interaction diagrams

Interaction Diagram:

1. Interaction diagram consisting of arcs and set of objects their relationships including messages that pass from object to another object.
2. Addresses the dynamic view
3. It has specialized types

Sequence Diagram

Collaboration Diagram

timing diagram

interaction overview diagram

Interaction Diagram:

UML provides four types of interaction diagrams useful in advanced modeling.

■■ The ***sequence diagram*** shows objects interacting along lifelines, showing the general order of this interaction.

■■ The ***timing diagram*** shows interactions with a precise time axis.

Interaction Diagram:

■■The ***interaction overview diagram*** allows a modeler to look at the interactions from a high level. These interactions are viewed on something like an activity diagram with many of the detailed elements not shown.

■■ The ***communication diagram***
A communication diagram, formerly called a **collaboration diagram** shows the messages exchanged between objects, focusing on the internal structure of the objects.

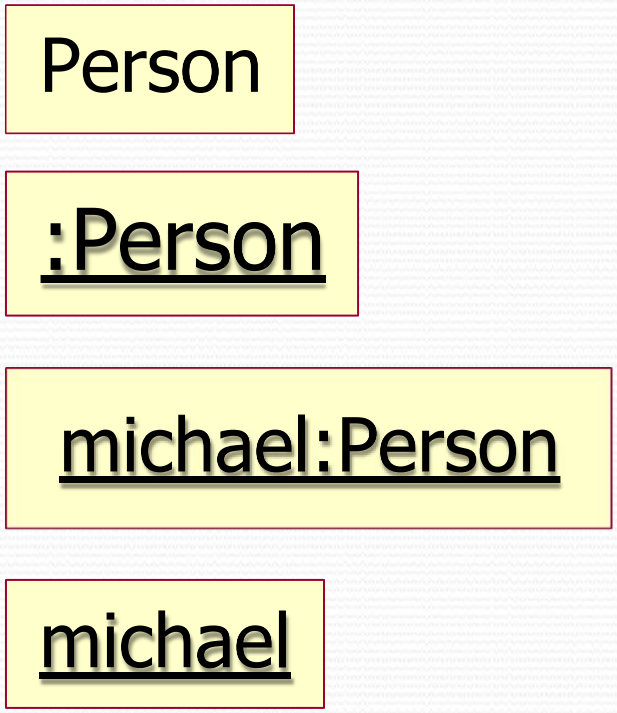
Collaboration and sequence - differences

- Collaboration diagram illustrates object interactions in a graph or network format, in which objects can be placed anywhere on the diagram. It demonstrates how objects are statically connected.

- Sequence diagram illustrates interaction in a kind of fence format, in which each new object is added to the right. It generally shows the sequence of events that occur.

Interaction diagrams: notation

- The following notation is used in the UML for classes and objects:
  - Class
  - Instance of a Class (object without a name)
  - Named instance of a class (named object)
  - Named object only (shown without class)



Sequence diagram

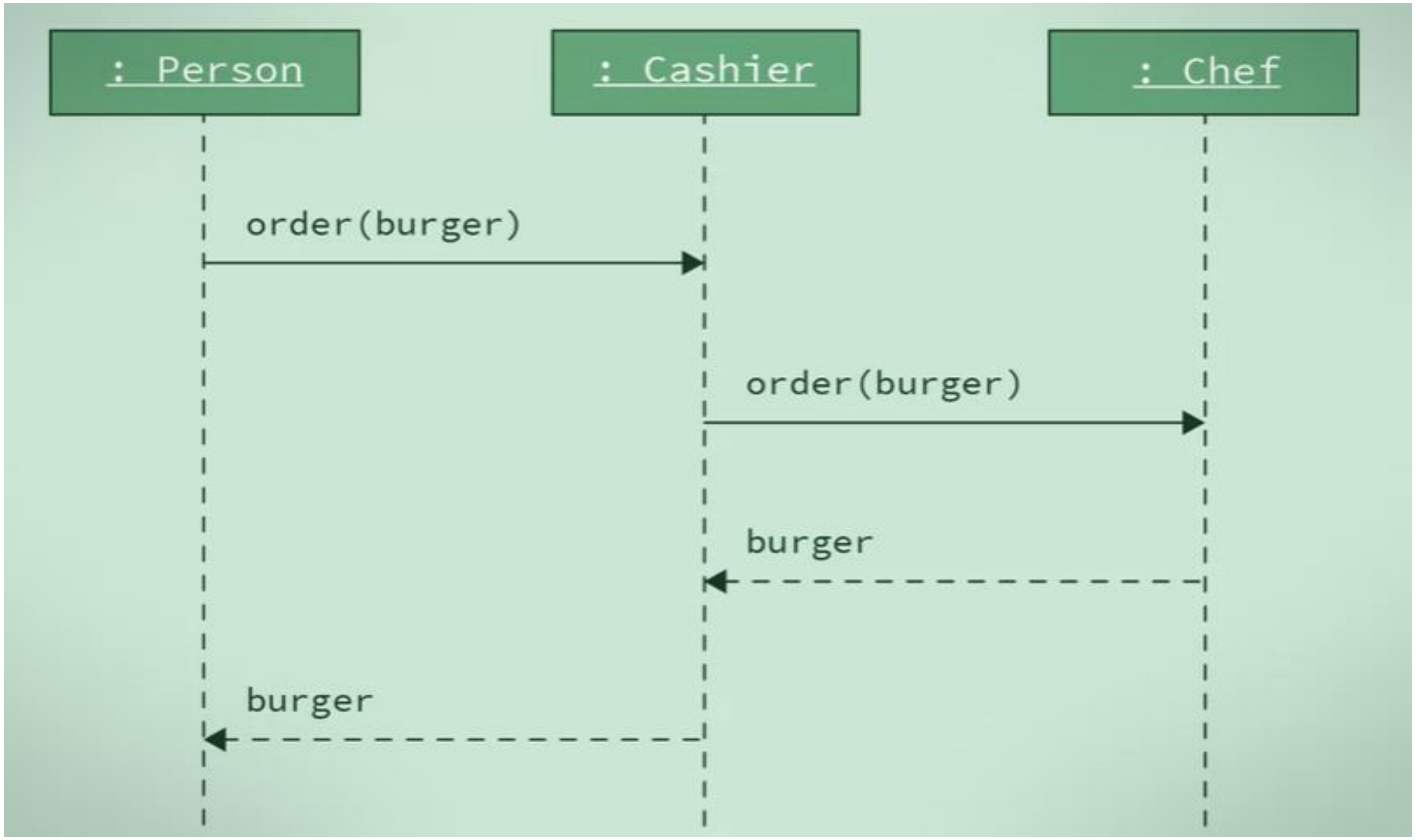
Sequence Diagrams are used to show your team how objects in your program interact with each other to complete tasks.

Simply put, think of a sequence diagram like a map of conversations between different people, where this map follows all the messages sent from person to person.

Sequence diagram

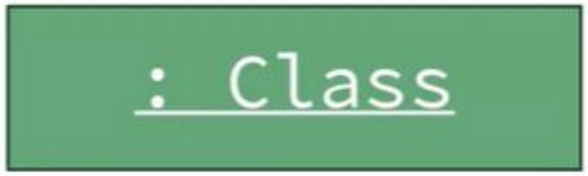
To fully understand it,
you should have a good grasp of objects and basic UML class diagrams.
Knowing how to break down a system into classes is essential in creating meaningful sequence diagrams.
A sequence diagram describes how objects in your system interact to complete a specific task.
Let's say that a person wants to order a burger at a local fast food restaurant

Example


: Person
: Cashier
: Chef
order(burger)
order(burger)
burger
burger

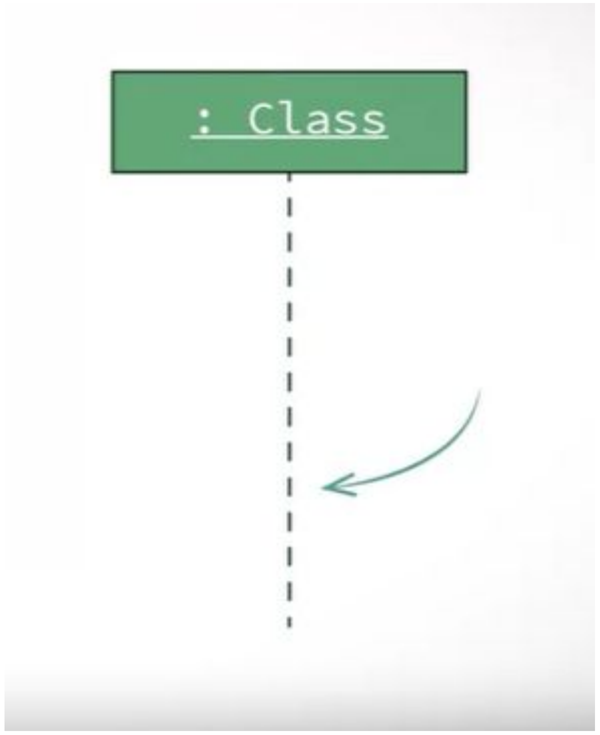
object

When creating sequence diagrams, first you use a box to represent role play by an object.
The role is typically labeled by the name of the class for the object.
each new object is added to the right draw objects from left to right in the sequence



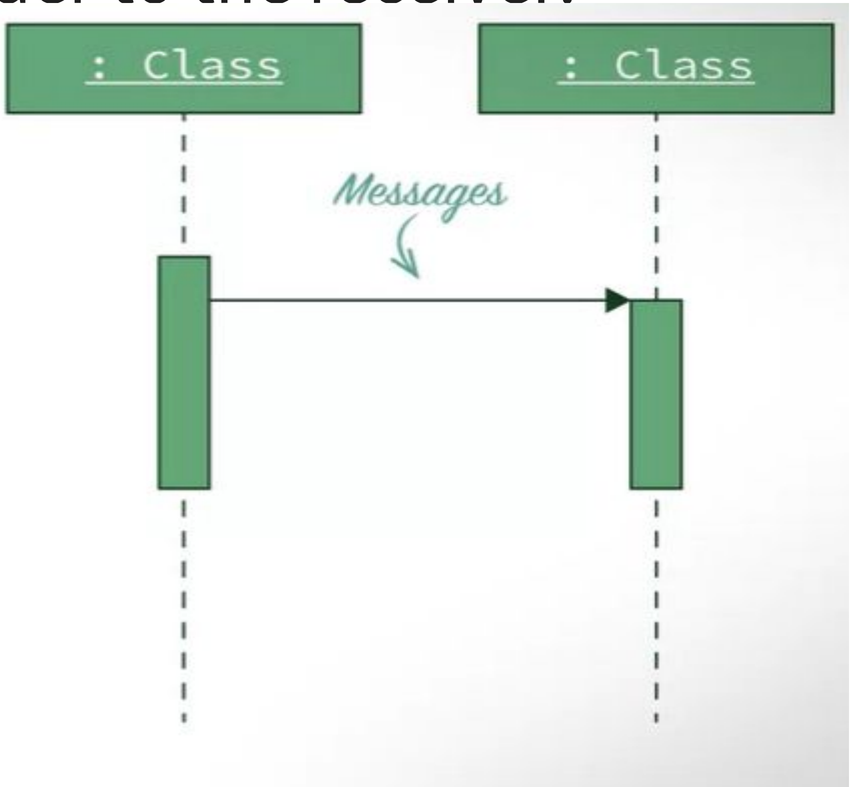
lifelines

Second, you use vertical dotted lines, known as lifelines,
to represent an object as time passes by.



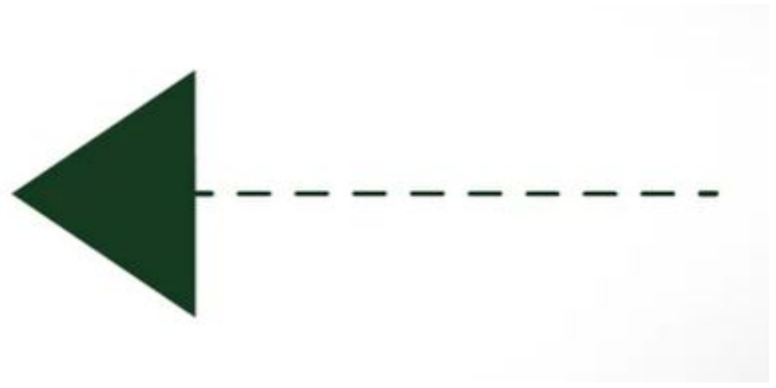
Messages

you use arrows to show messages that are sent from one object to another.
if one object sends a message to another object or objects, we denote this by drawing a solid line arrow from the sender to the receiver.



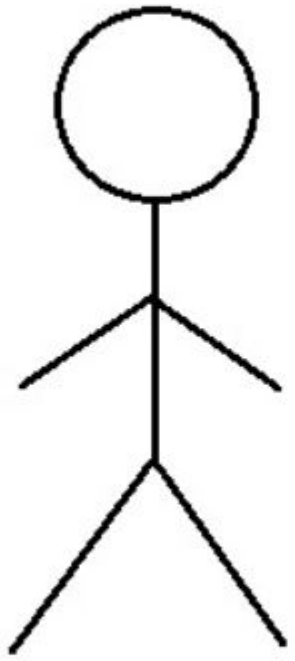
Acknowledgement

To return data and to control back to initiating objects, we would use a dotted line arrow.



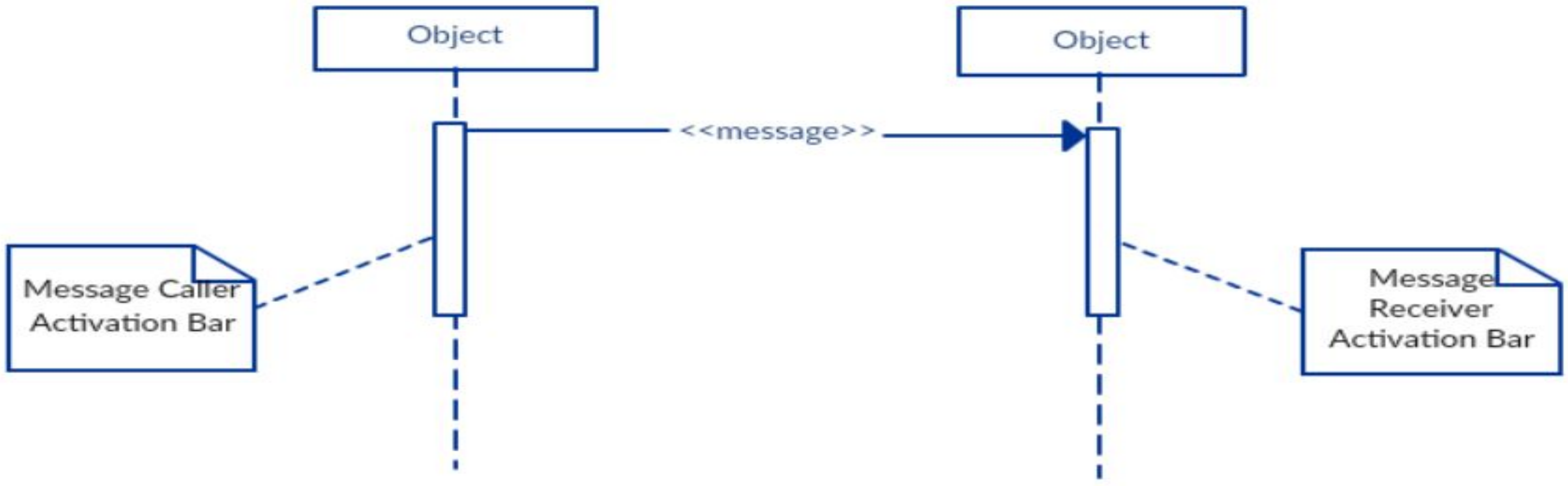
Actor

If there are people(external objects) in your example, who will be using or interacting with objects,
this are typically draw on a stick figures.



Activation Bars

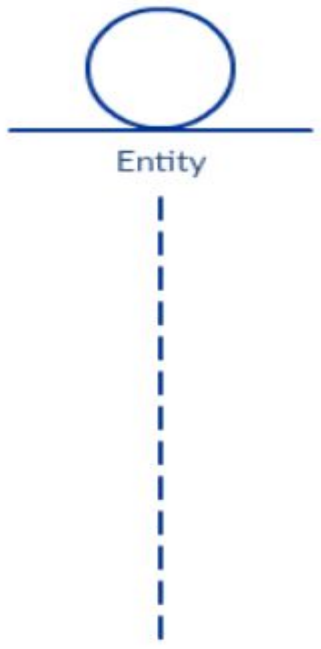
When an object is activated, we denote this on our sequence diagram using small rectangles on the objects lifeline. The length of the rectangle indicates the duration of the objects staying active.
You activate an object whenever an object sends, receives or it's waiting for a message.



Object
Object
<<message>>
Message Caller Activation Bar
Message Receiver Activation Bar

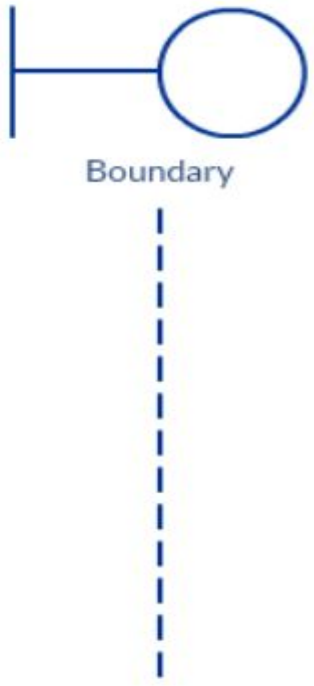
entity element

A lifeline with an entity element represents system data. For example, in a customer service application, the Customer entity would manage all data related to a customer.



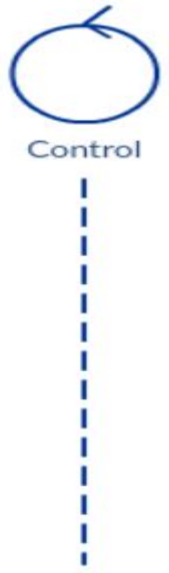
boundary element

A lifeline with a boundary element indicates a system boundary/ software element in a system; for example, user interface screens, database gateways or menus that users interact with, are boundaries.



control element

And a lifeline with a control element indicates a controlling entity or manager. It organizes and schedules the interactions between the boundaries and entities and serves as the mediator between them.

