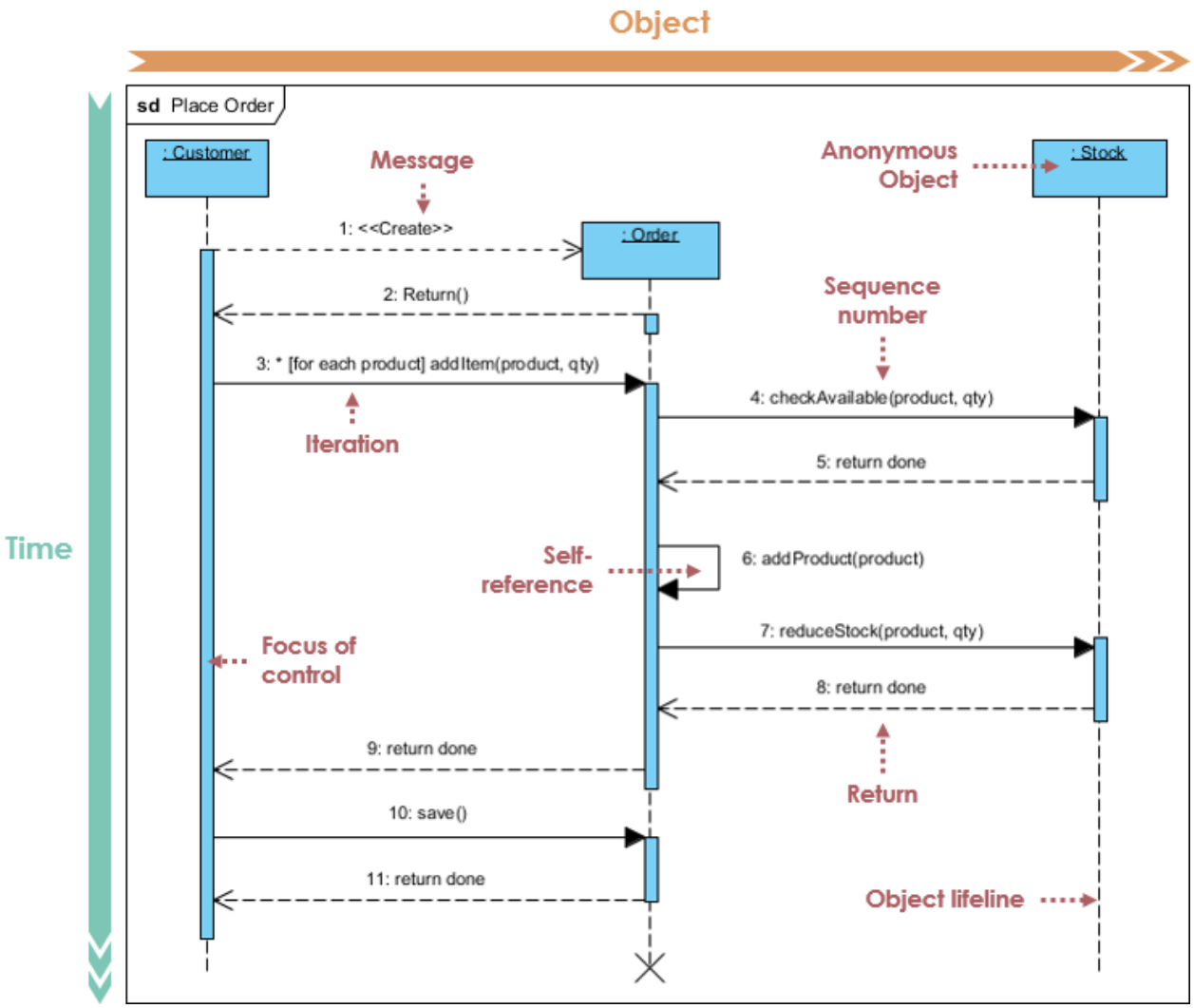
Step 3: Customer add items to the order.