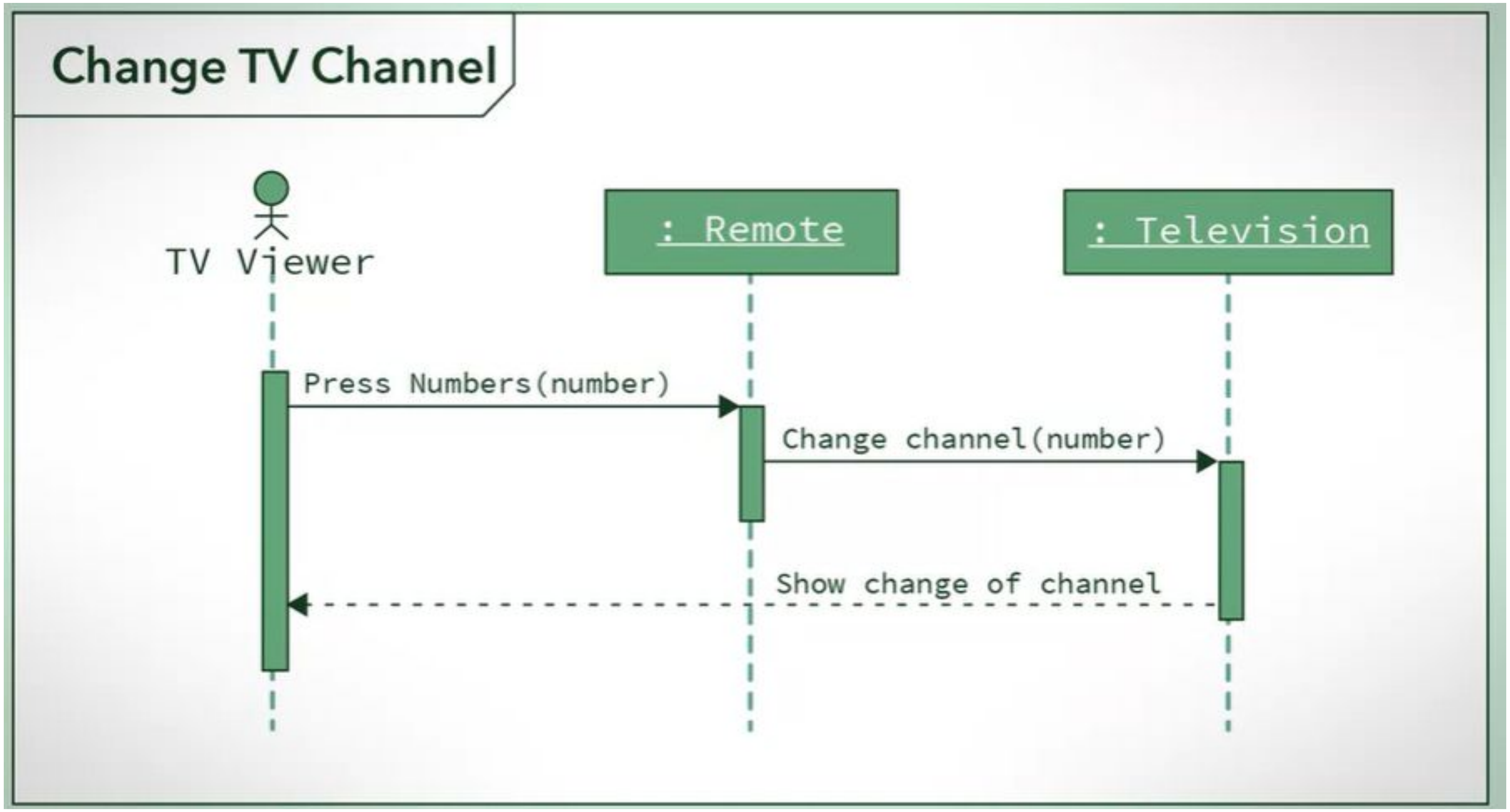


example

Let's use changing the channel of your television using a remote control.

What objects are important in this example?

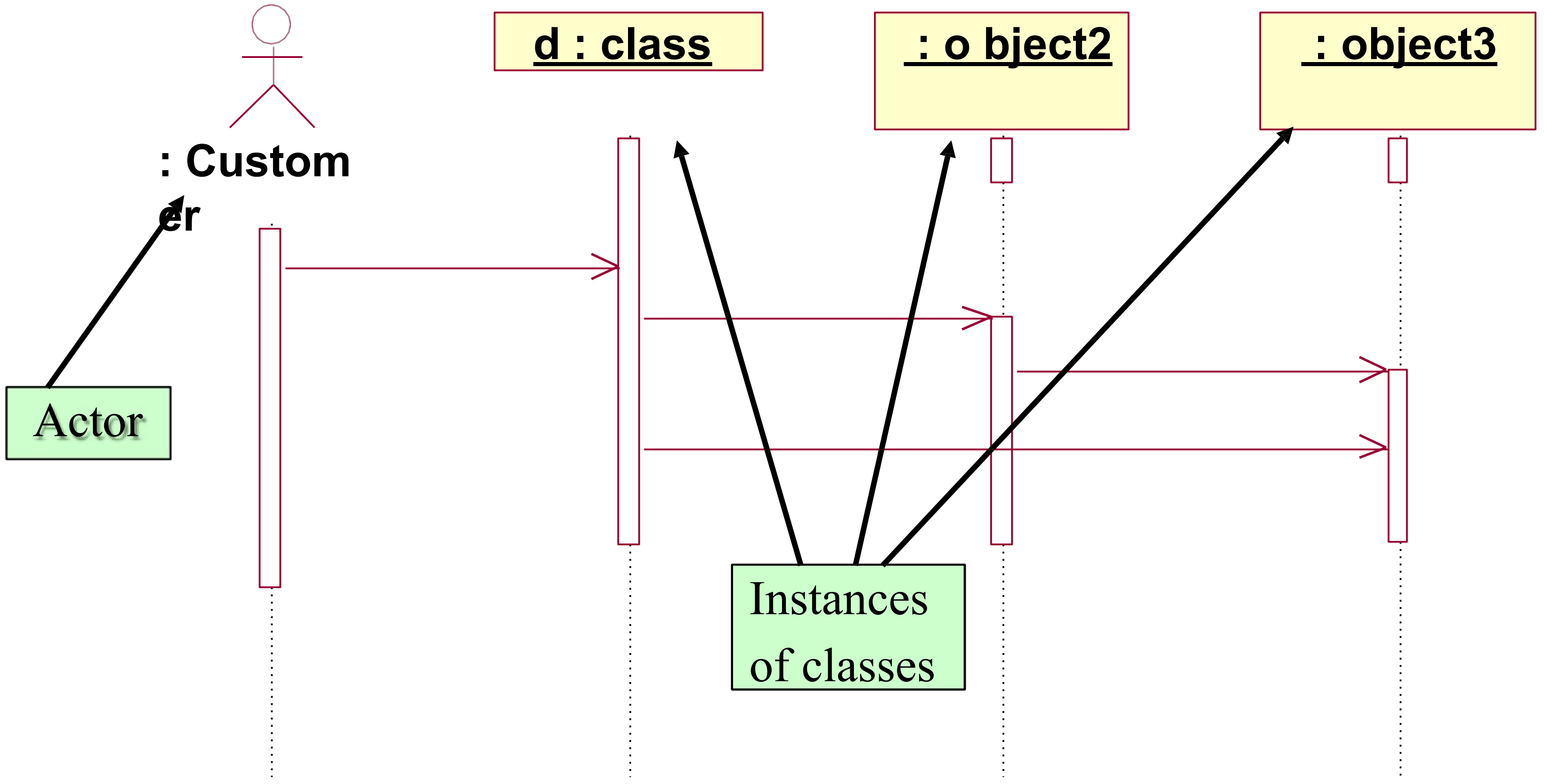
example


Change TV Channel
TV Viewer
: Remote
: Television
Press Numbers(number)
Change channel(number)
Show change of channel

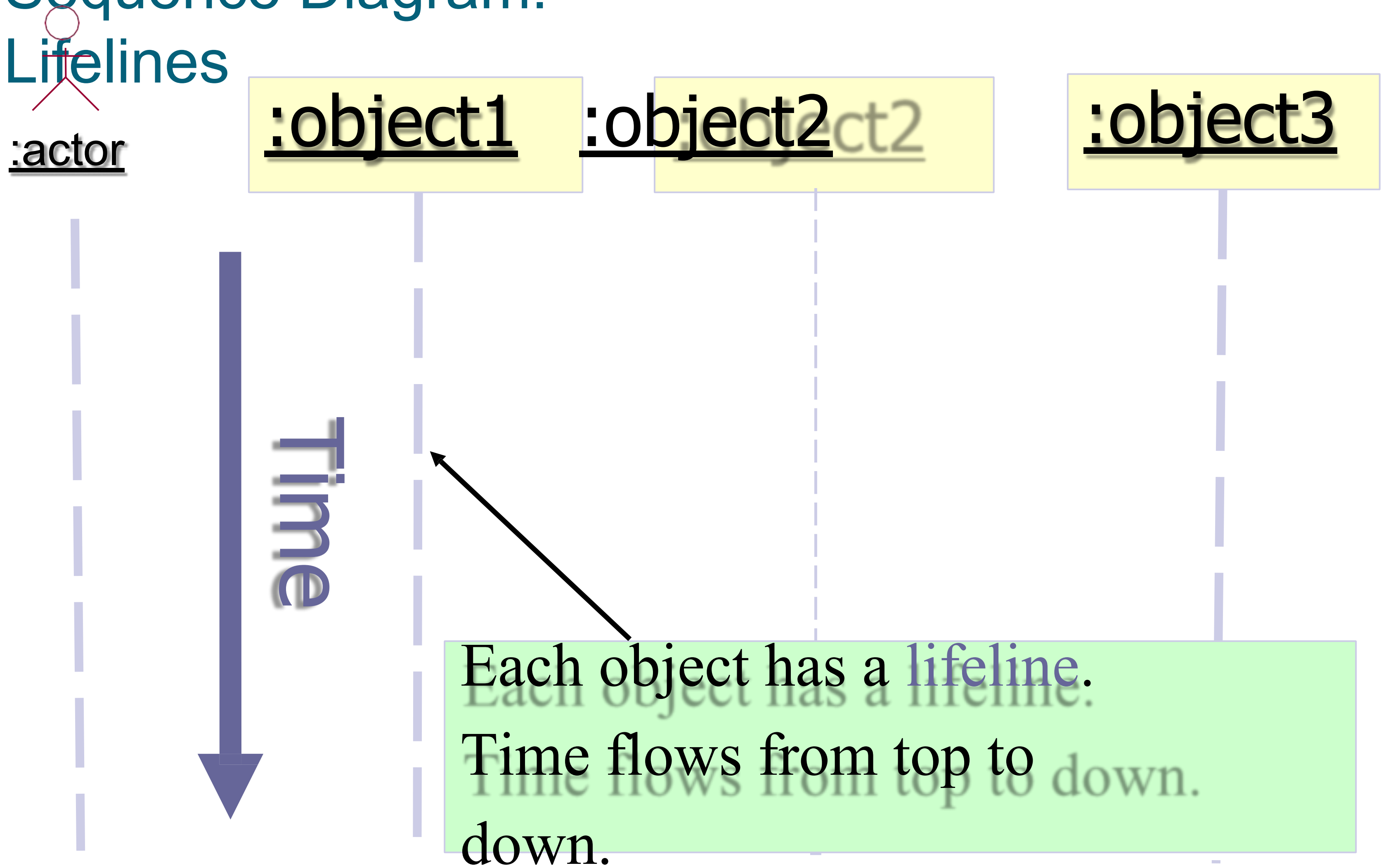
Sequence diagrams

- A sequence diagram shows relations between objects.
- It should be read from left to right and from top to bottom.
- At the top of the diagram are names of objects that interact with each other. These are the concepts in the conceptual model.
- When the course of events is initiated by an actor the actor symbol is displayed as the leftmost object.

Sequence diagrams: an example

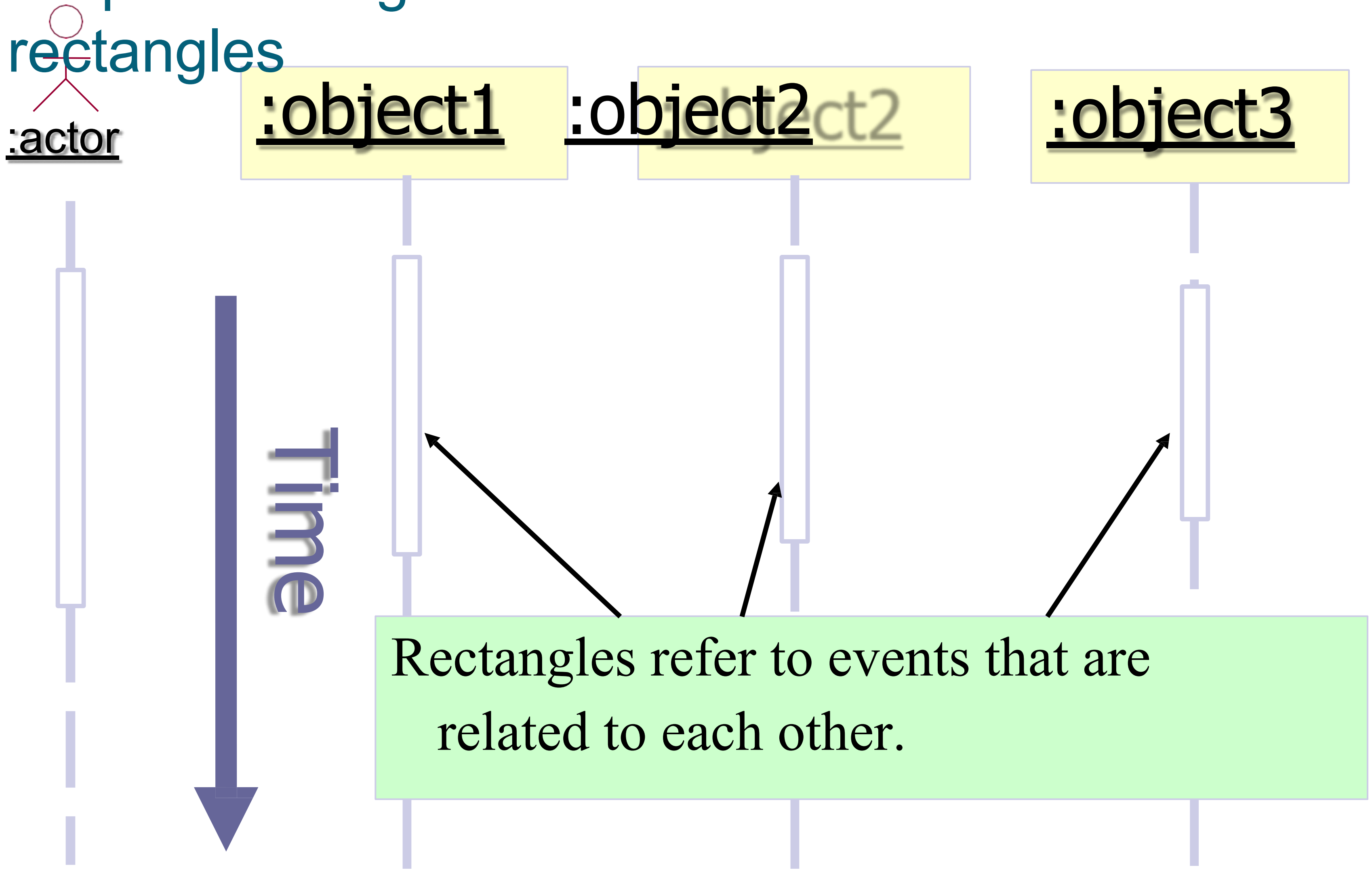


Sequence Diagram: Lifelines



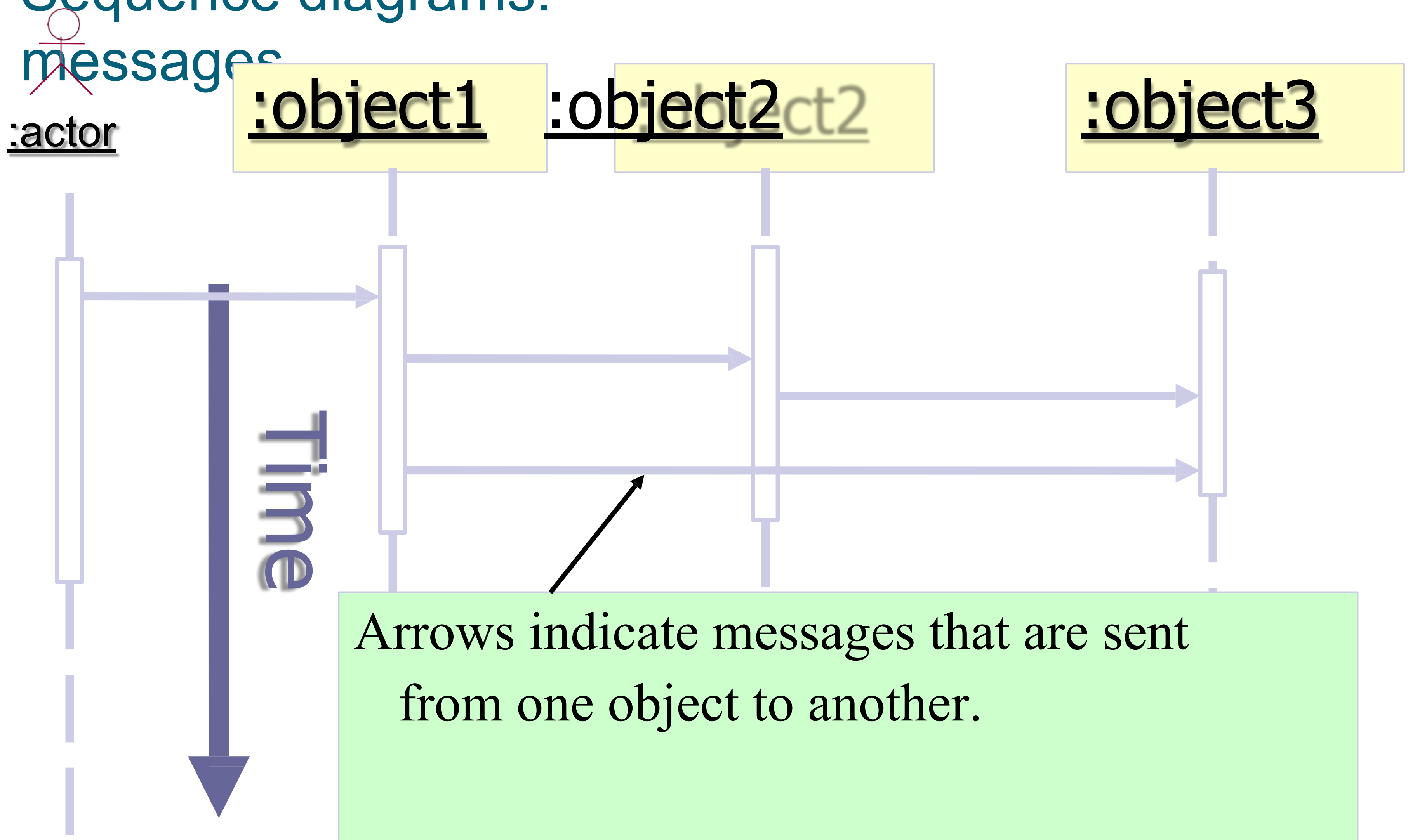
Each object has a lifeline.
Time flows from top to down.

Sequence diagrams: rectangles



Rectangles refer to events that are related to each other.