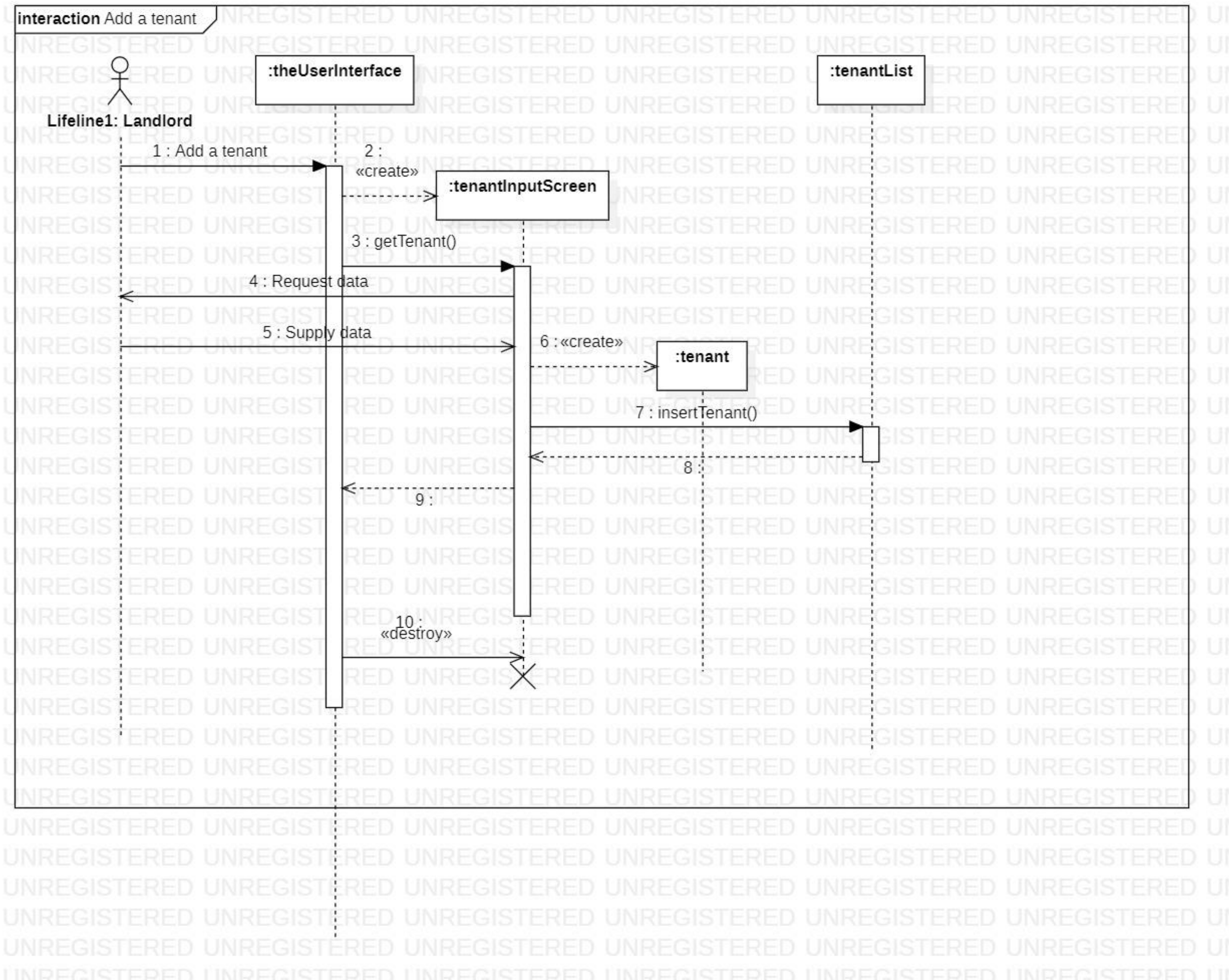
Step 4, 5: Each item is checked for availability in inventory.

Step 6, 7, 8: If the product is available, it is added to the order.