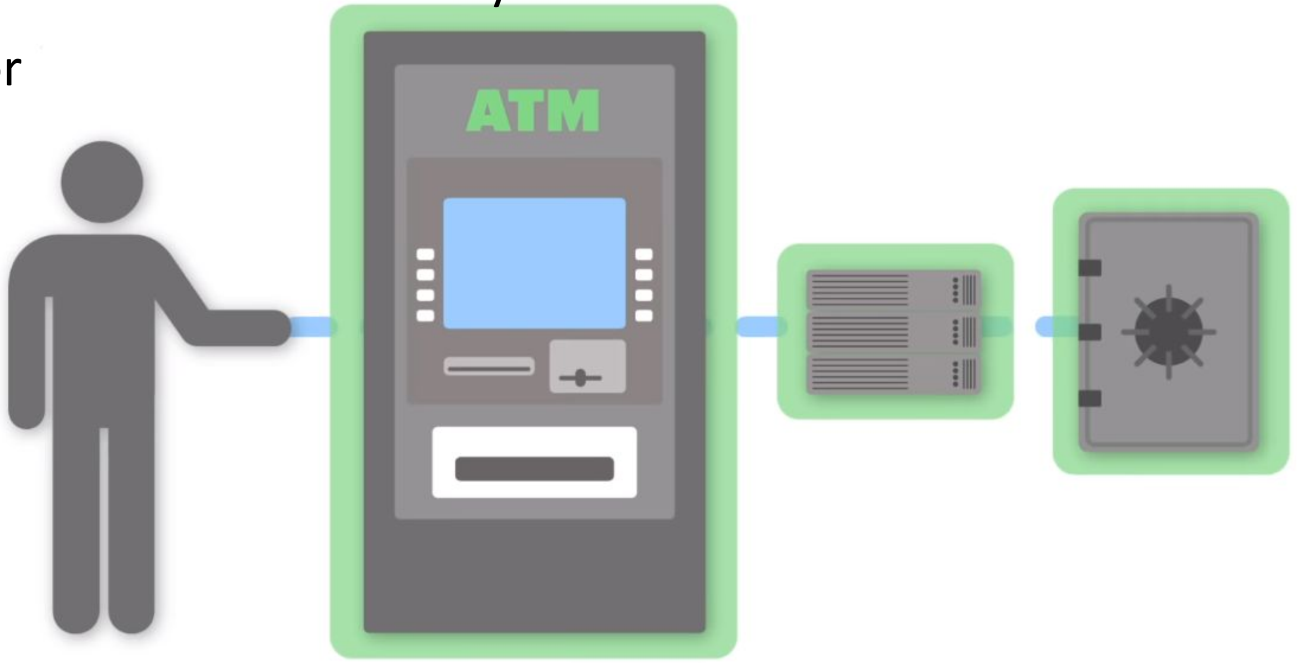
Sequence diagrams: messages



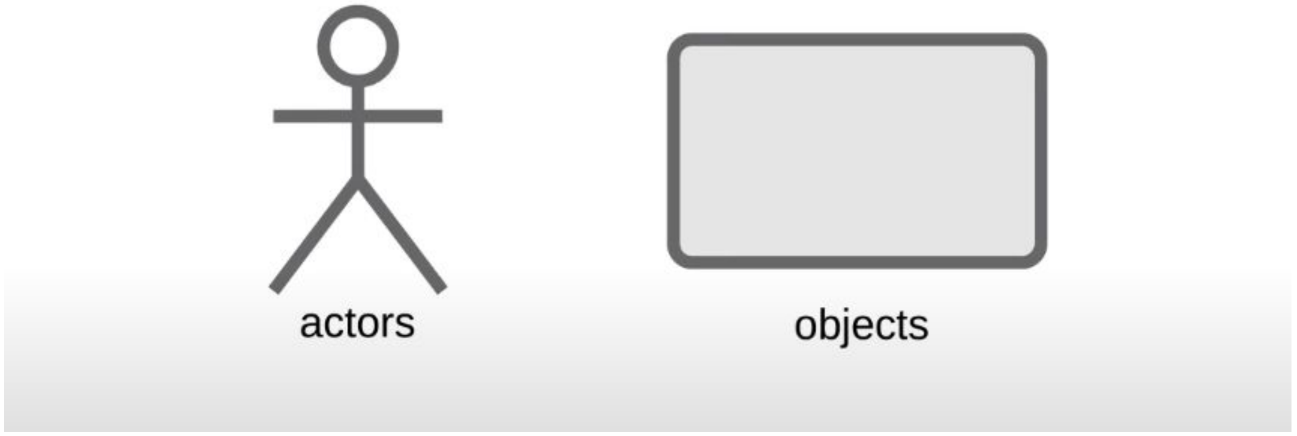
Arrows indicate messages that are sent from one object to another.

Example: ATM System

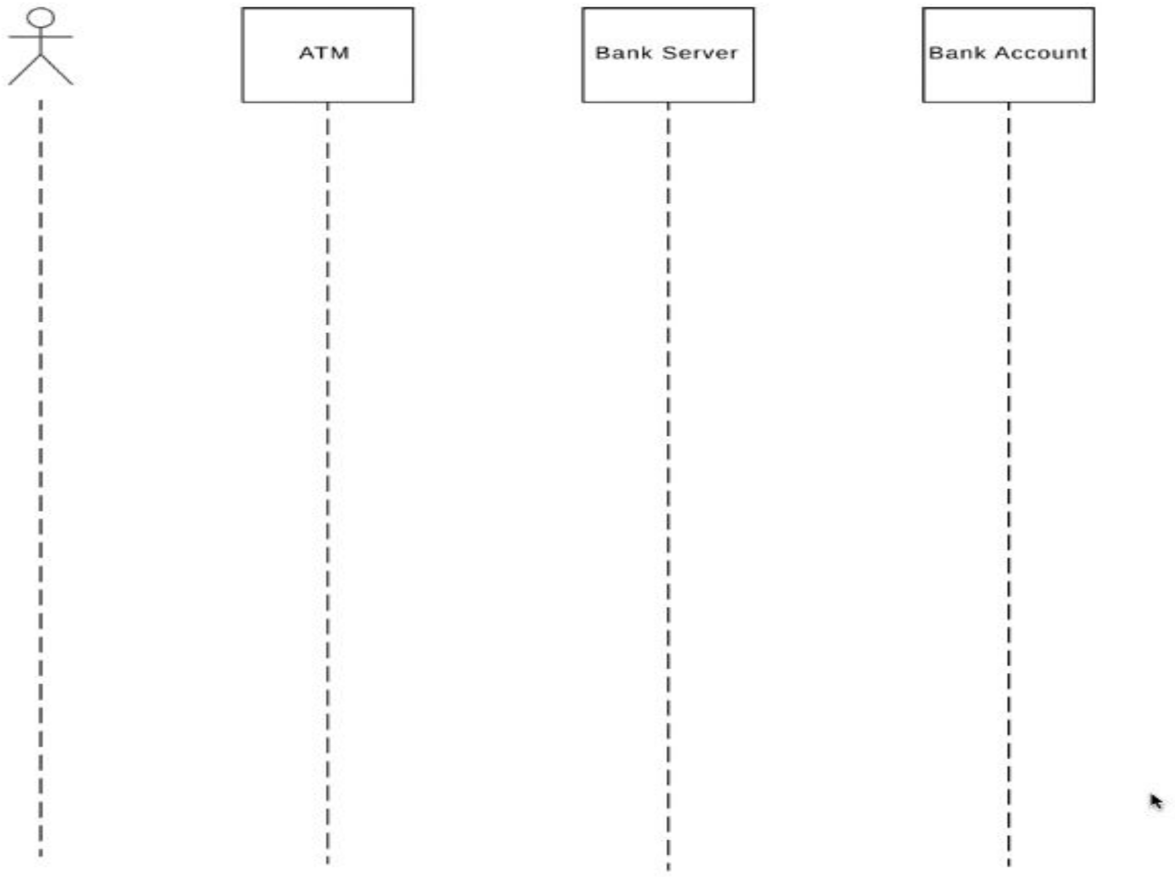
- Parts of ATM
- 3 Objects: ATM, Bank Server, Bank Account
- Person is external to the system
- Actor



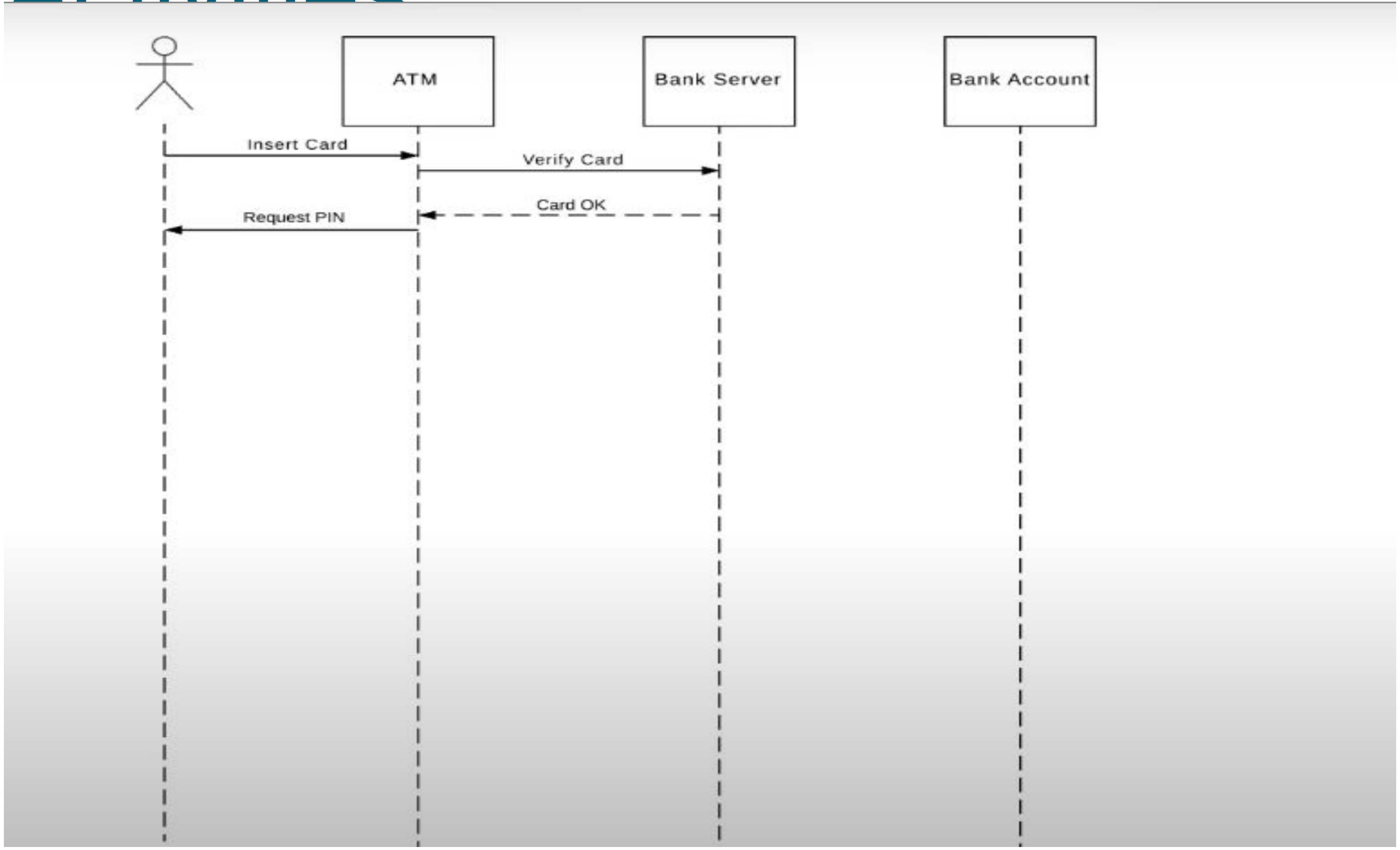
Representation



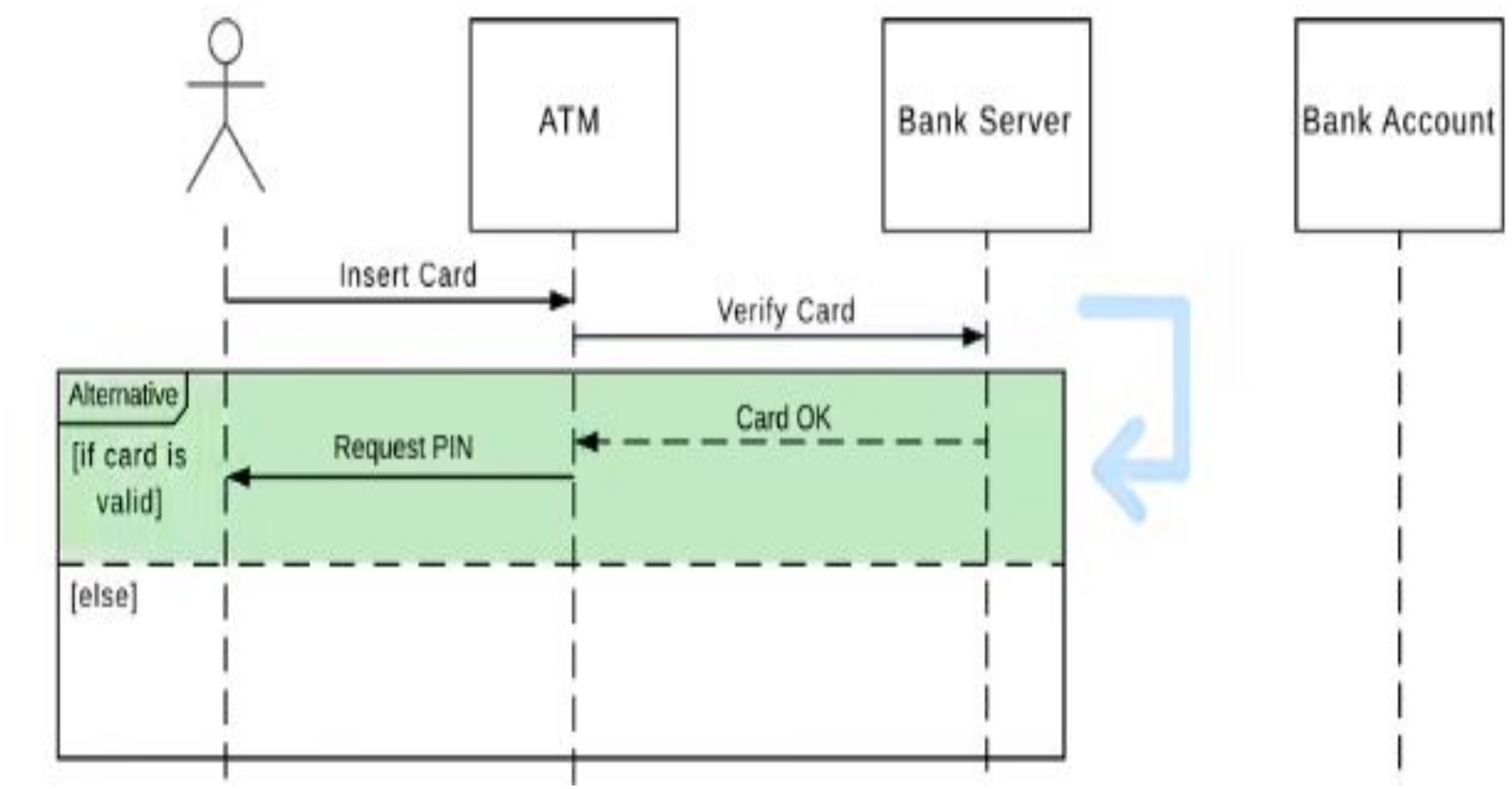
Life line

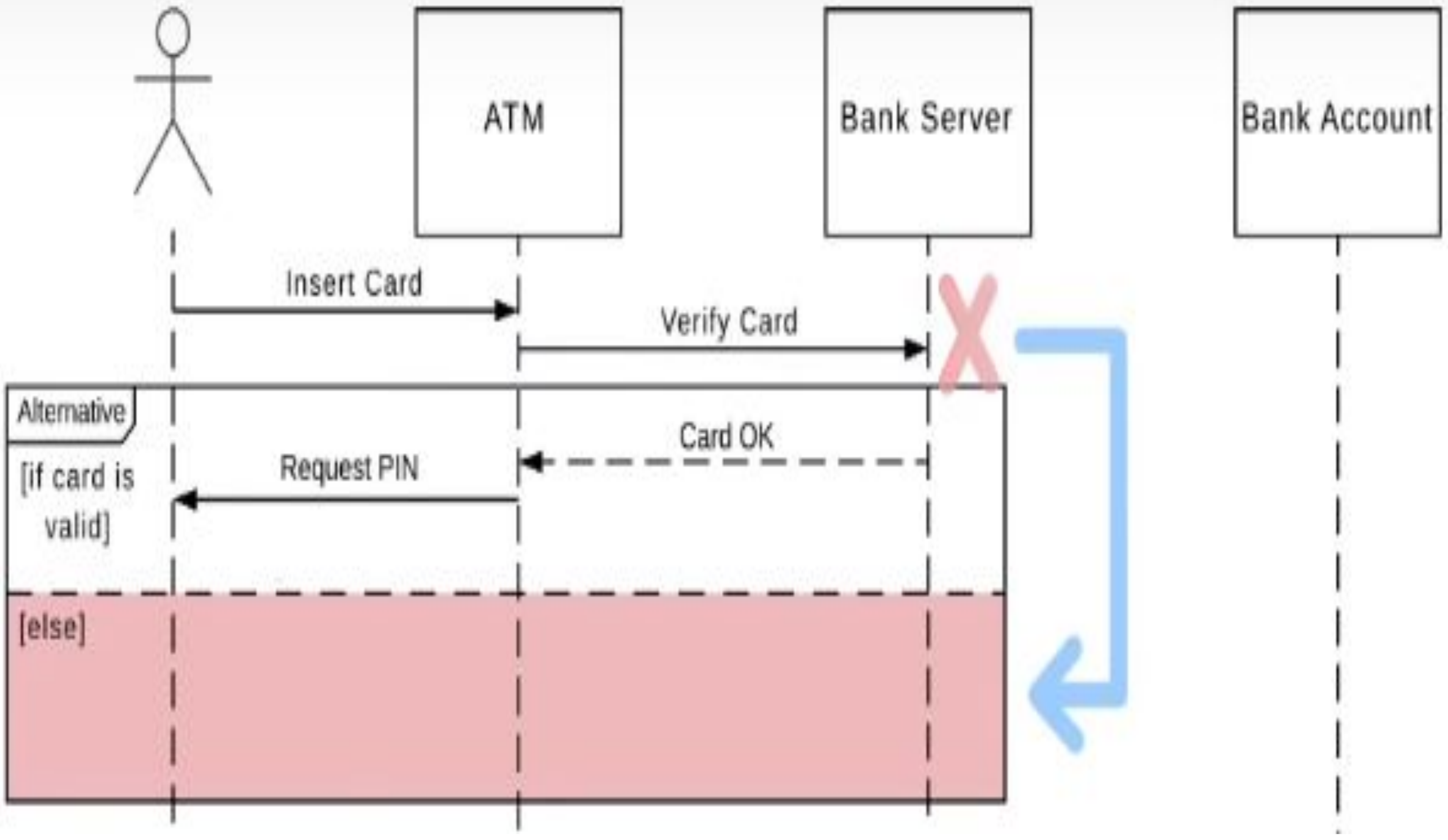


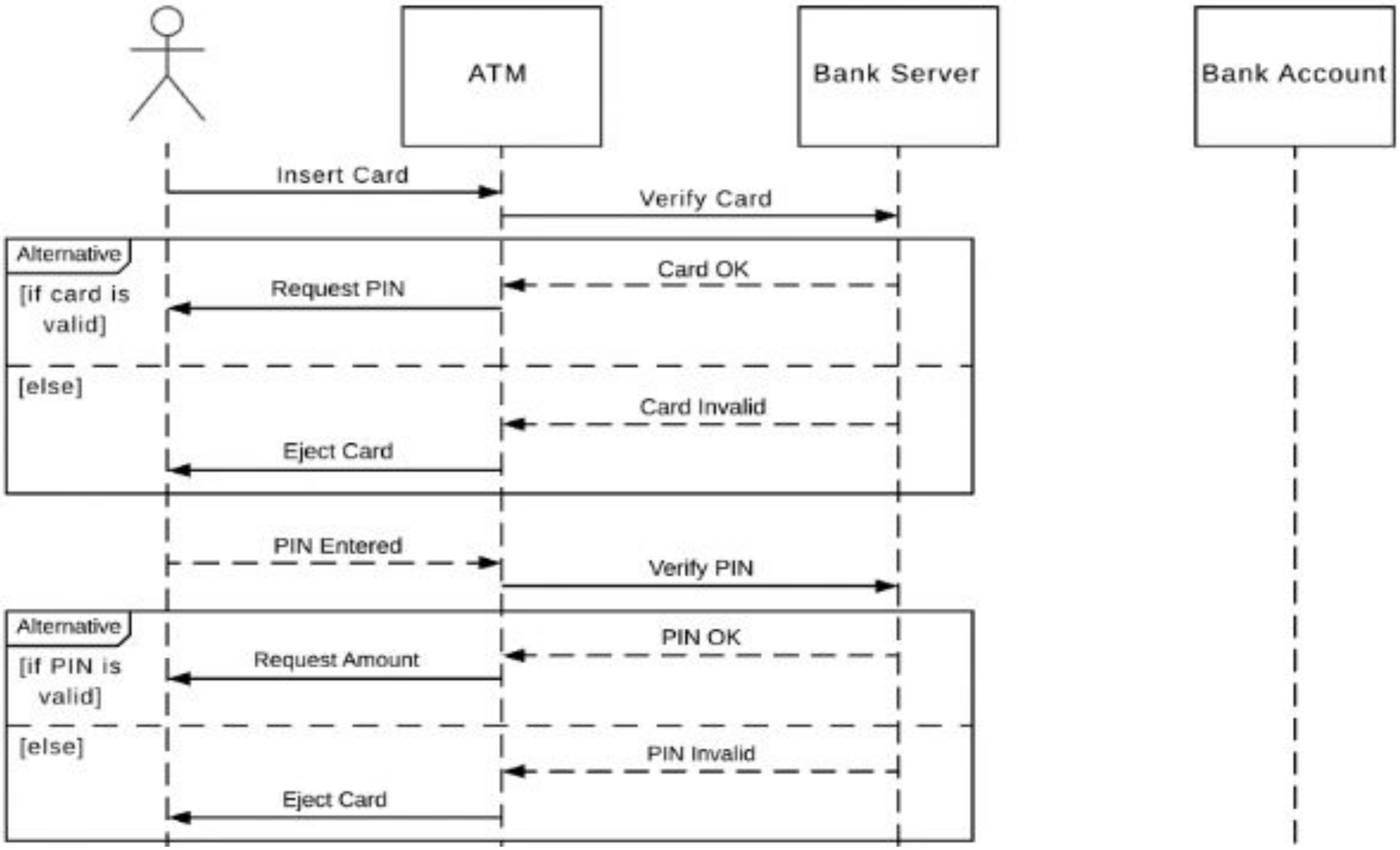
Activities

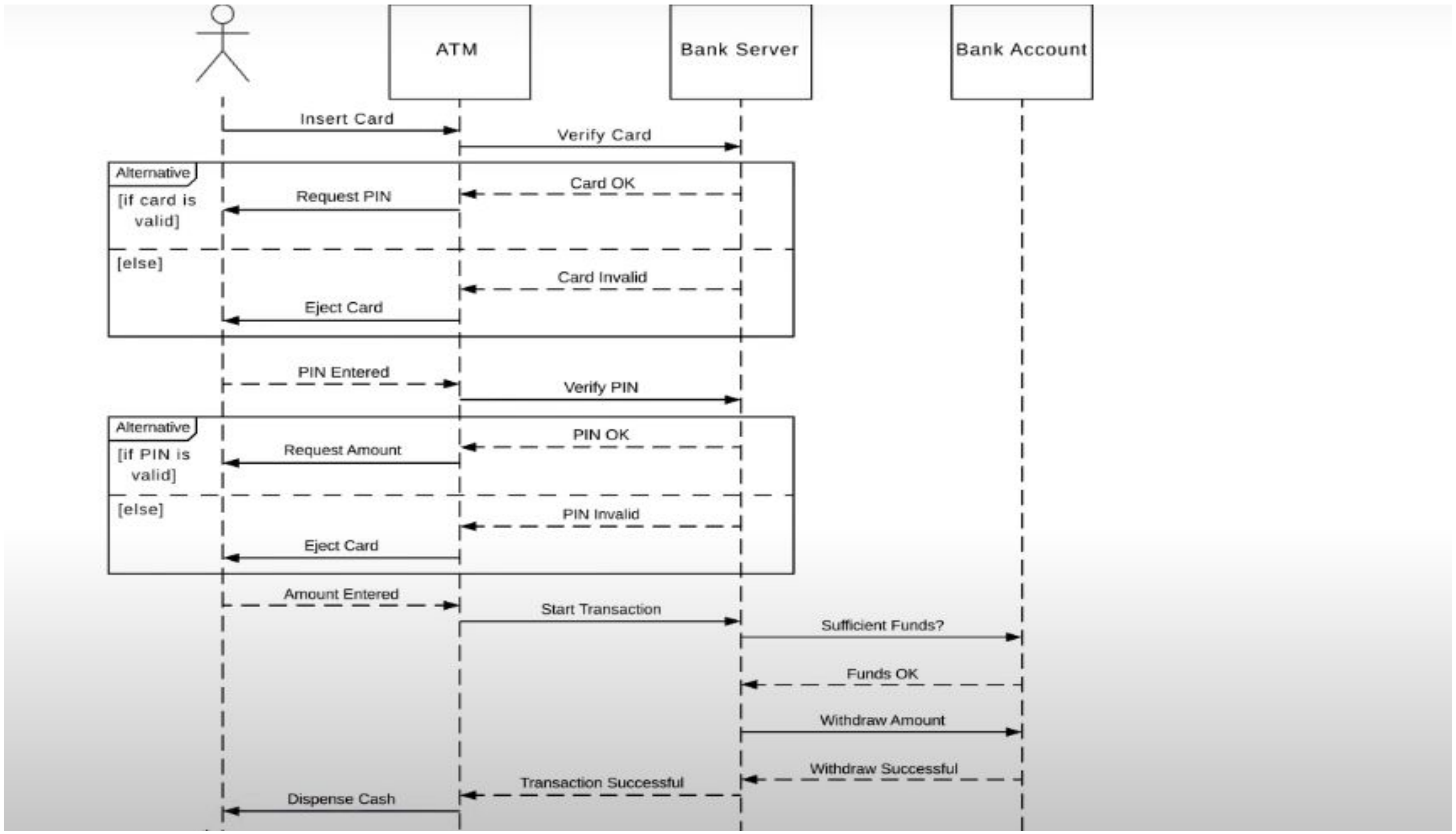

ATM
Bank Server
Bank Account
Insert Card
Verify Card
Card OK
Request PIN

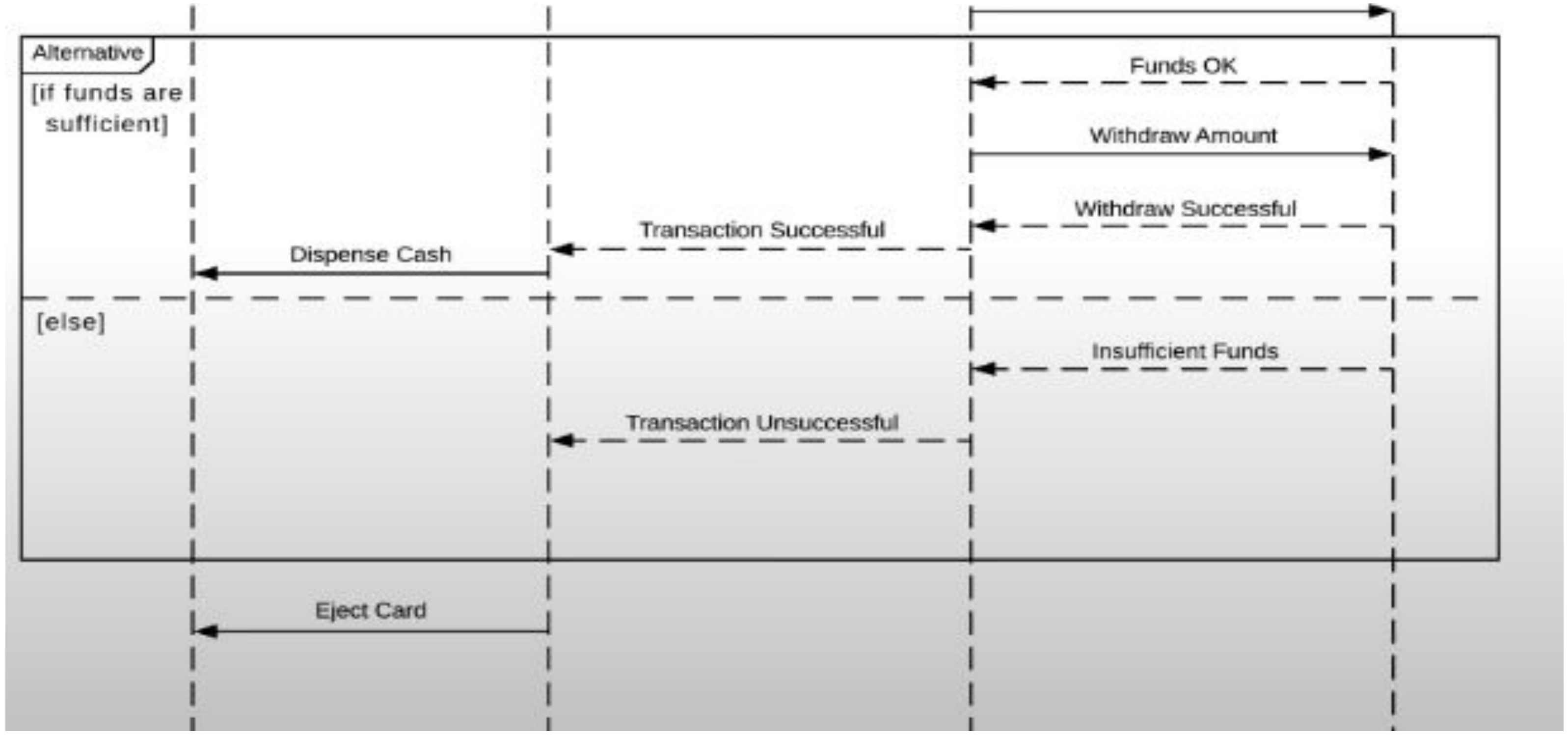
Alternative Frame



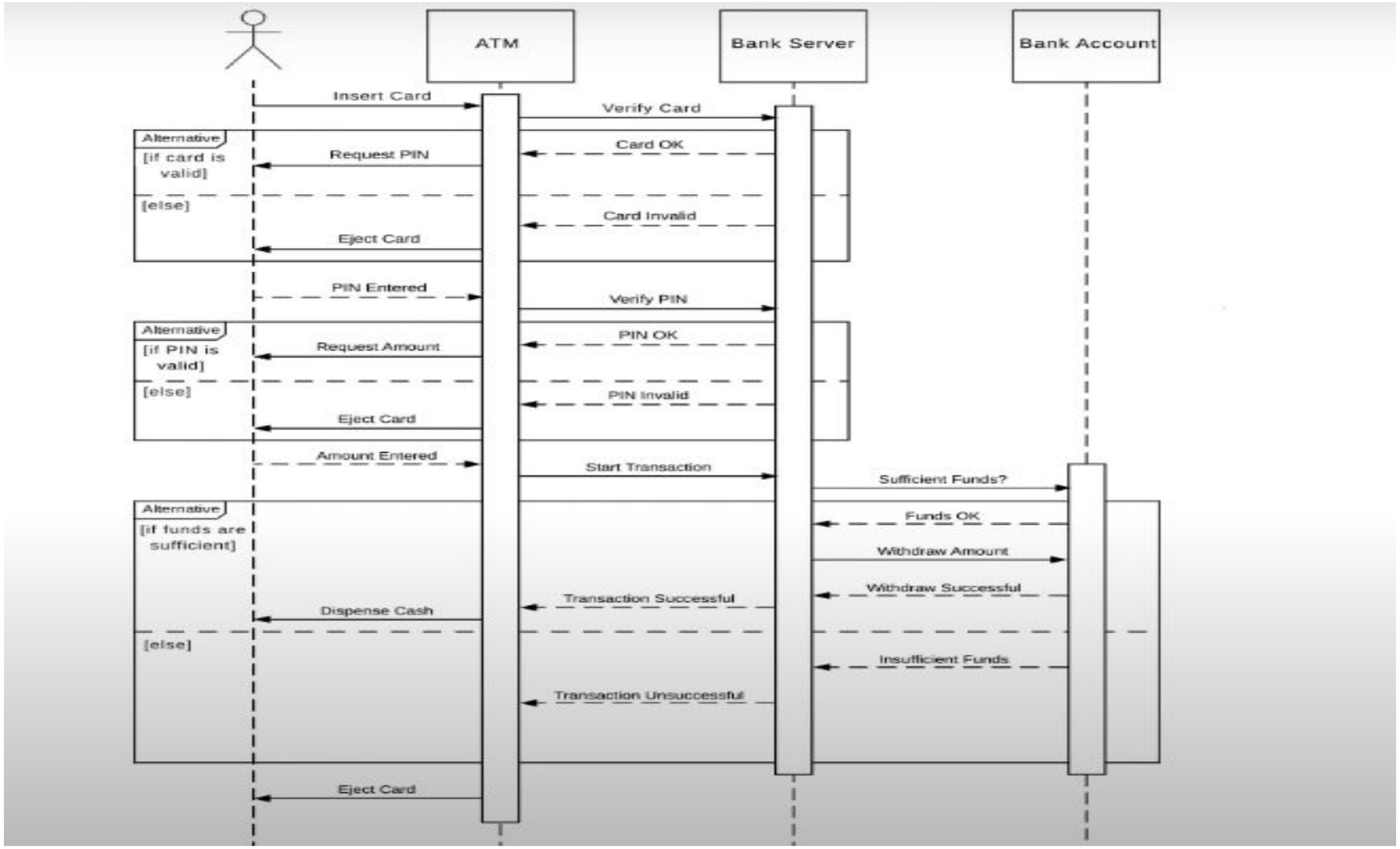

ATM
Bank Server
Bank Account
Insert Card
Verify Card
Alternative
[if card is valid]
Card OK
Request PIN
[else]


ATM
Bank Server
Bank Account
Insert Card
Verify Card
Alternative
[if card is valid]
Card OK
Request PIN
[else]
Card Invalid
Eject Card
PIN Entered
Verify PIN
Alternative
[if PIN is valid]
PIN OK
Request Amount
[else]
PIN Invalid
Eject Card


ATM
Bank Server
Bank Account
Insert Card
Verify Card
Alternative
[if card is valid]
Card OK
Request PIN
[else]
Card Invalid
Eject Card
PIN Entered
Verify PIN
Alternative
[if PIN is valid]
PIN OK
Request Amount
[else]
PIN Invalid
Eject Card
Amount Entered
Start Transaction
Sufficient Funds?
Funds OK
Withdraw Amount
Withdraw Successful
Transaction Successful
Dispense Cash


Alternative
[if funds are sufficient]
Funds OK
Withdraw Amount
Withdraw Successful
Transaction Successful
Dispense Cash
[else]
Insufficient Funds
Transaction Unsuccessful
Eject Card

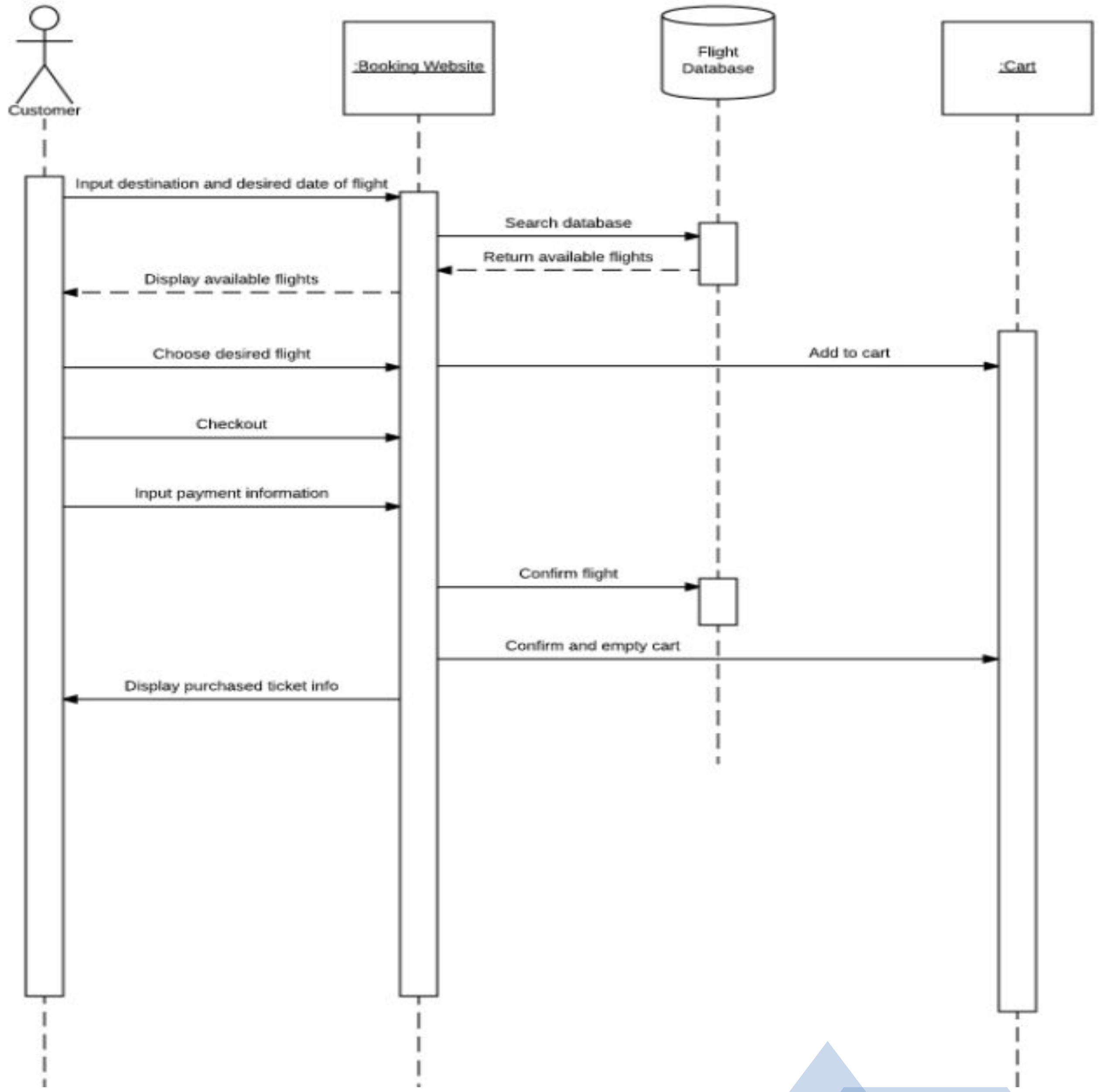
Activation Box



Task

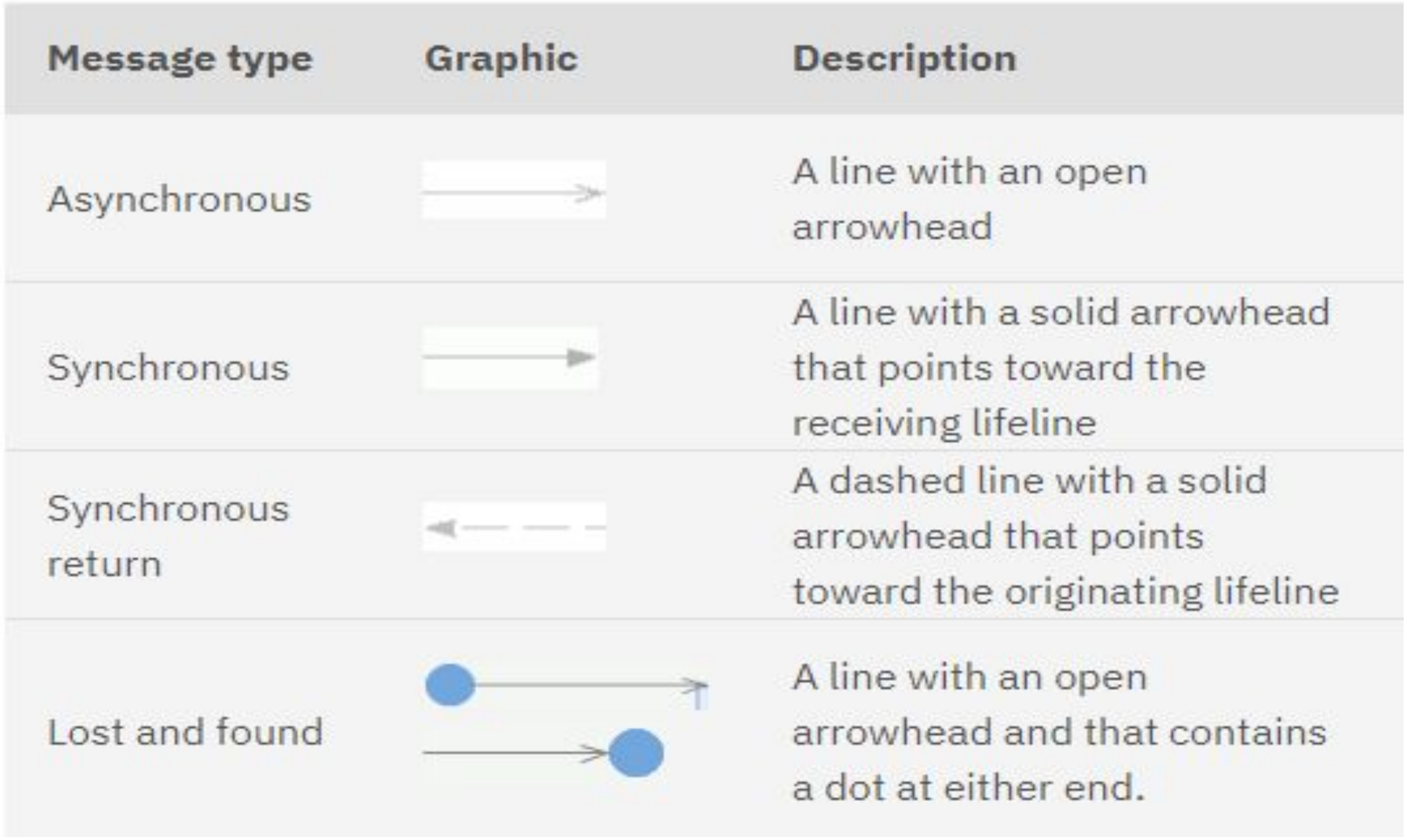
Create a UML sequence diagram that will show your clients how the system’s classes will interact when customers are buying their flight tickets on the booking website.