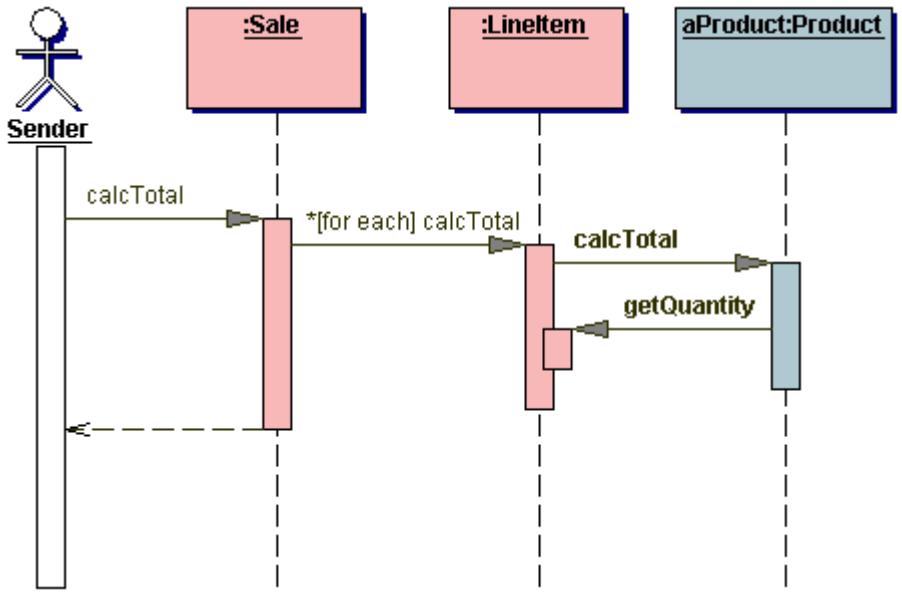
Step 9 return

Step 10, 11: save and destroy order


Object
Time
sd Place Order
: Customer
Message
Anonymous Object
: Stock
1: <<Create>>
: Order
2: Return()
Sequence number
3: * [for each product] addItem(product, qty)
4: checkAvailable(product, qty)
Iteration
5: return done
Self-reference
6: addProduct(product)
7: reduceStock(product, qty)
Focus of control
8: return done
9: return done
Return
10: save()
11: return done
Object lifeline


interaction Add a tenant
:theUserInterface
:tenantList
Lifeline1: Landlord
1 : Add a tenant
2 : «create»
:tenantInputScreen
3 : getTenant()
4 : Request data
5 : Supply data
6 : «create»
:tenant
7 : insertTenant()
8 :
9 :
10 : «destroy»

Coding from sequence diagram




```
/**
   * From the Sale class:
   * calculates the total of the sale from the lineItem subtotals
   * @return total of the sale
   */
  public double calcTotal() {
    total = 0.0;
    Iterator i = lineItems.iterator();
    while (i.hasNext()) total += ((LineItem)i.next()).calcTotal();
    return total;
  }

  /**

   * From the LineItem class:
   * calculates the cost of this amount of this kind of item
   * @return the cost of this amount of the item
   */
  public double calcTotal() {
    total = product.calcTotal(this);
    return total;
  }

  /**
   * From the Product class:
   * calculates the current cost of a quantity of the product
   * @return cost of the line item supplied
   */
  public double calcTotal(LineItem li) {
    return amount * li.getQuantity();
  }
```

Combined fragment

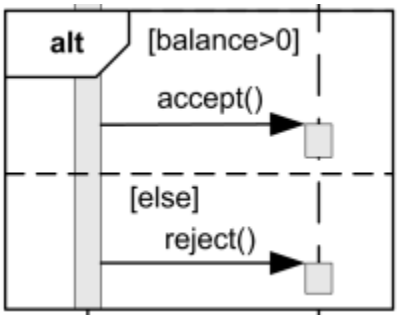
Combined fragment is an interaction fragment which defines a combination (expression) of interaction fragments. A combined fragment is defined by an interaction operator and corresponding interaction operands. Combined fragment may have interaction constraints also called guards in UML 2.4.

Interaction operator could be one of:

- **alt** - **alternatives**
- **opt** - **option**
- **loop** - **iteration**
- **break** - **break**
- **par** - **parallel**
- **strict** - **strict sequencing**
- **seq** - **weak sequencing**
- **critical** - **critical region**
- **ignore** - **ignore**
- **consider** - **consider**
- **assert** - **assertion**
- **neg** - **negative**

Alternatives

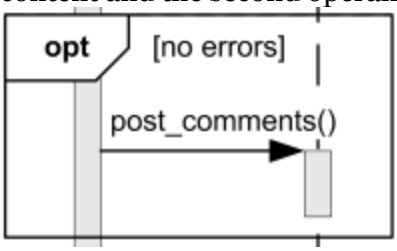
The interaction operator **alt** means that the combined fragment represents a **choice** or alternatives of behavior. At most one of the operands will be chosen. The chosen operand must have an explicit or implicit guard expression that evaluates to true at this point in the interaction.
An implicit true guard is implied if the operand has no guard.
An operand guarded by **else** means a guard that is the negation of the disjunction of all other guards. If none of the operands has a guard that evaluates to true, none of the operands are executed and the remainder of the enclosing interaction fragment is executed.