The system should allow the customer to search available flights from the database by inputting their desired location and departure/arrival date. The website will search the database and return the available flights to display. Once the customer has chosen the flight, they will add the flight to their cart and checkout. They will also then input their payment information and once everything is complete, the website should confirm the flight, empty cart and lastly display a confirmation of the ticket for the flight.


Customer
:Booking Website
Flight Database
:Cart
Input destination and desired date of flight
Search database
Return available flights
Display available flights
Choose desired flight
Add to cart
Checkout
Input payment information
Confirm flight
Confirm and empty cart
Display purchased ticket info

Message type	Description	Example
Create	A create message represents the creation of an instance in an interaction. The create message is represented by the keyword «create». The target lifeline begins at the point of the create message.	In a banking scenario, a bank manager might start a credit check on a client by sending a create message to the server.
Destroy	A destroy message represents the destruction of an instance in an interaction. The destroy message is represented by the keyword «destroy». The target lifeline ends at the point of the destroy message, and is denoted by an X.	A bank manager, after starting a credit check, might close or destroy the credit program application for a customer.
Synchronous call	Synchronous calls, which are associated with an operation, have a send and a receive message. A message is sent from the source lifeline to the target lifeline. The source lifeline is blocked from other operations until it receives a response from the target lifeline.	A bank teller might send a credit request to the bank manager for approval and must wait for a response before further serving the customer.
Asynchronous call	Asynchronous calls, which are associated with operations, typically have only a send message, but can also have a reply message. In contrast to a synchronous message, the source lifeline is not blocked from receiving or sending other messages. You can also move the send and receive points individually to delay the time between the send and receive events. You might choose to do this if a response is not time	A bank customer could apply for credit but can receive banking information over the phone or request money from an ATM, while waiting to hear about the credit application.

Lost and found	A lost message is a message that has a known sender but the receiver is not known. A found message is a message that does not have a known sender but has a receiver.	An outside actor sends a message to a bank manager. The actor is outside the scope of the sequence diagram and is therefore a found message. A lost message can occur when a message is sent to an element outside the scope of the UML diagram.



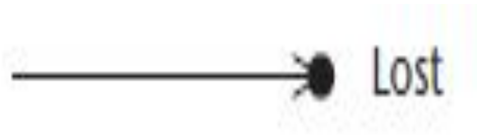
Message type	Graphic	Description
Asynchronous	——>	A line with an open arrowhead
Synchronous	——▶	A line with a solid arrowhead that points toward the receiving lifeline
Synchronous return	◀– – –	A dashed line with a solid arrowhead that points toward the originating lifeline
Lost and found	●——> ——>●	A line with an open arrowhead and that contains a dot at either end.

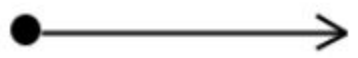
Message Type Notations

Message	Description
——▶	**Synchronous:** A synchronous message between active objects indicates wait semantics; the sender waits for the message to be handled before it continues. This typically shows a method call.
⟶ ⇀ Asynchronous	**Asynchronous:** With an asynchronous flow of control, there is no explicit return message to the caller. An asynchronous message between objects indicates no-wait semantics; the sender does not wait for the message before it continues. This allows objects to execute concurrently.
<------------	**Reply:** This shows the return message from another message.

Message type notations

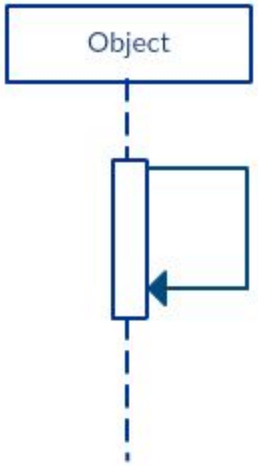
Lost. A lost message occurs when the sender of the message is known but there is no reception of the message. This message allows advanced dynamic models to be built up by fragments without complete knowledge of all the messages in the system. This also allows the modeler to consider the possible impact of a message's being lost.



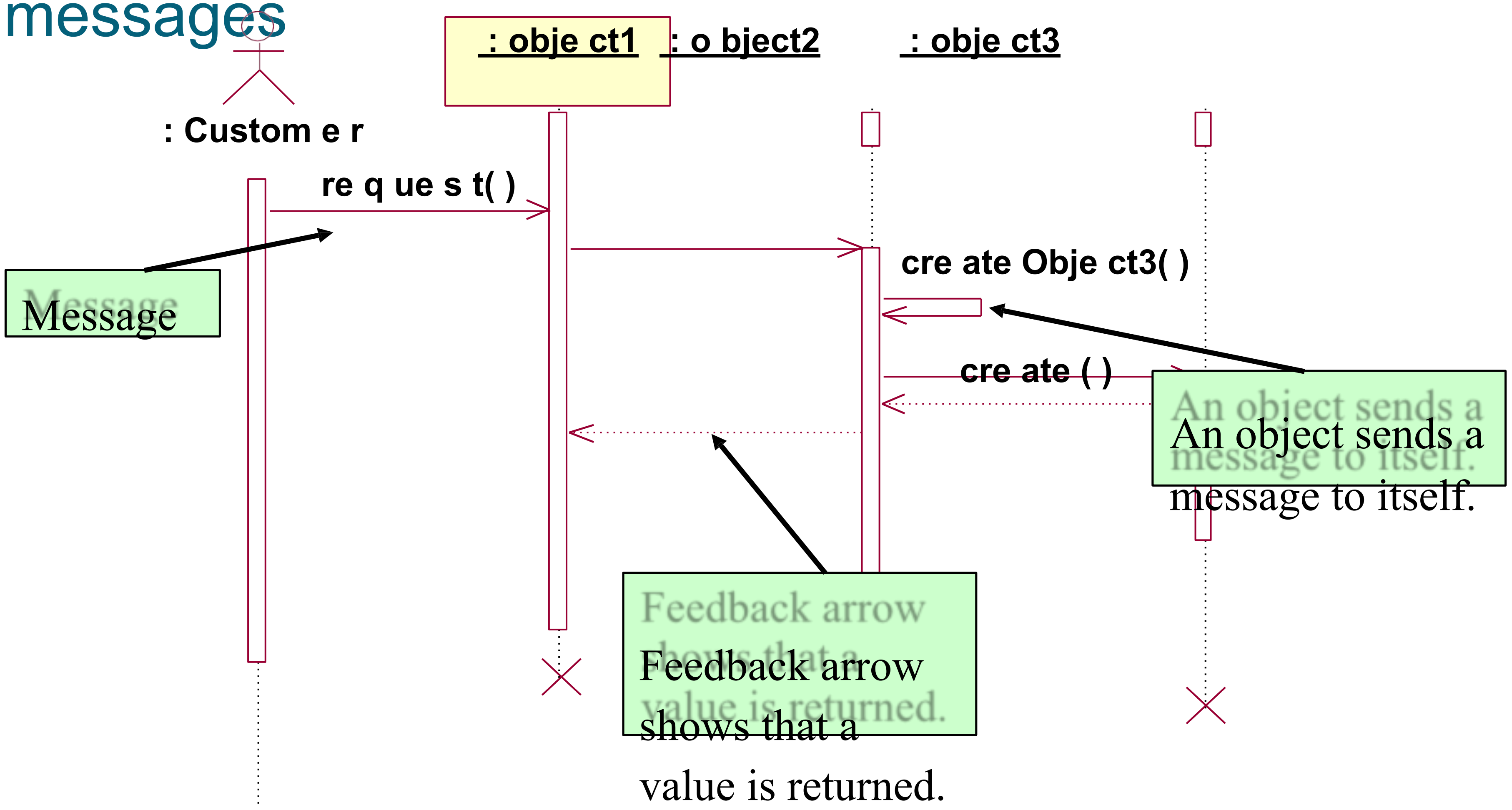
	**Found:** A found message indicates that although the receiver of the message is known in the current interaction fragment, the sender of the message is unknown.

Message type notationS

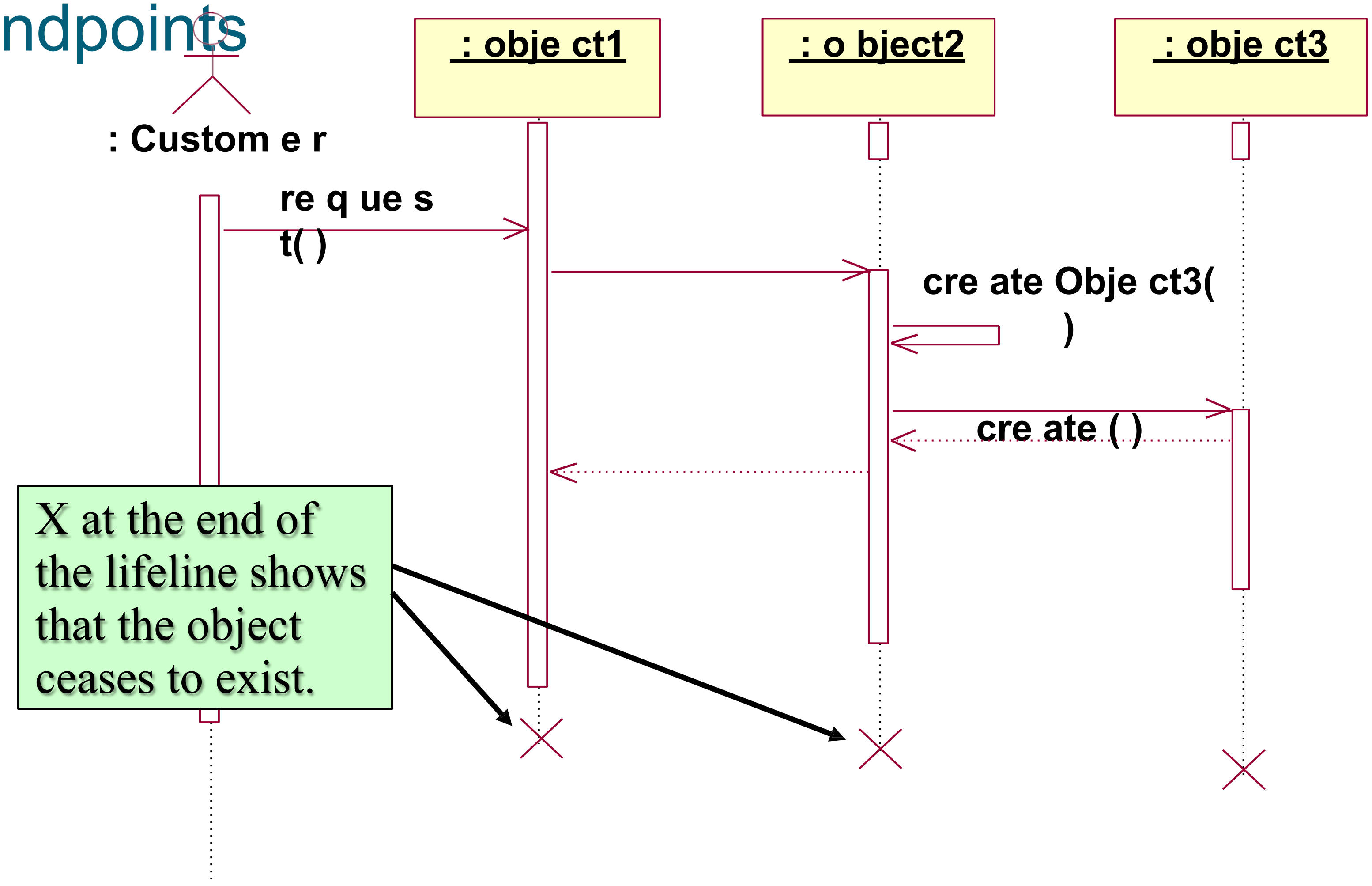
reflexive message: When an object sends a message to itself, it is called a reflexive message. It is indicated with a message arrow that starts and ends at the same lifeline as shown in the example below.



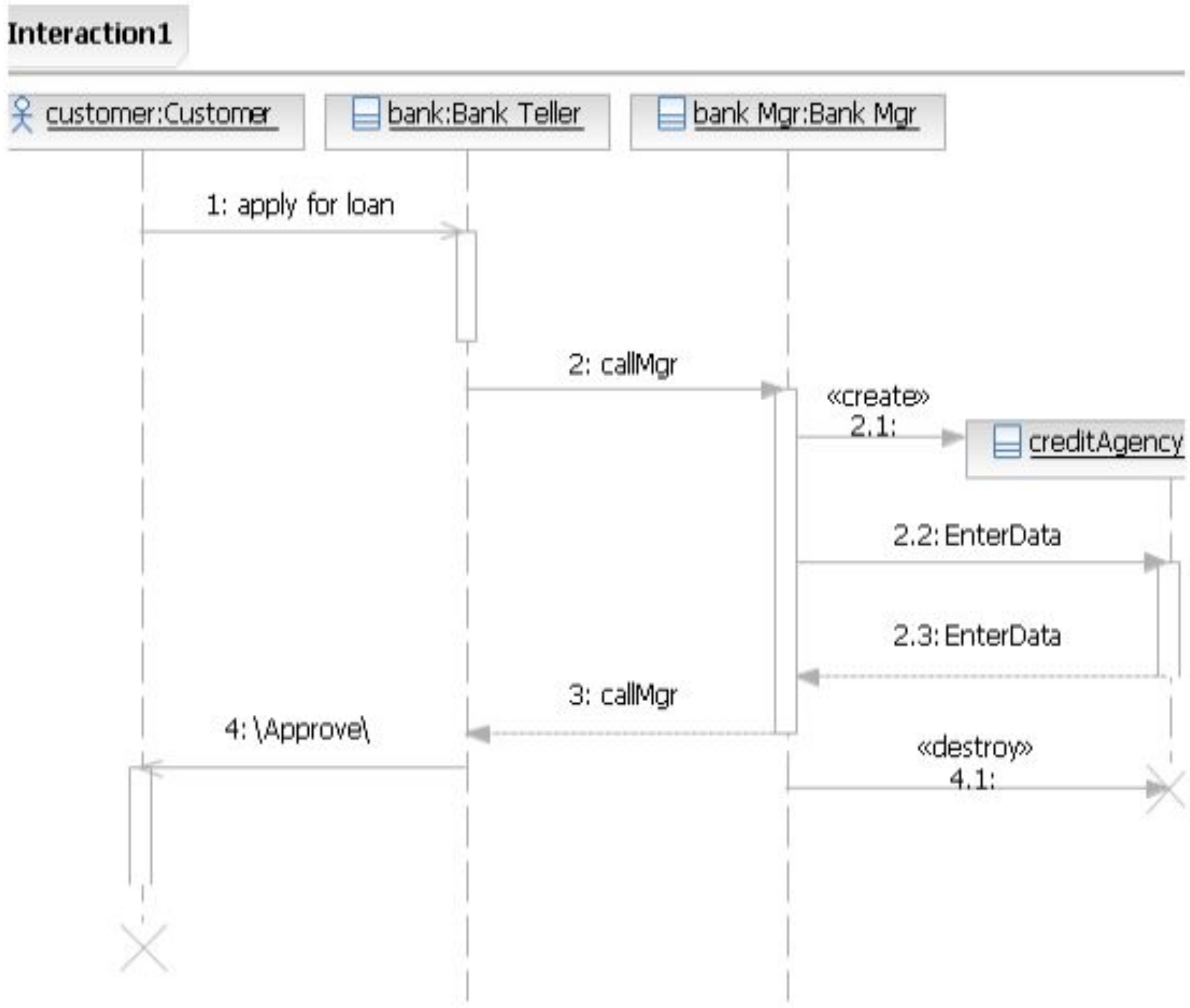
Sequence diagrams - different types of messages



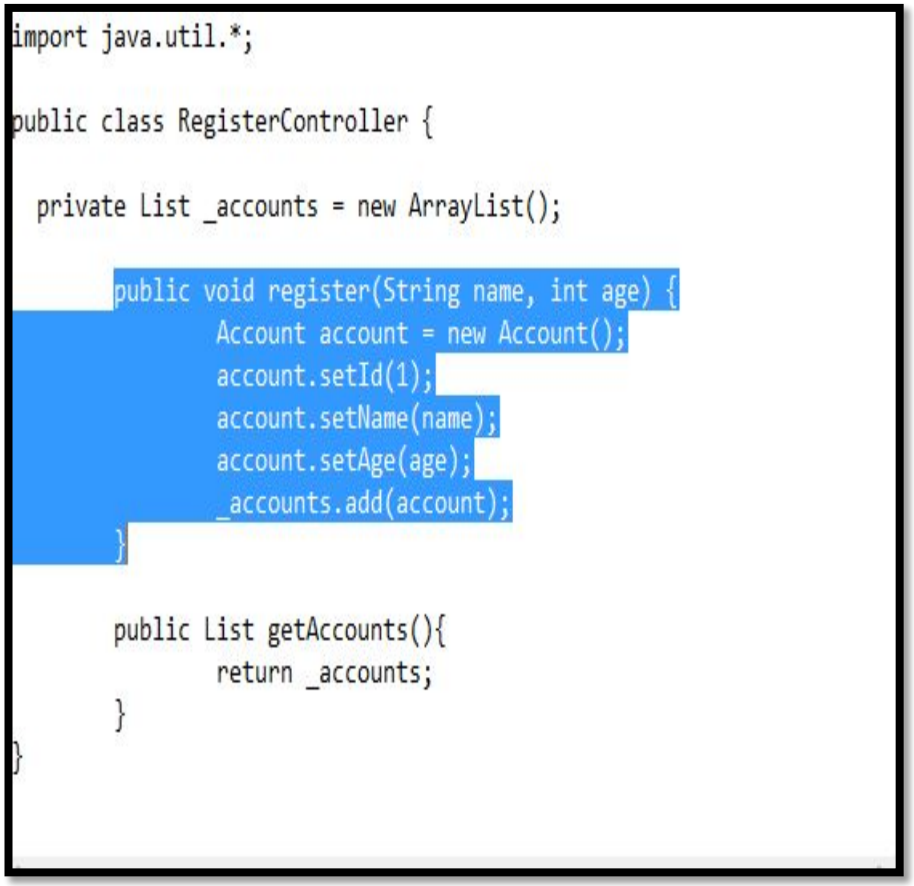
Sequence diagrams: endpoints



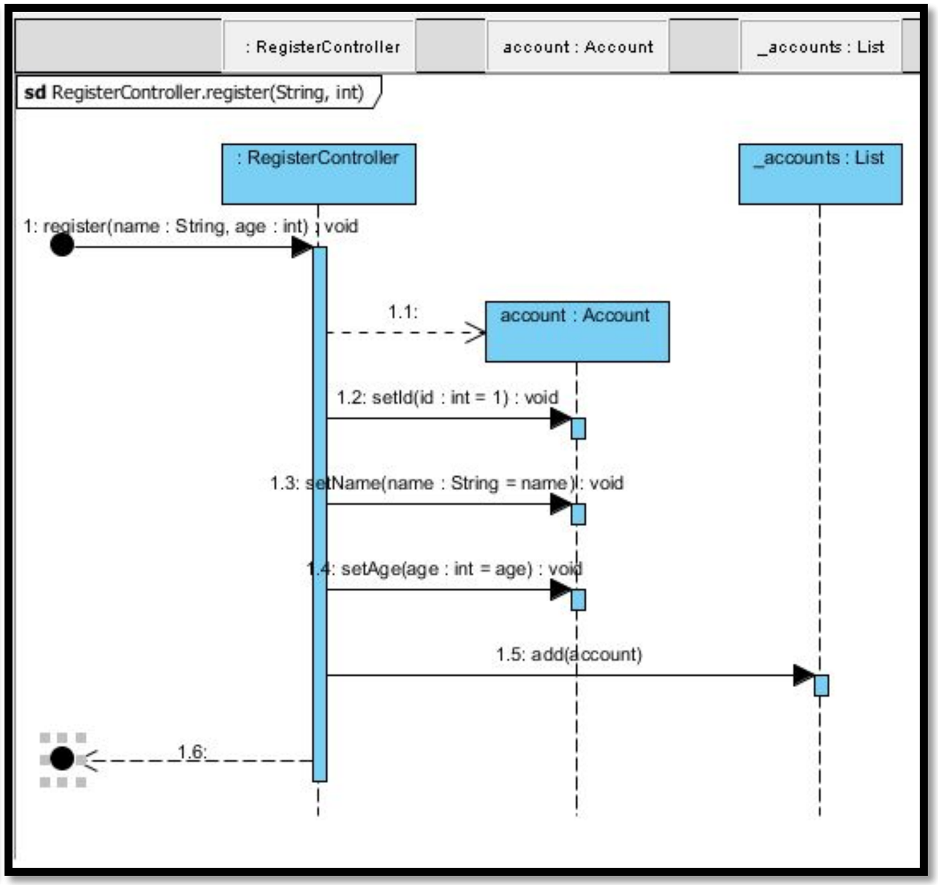
X at the end of the lifeline shows that the object ceases to exist.


Interaction1
customer:Customer
bank:Bank Teller
bank Mgr:Bank Mgr
1: apply for loan
2: callMgr
«create»
2.1:
creditAgency
2.2: EnterData
2.3: EnterData
3: callMgr
4: \Approve\
«destroy»
4.1:

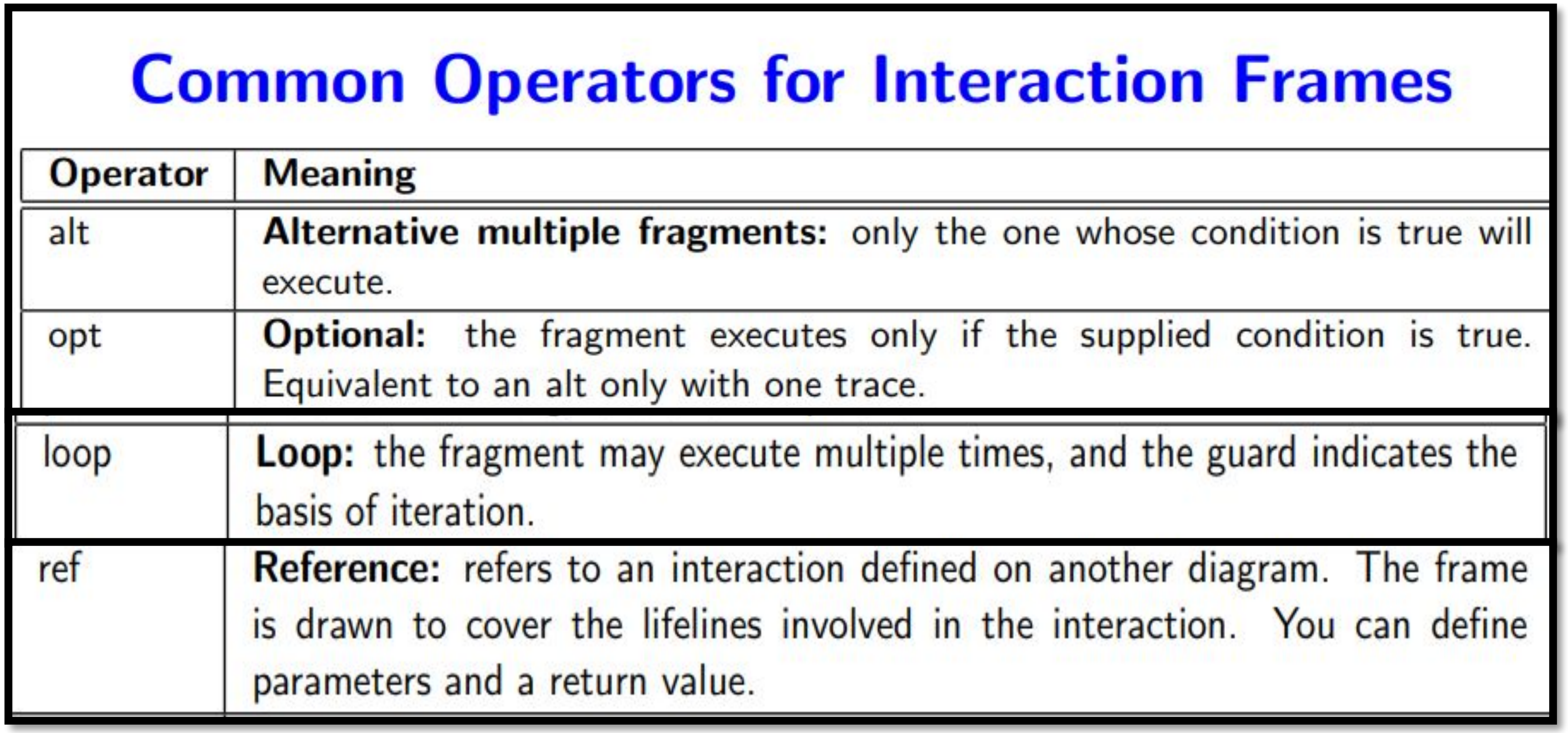
Generate Sequence Diagram



```
import java.util.*;

public class RegisterController {

  private List _accounts = new ArrayList();

        public void register(String name, int age) {
                Account account = new Account();
                account.setId(1);
                account.setName(name);
                account.setAge(age);
                _accounts.add(account);
        }

        public List getAccounts(){
                return _accounts;
        }
}
```


: RegisterController
account : Account
_accounts : List
sd RegisterController.register(String, int)
: RegisterController
_accounts : List
1: register(name : String, age : int) : void
1.1:
account : Account
1.2: setId(id : int = 1) : void
1.3: setName(name : String = name) : void
1.4: setAge(age : int = age) : void
1.5: add(account)
1.6:

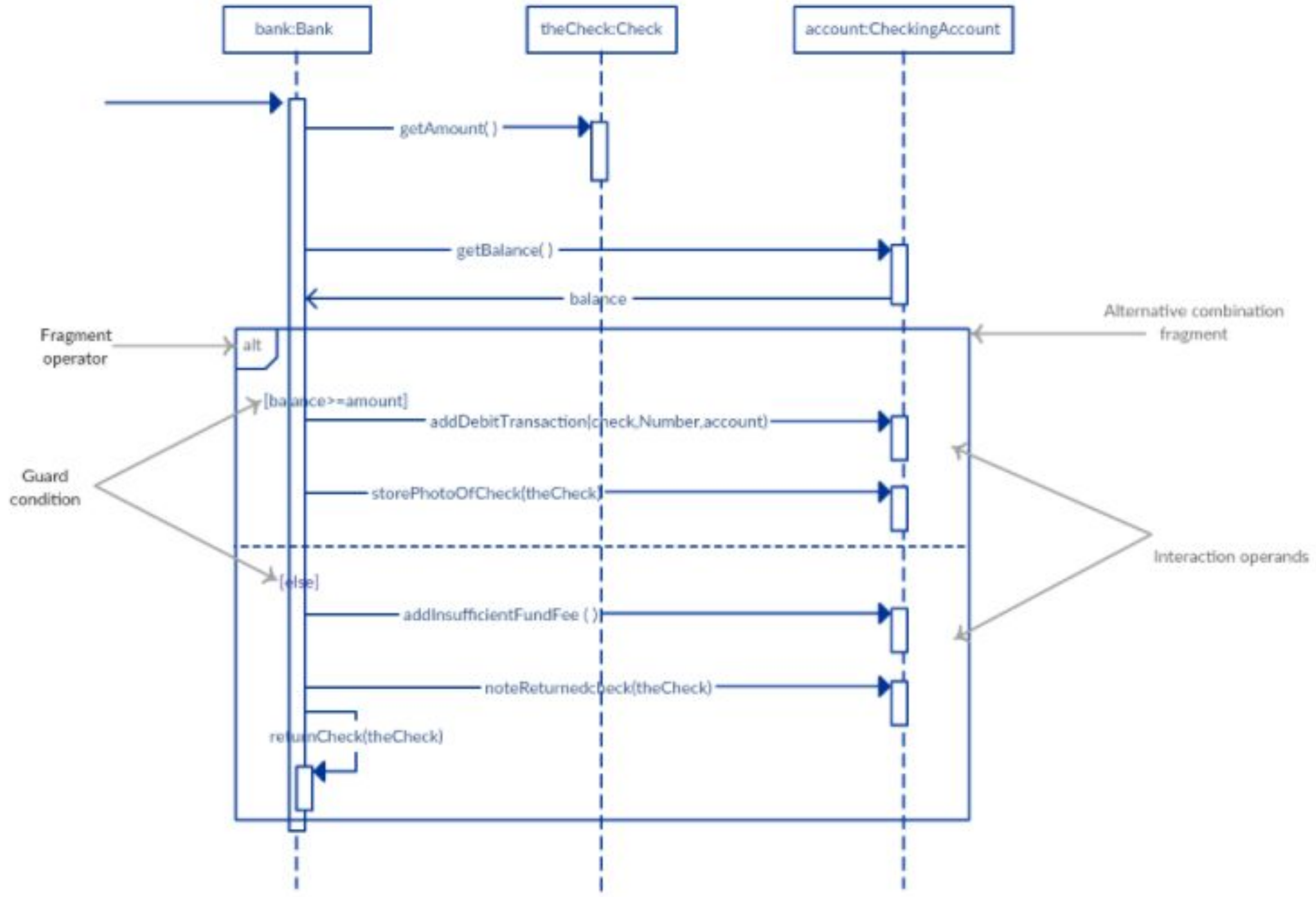
Frames in UML Sequence Diagram:



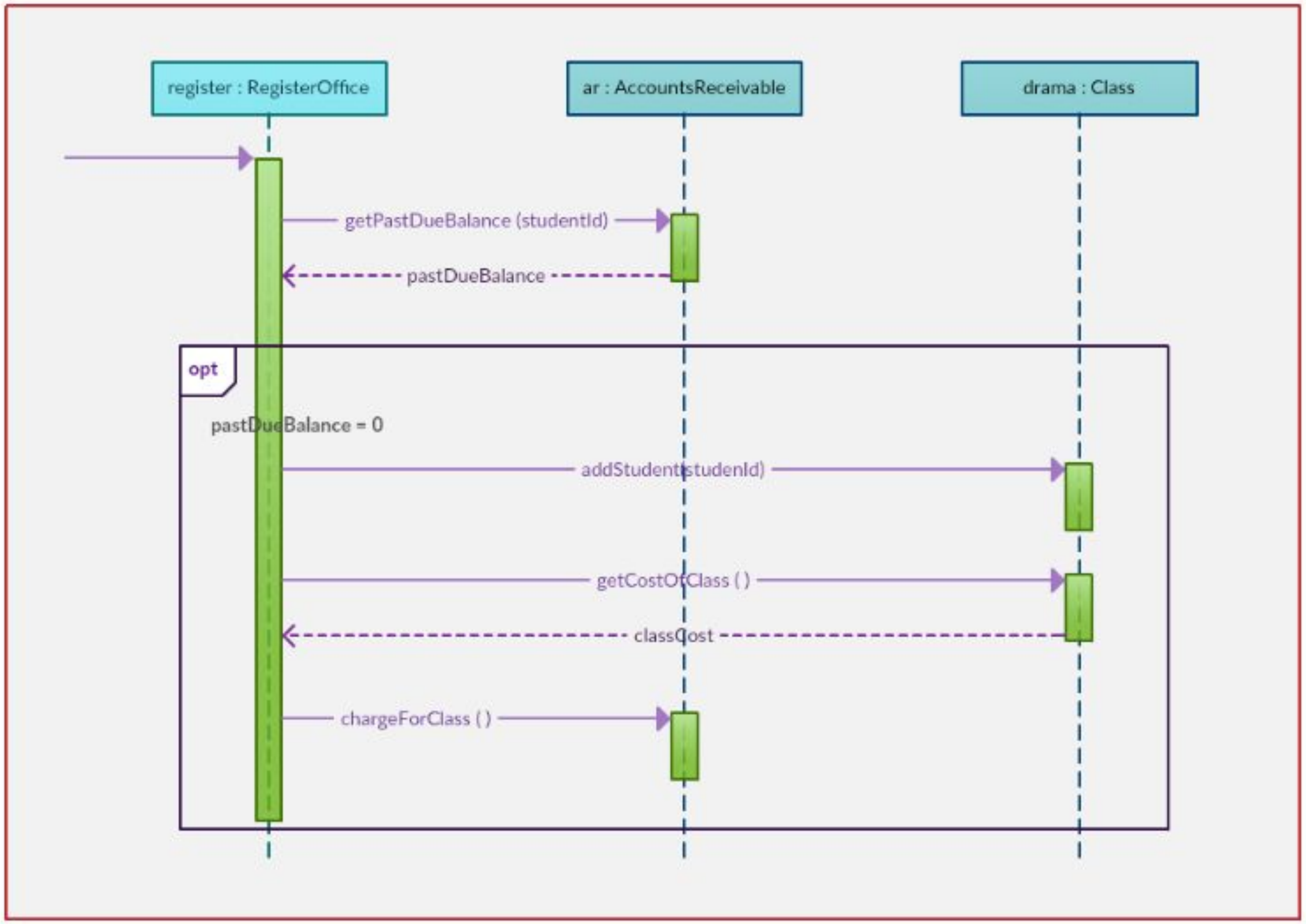
Common Operators for Interaction Frames

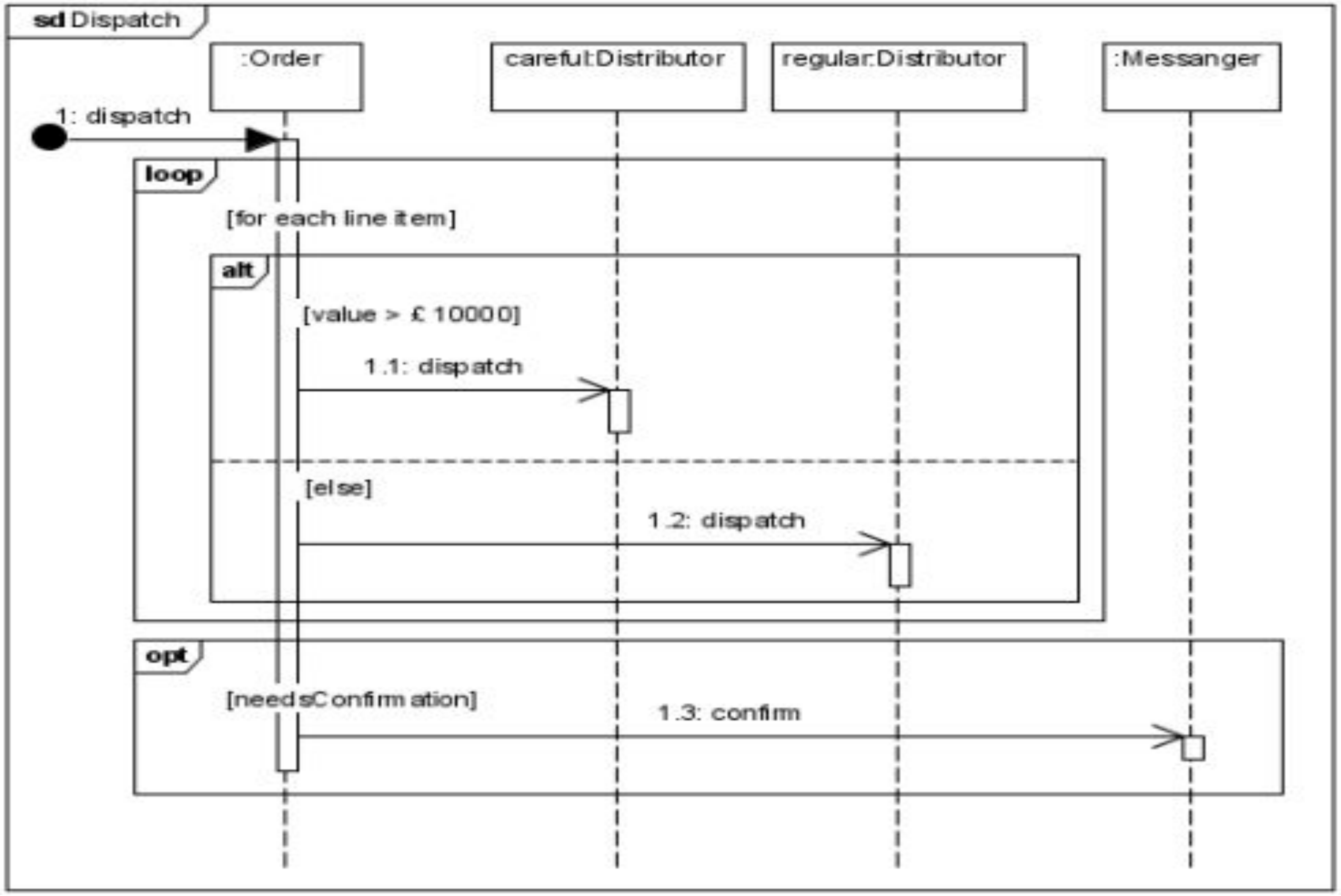
Operator	Meaning
alt	**Alternative multiple fragments:** only the one whose condition is true will execute.
opt	**Optional:** the fragment executes only if the supplied condition is true. Equivalent to an alt only with one trace.
loop	**Loop:** the fragment may execute multiple times, and the guard indicates the basis of iteration.
ref	**Reference:** refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value.