Call accept() if balance > 0, call reject() otherwise.

Option

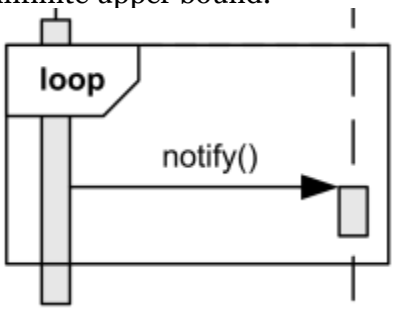
The interaction operator **opt** means that the combined fragment represents a **choice** of behavior where either the (sole) operand happens or nothing happens. An option is semantically equivalent to an alternative combined fragment where there is one operand with non-empty content and the second operand is empty.



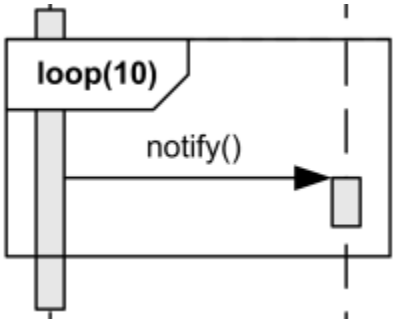
Post comments if there were no errors.

Loop

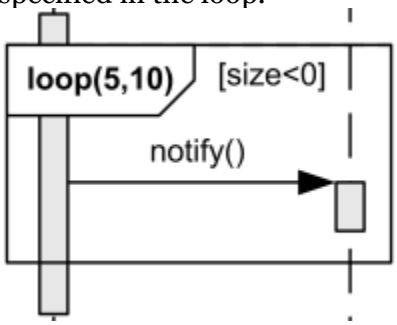
The interaction operator **loop** means that the combined fragment represents a loop. The loop operand will be repeated a number of times. The loop construct represents a recursive application of the **seq** operator where the loop operand is sequenced after the result of earlier iterations.
UML 2.4 specification provides weird description of the loop operator with odd examples. I will try to extract here some sense from that.
Loop could be controlled by either or both iteration bounds and a guard.
Loop operand could have iteration bounds which may include a lower and an upper number of iterations of the loop. Textual syntax of the loop is:
loop-operand ::= **loop** ['(' ***min-int*** [',' ***max-int***] ')']
min-int ::= ***non-negative-integer***
max-int ::= ***positive-integer*** | '*'
If loop has no bounds specified, it means potentially infinite loop with zero as lower bound and infinite upper bound.



Potentially infinite loop.
If only ***min-int*** is specified, it means that upper bound is equal to the lower bound, and loop will be executed exactly the specified number of times.



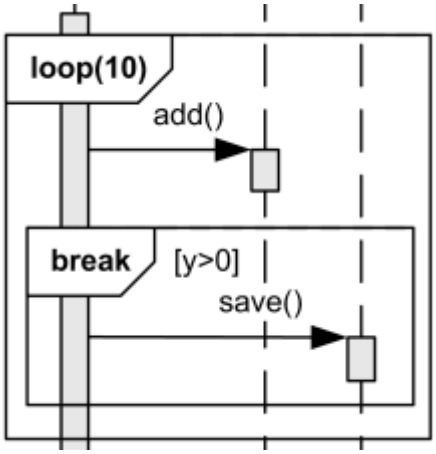
Loop to execute exactly 10 times.
If ***max-int*** is specified, it should be greater than or equal to ***min-int***. Loop will iterate minimum the ***min-int*** number of times and at most the ***max-int*** number of times.
Besides **iteration bounds** loop could also have an **interaction constraint** - a Boolean expression in square brackets. To add to the other confusions, UML 2.4 also calls both of them **guards**.
UML tries to shuffle the simplest form of **for loop** and **while loop** which causes weird UML 2.3 loop semantics on p.488 [UML 2.3 - Superstructure]: "after the minimum number of iterations have executed and the Boolean expression is false the loop will terminate". This is clarified - though with opposite meaning - on the next page as "the loop will only continue if that specification evaluates to true during execution regardless of the minimum number of iterations specified in the loop."