Alternatives: alt



Example of an Option Combination Fragment




sd Dispatch
:Order
careful:Distributor
regular:Distributor
:Messanger
1: dispatch
loop
[for each line item]
alt
[value > £10000]
1.1: dispatch
[else]
1.2: dispatch
opt
[needsConfirmation]
1.3: confirm

Sequence diagrams - strengths and weaknesses

Strengths:

-they clearly show sequence or time ordering of messages

-the notation is rather simple

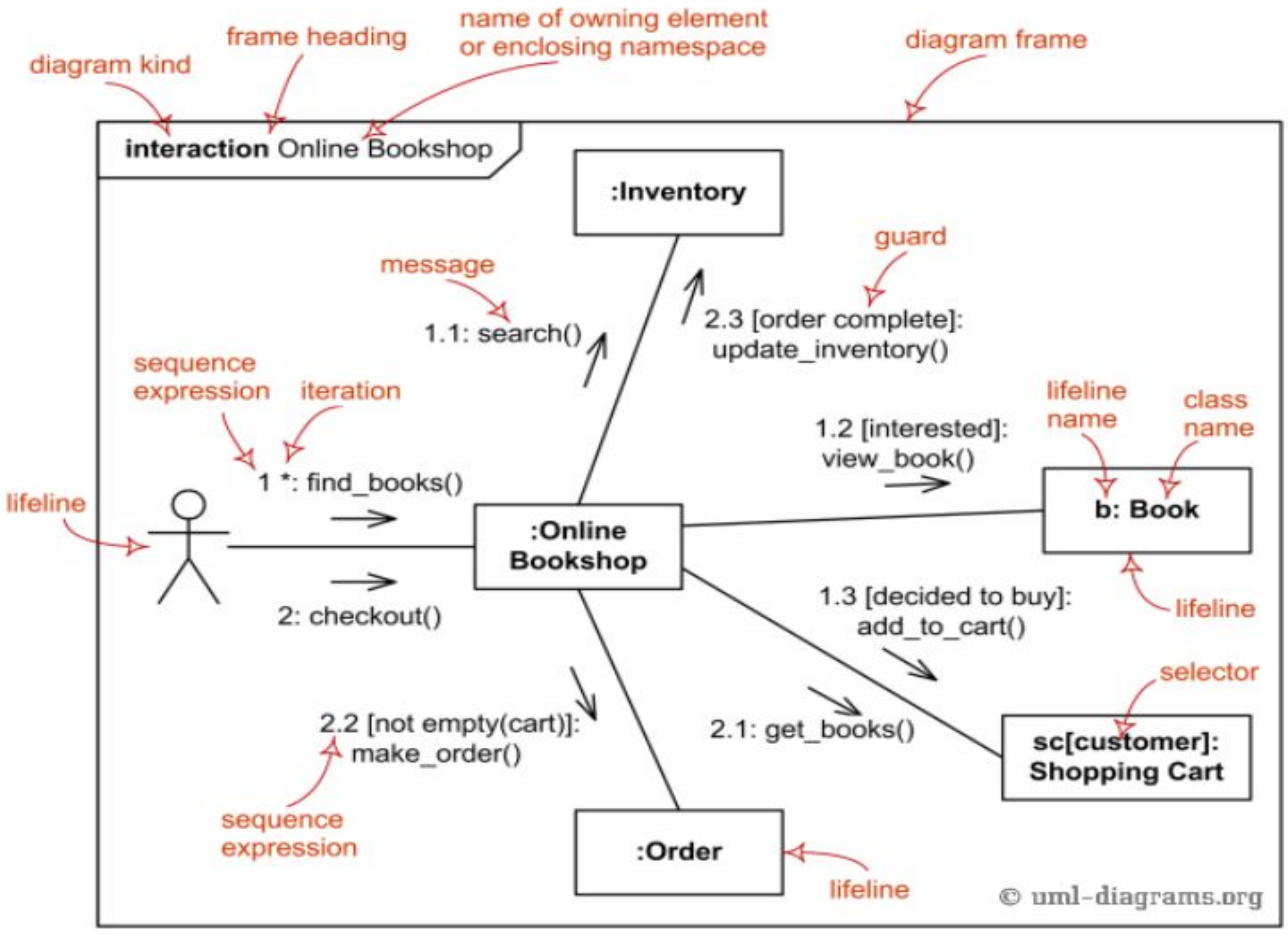
Weaknesses:

-forced to extend to right when adding new objects, consumes a lot of space

Collaboration/Communication Diagram:

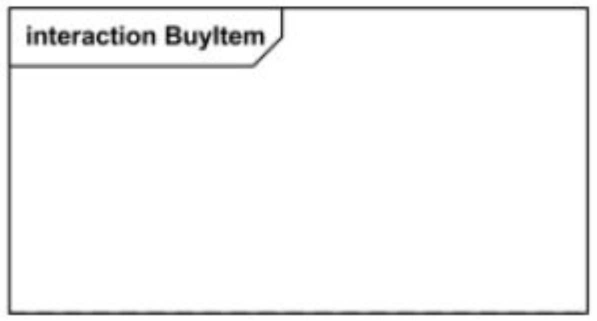
- A collaboration diagram shows the relationship between objects and the order of messages passed between them.
- The objects are listed as icons and arrows indicate the messages being passed between them.
- The numbers next to the messages are called sequence numbers and, as the name suggests, they show the sequence of the messages as they are passed between the objects.

Communication Diagram Elements



Frame

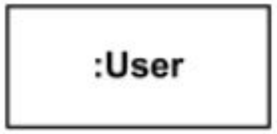
Communication diagrams could be shown within a rectangular **frame** with the **name** in a compartment in the upper left corner.



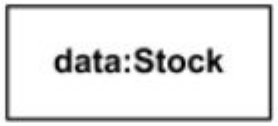
Interaction ***Frame*** *for* **Communication Diagram** *BuyItem*

Lifeline

A **Lifeline** is shown as a rectangle (corresponding to the "head" in sequence diagrams). Lifeline in sequence diagrams does have "tail" representing the **line of life** whereas "lifeline" in **communication diagram** has no line, just "head". Usually the head is a white rectangle containing name of the class after colon.



Anonymous lifeline of class User.



Lifeline *"data" of class Stock*

Objects

- rectanglescontaining the object signature
- – object signature:

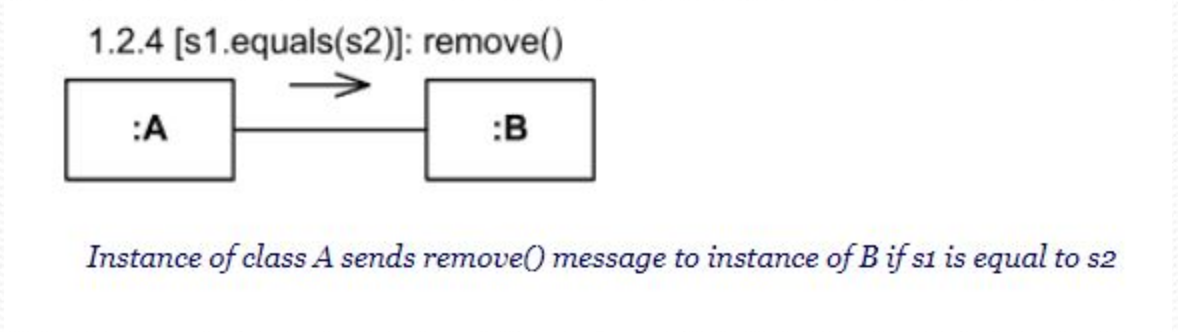
object name : object Class

• object name (optional)

- starts with lowercase letter
- class name (mandatory)
- - starts with uppercase letter
- – objects connected by lines
- – user (stick man) can appear

Messages

- **Messages** – are labeled like C and Java function calls followed by round brackets, and can have parameters and return values are followed by an arrow to show direction internal messages are numbered, starting from 1



Instance of class A sends remove() message to instance of B if s1 is equal to s2

Sequence Expression

The **sequence expression** is a dot separated list of **sequence terms** followed by a colon (":") and message name after that:

sequence-expression = ***sequence-term*** '.' ':' ***message-name***

For example,

3b.2.2:m5

contains sequence expression **3b.2.2** and message name **m5**.

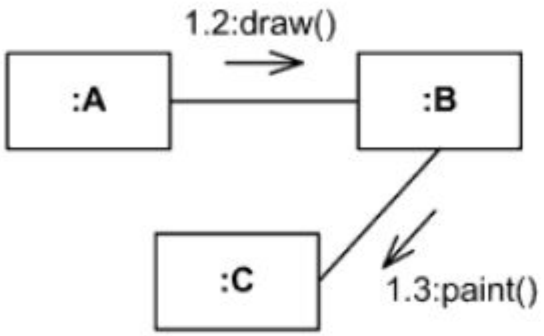
Each **sequence term** represents a level of **procedural nesting** within the overall interaction. Each sequence-term has the following syntax:

sequence-term ::= [***integer*** [***name***]] [***recurrence***]

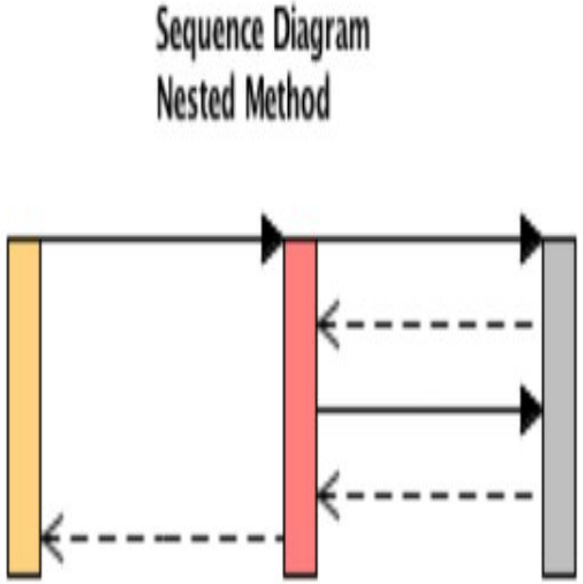
Sequence Expression

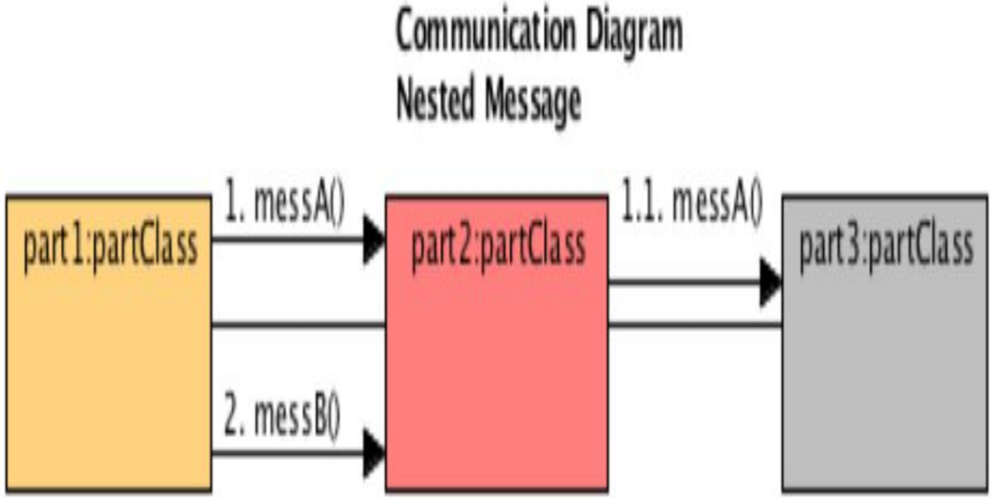
For example,

- message with sequence 2 follows message with sequence 1,
- 2.1 follows 2
- 5.3 follows 5.2 within activation 5
- 1.2.4 follows message 1.2.3 within activation 1.2.



Instance of A sends draw() message to instance of B, and after that B sends paint() to C

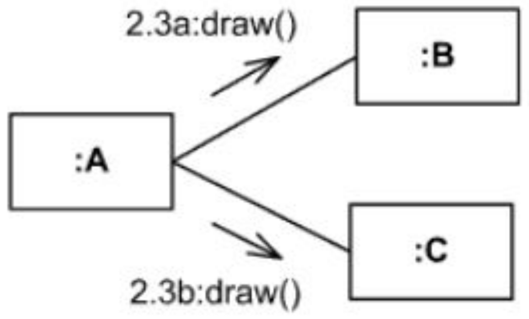

Sequence Diagram
Nested Method


Communication Diagram
Nested Message
part1:partClass
1. messA()
part2:partClass
1.1. messA()
part3:partClass
2. messB()

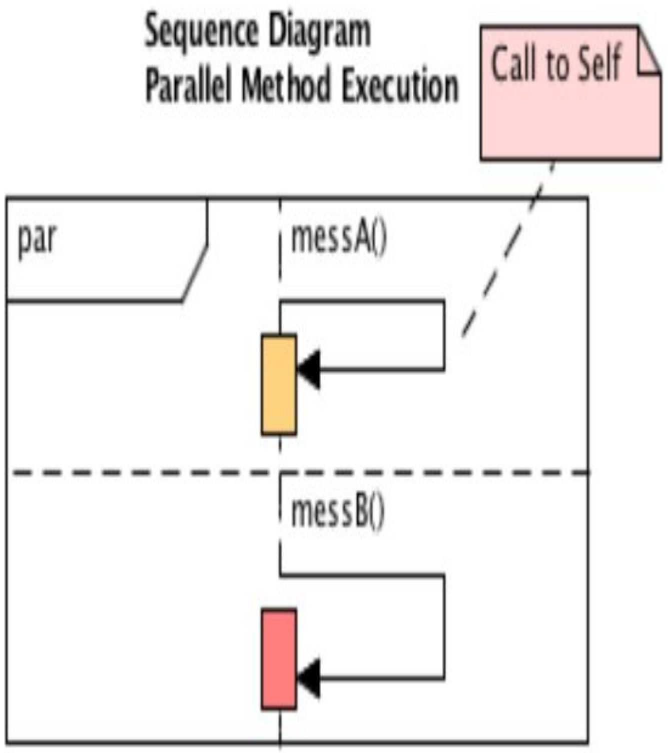
concurrent thread

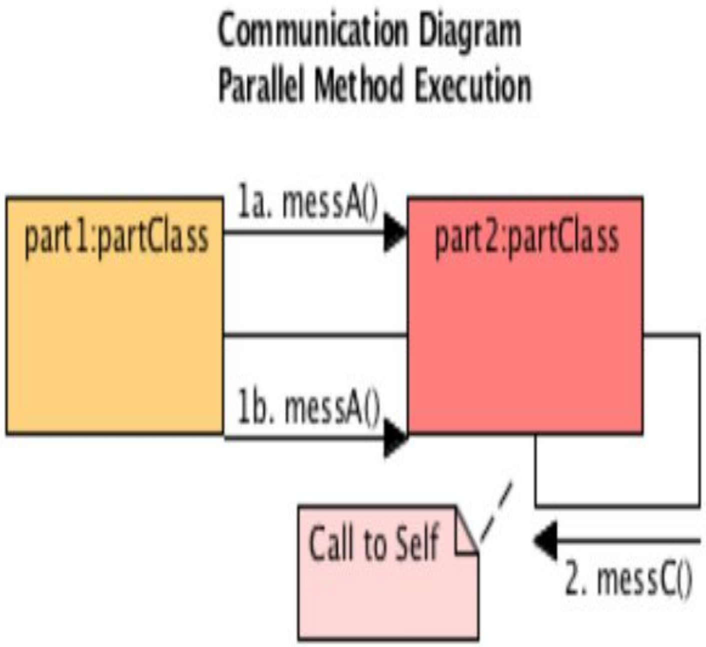
The **name** represents a **concurrent thread** of control. Messages that differ in the final name are concurrent at that level of nesting.
For example,

- messages 2.3a and 2.3b are concurrent within activation 2.3,
- 1.1 follows 1a and 1b,
- 3a.2.1 and 3b.2.1 follow 3.2.



Instance of A sends draw() messages concurrently to instance of B and to instance of C


Sequence Diagram
Parallel Method Execution
Call to Self
par
messA()
messB()


Communication Diagram
Parallel Method Execution
1a. messA()
part1:partClass
part2:partClass
1b. messA()
Call to Self
2. messC()

recurrence

The **recurrence** defines **conditional** or **iterati ve** execution of zero or more messages that are executed depending on the specified condition.

recurrence ::= ***branch*** | ***loop***

branch ::= '[' ***guard*** ']'

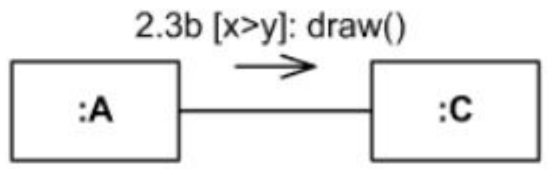
loop ::= '*' ['||']   ['['***iteration-clause*** ']']

guard

A **guard** specifies condition for the message to be sent (executed) at the given nesting depth. UML does not specify guard syntax, so it could be expressed in pseudocode, some programming language, or something else.

For example,

- **2.3b [x>y]: draw()** - message draw() will be executed if x is greater than y,
- **1.1.1 [s1.equals(s2)]: remove()** - message remove() will be executed if s1 equals s2.



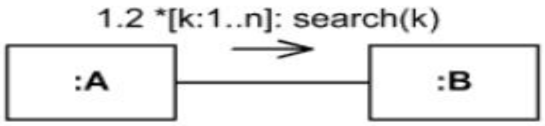
Instance of class A will send message draw() to the instance of C, if x > y

iteration notation

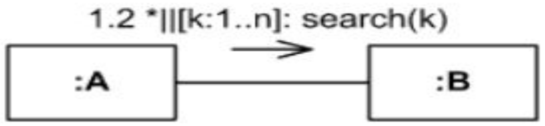
The * iteration notation specifies that the messages in the iteration will be executed **sequentially**. The *|| (star followed by a double vertical line) iteration notation specifies **concurrent** (parallel) execution of messages.

For example,

- **4.2c *[i=1..12]: search(t[i])** - search() will be executed 12 times, one after another
- **4.2c *||[i=1..12]: search(t[i])** - 12 search() messages will be sent concurrently,
- **2.2 *: notify()** - message notify() will be repeated some unspecified number of times.