We may guess that as per UML 2.3, the loop is expected to execute minimum 5 times and no more than 10 times.
If guard condition [size<0] becomes false loop terminates regardless of the minimum number of iterations specified.
(Then why do we need that min number specified?!)

Break

The interaction operator **break** represents a **breaking** or exceptional scenario that is performed instead of the remainder of the enclosing interaction fragment.
A break operator with a **guard** is chosen when the guard is true. In this case the rest of the directly enclosing interaction fragment is ignored. When the guard of the break operand is false, the break operand is ignored and the rest of the enclosing interaction fragment proceeds.



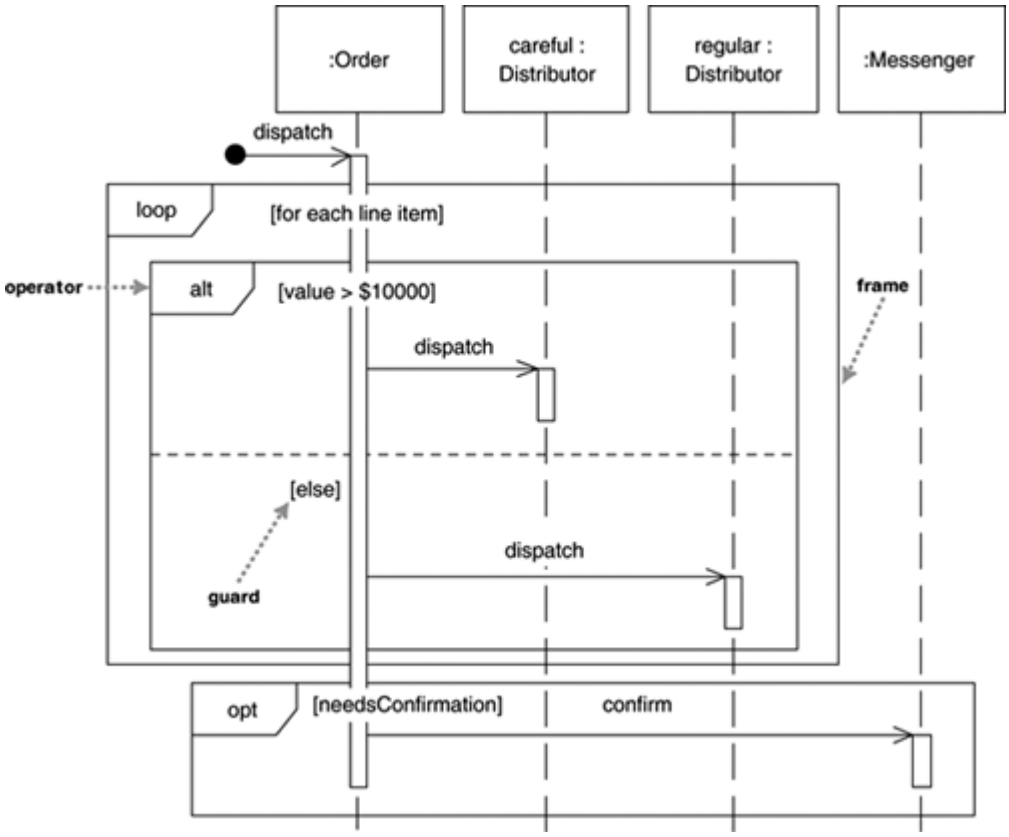
Break enclosing loop if y>0.

A combined fragment with the operator **break** should cover all lifelines of the enclosing interaction fragment.

Note, UML allows only one level - directly enclosing interaction fragment - to be abandoned. This could become really annoying if double loop or loop with other combined fragments should be broken.

UML 2.3 states that when break operand has **no guard**, the choice between the break operand and the rest of the enclosing interaction fragment is done "non-deterministically" which most likely means "unpredictable". Don't use break without guard.

```
procedure dispatch
  foreach (lineitem)
    if (product.value > $10K)
      careful.dispatch
    else
      regular.dispatch
    end if
  end for
  if (needsConfirmation) messenger.confirm
end procedure
```



A sequence diagram that references two different sequence diagrams