Instance of class A will send search() message to instance of B n times, one by one

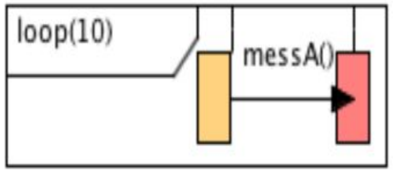


Instance of class A will send n concurrent search() messages to instance of B

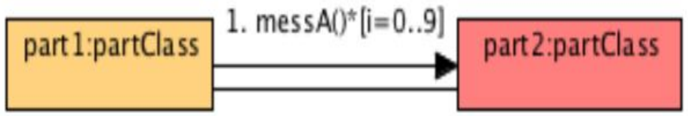
- **Return value**
- name followed by :=

- **Argument list**
  - names separated by commas, within round brackets
  - Types of message flows

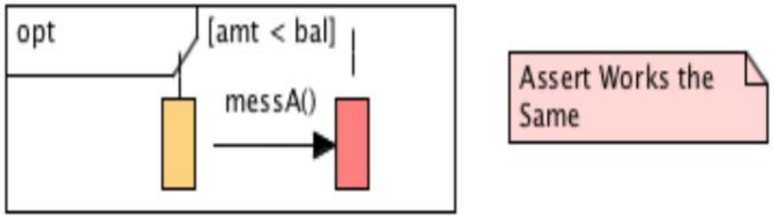
Sequence Diagram
Execute Multiple Times



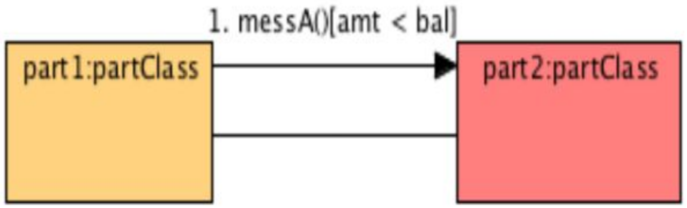
Communication Diagram
Execute Multiple Times

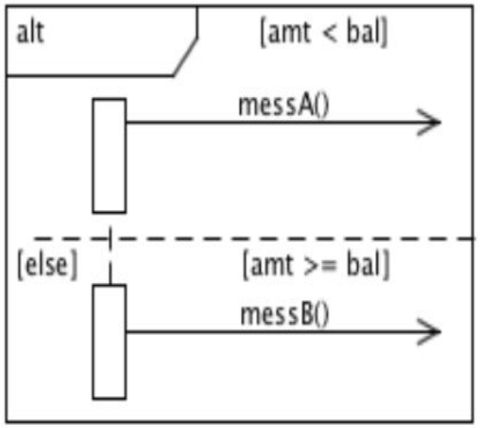


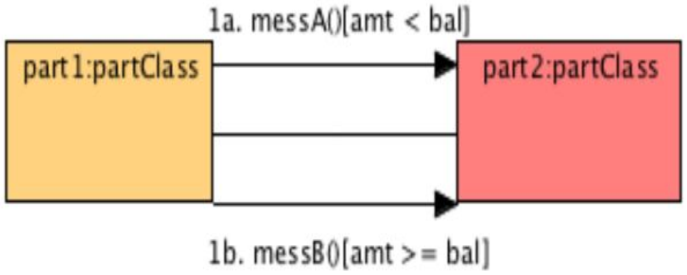
Sequence Diagram
Conditional Execution



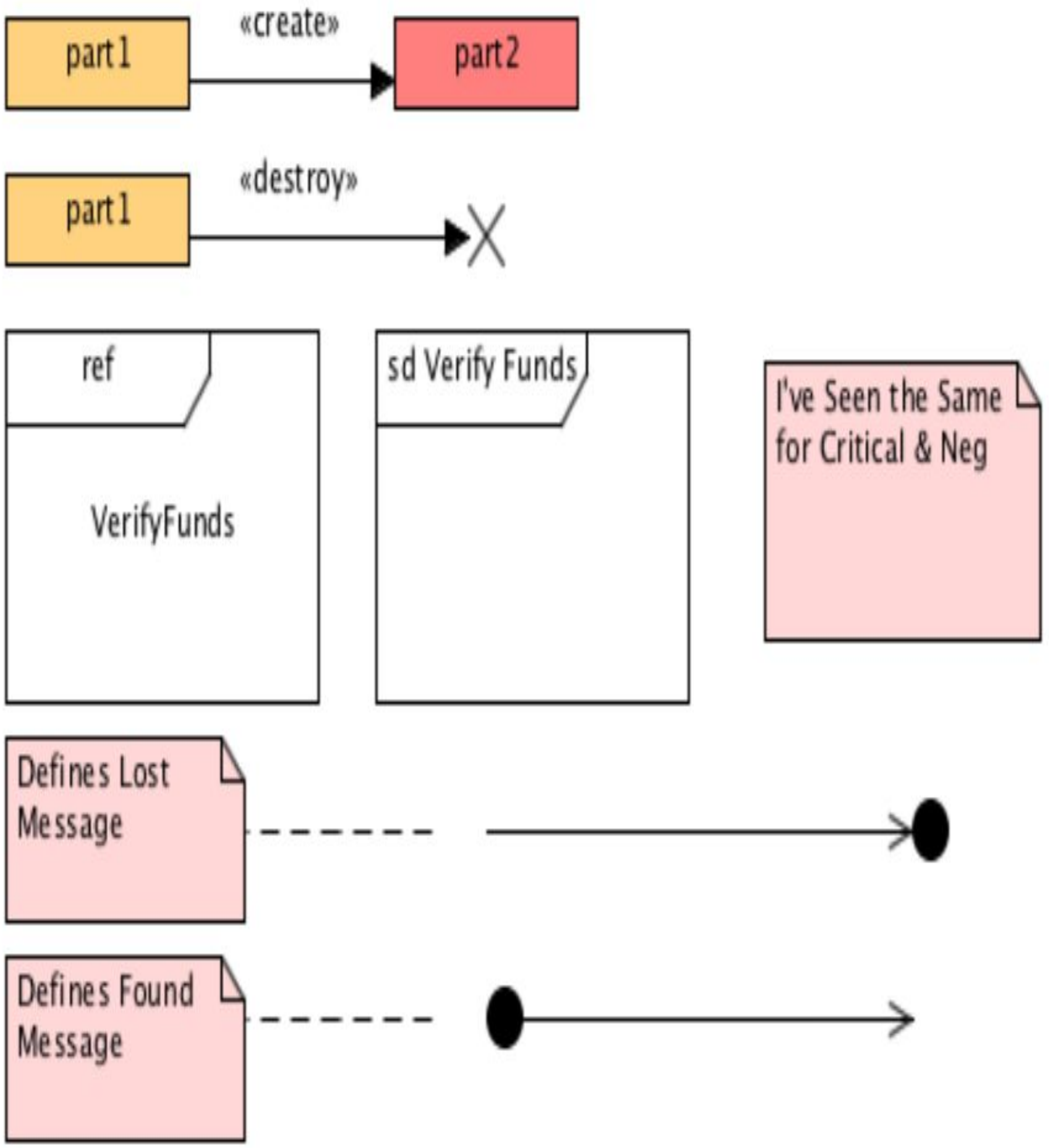
Communication Diagram
Conditional Execution



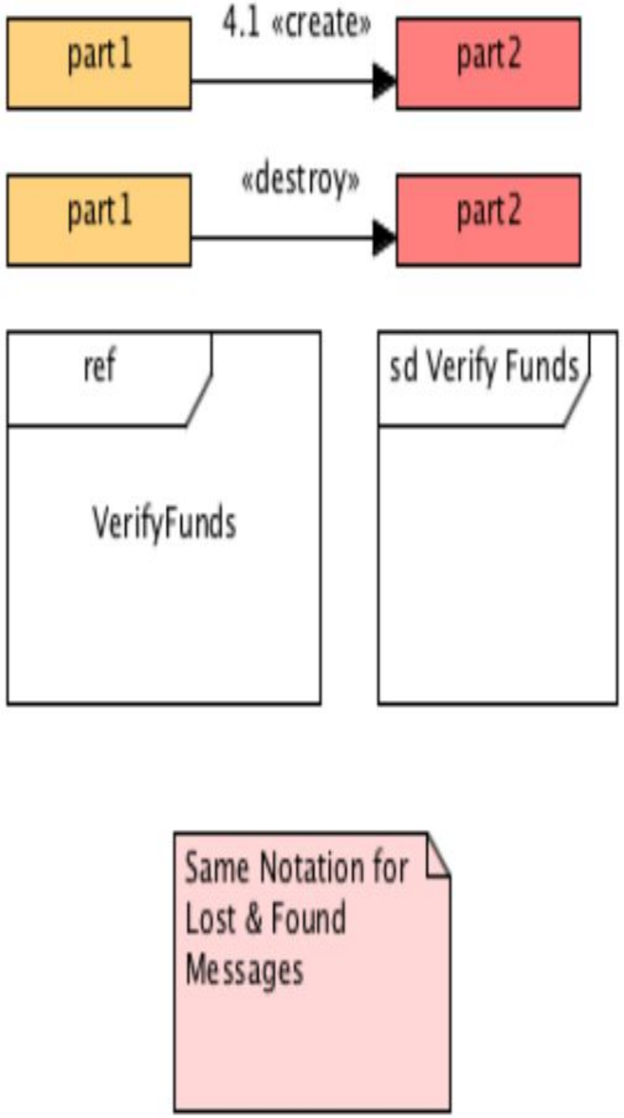




Sequence Diagram
Random Stuff



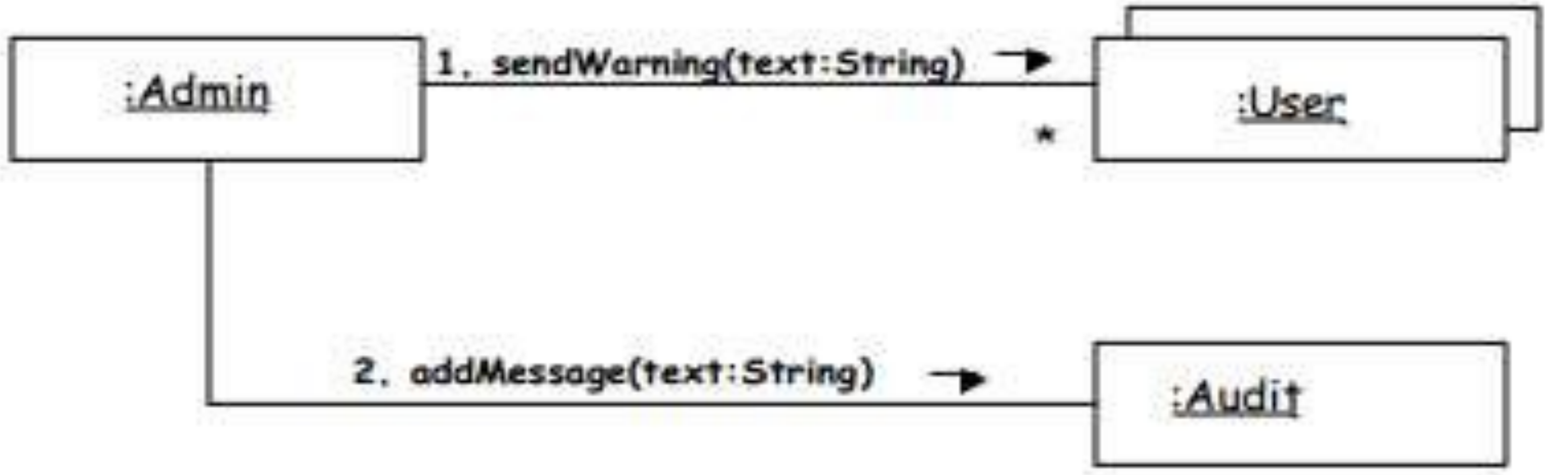
Communication Diagram
Random Stuff

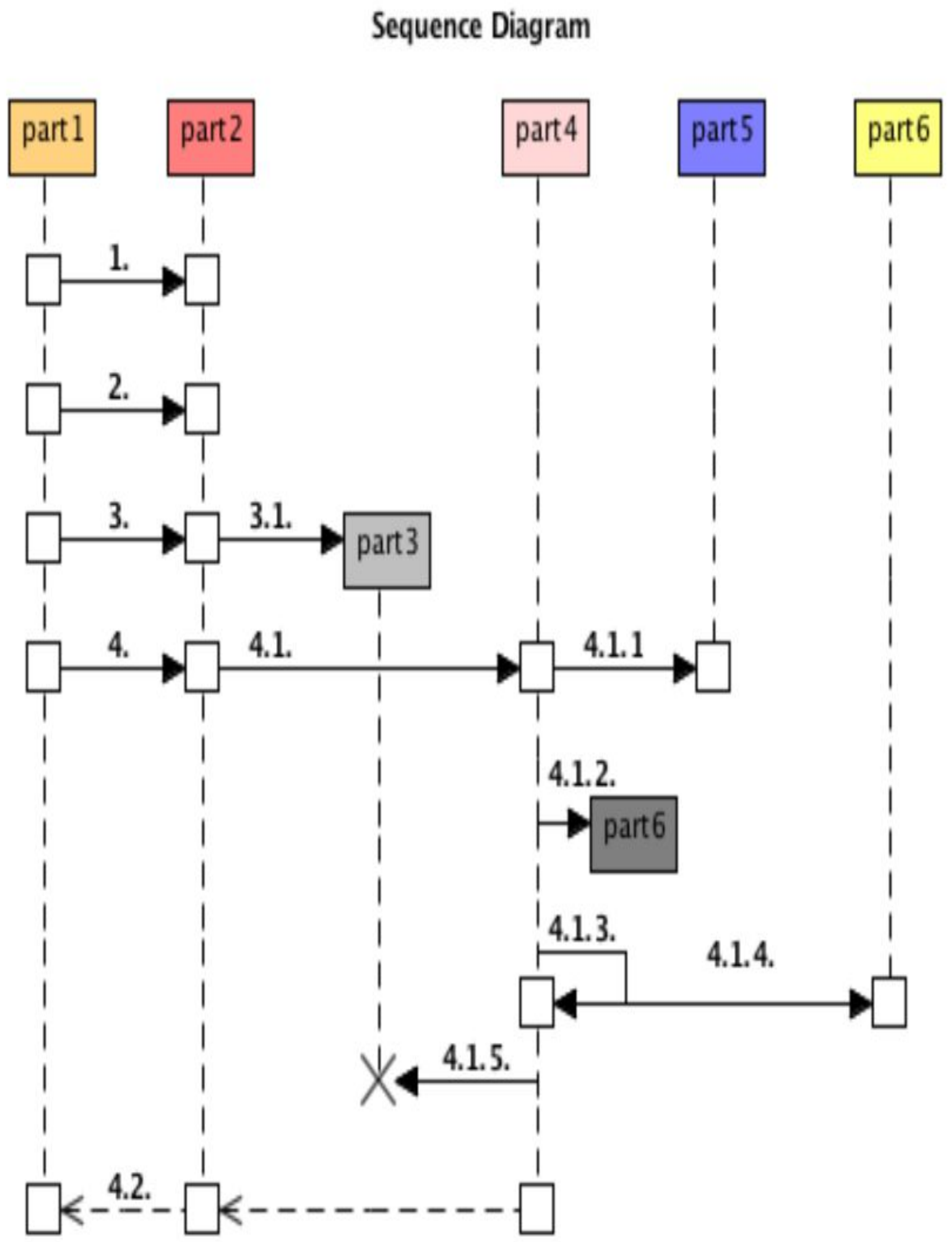


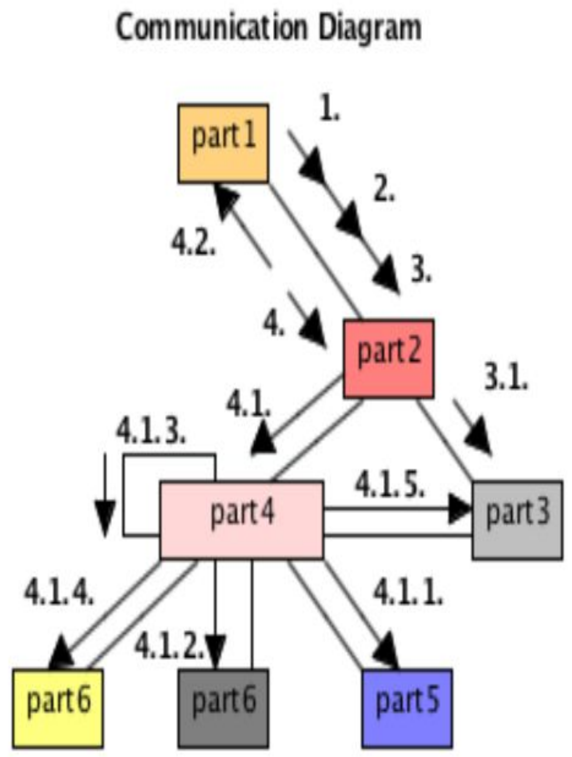
Multiplicity in Collaboration Diagrams

How can messages sent to a multiplicity (e.g., an array) of objects be represented?

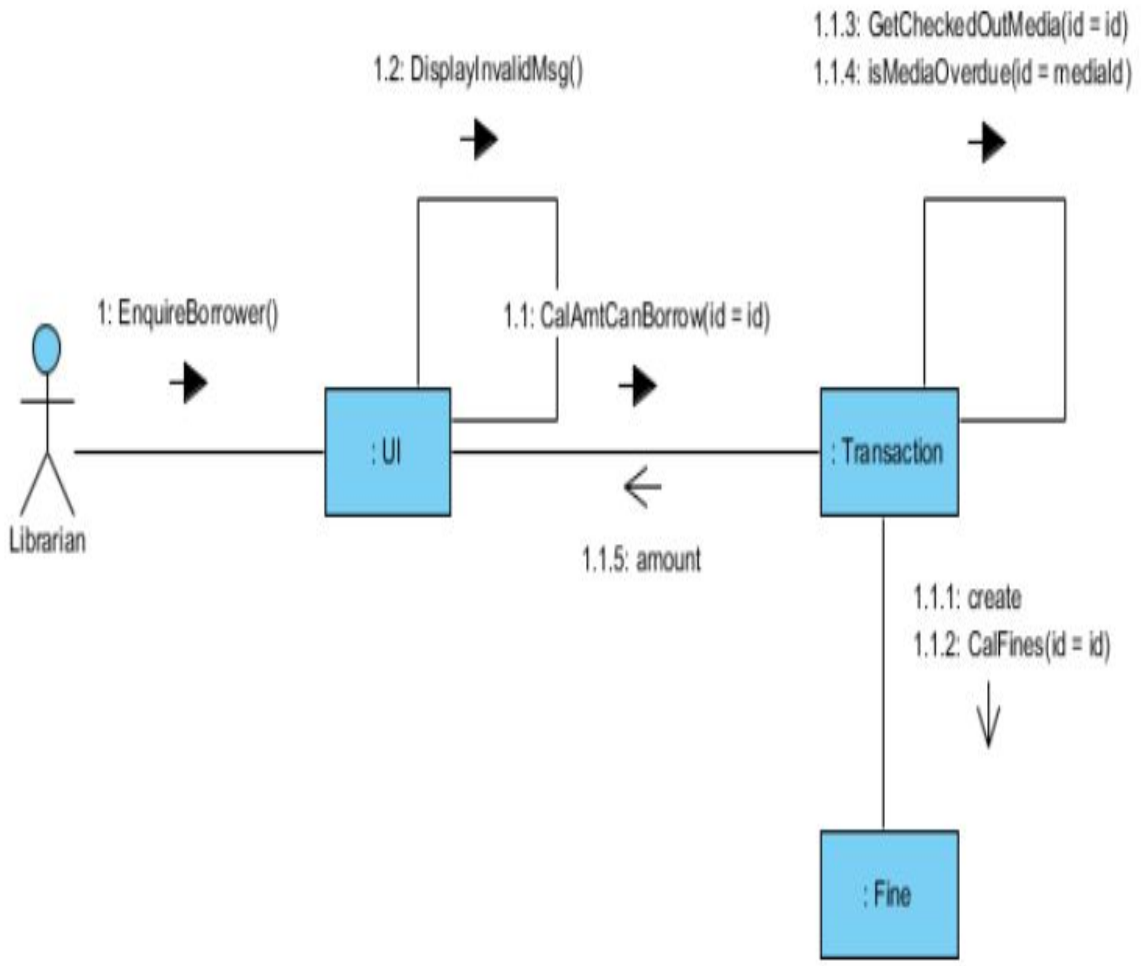
A server object needs to send a message to each user logged in, and another message to the audit object (which keeps a track of messages sent)







I. 1st Message after a new Participant add a Number
II. Every message after increments the number



1. EnquireBorrower

1.1 CalAmtCanBorrow

1.1.1 create

1.1.2 CalFines

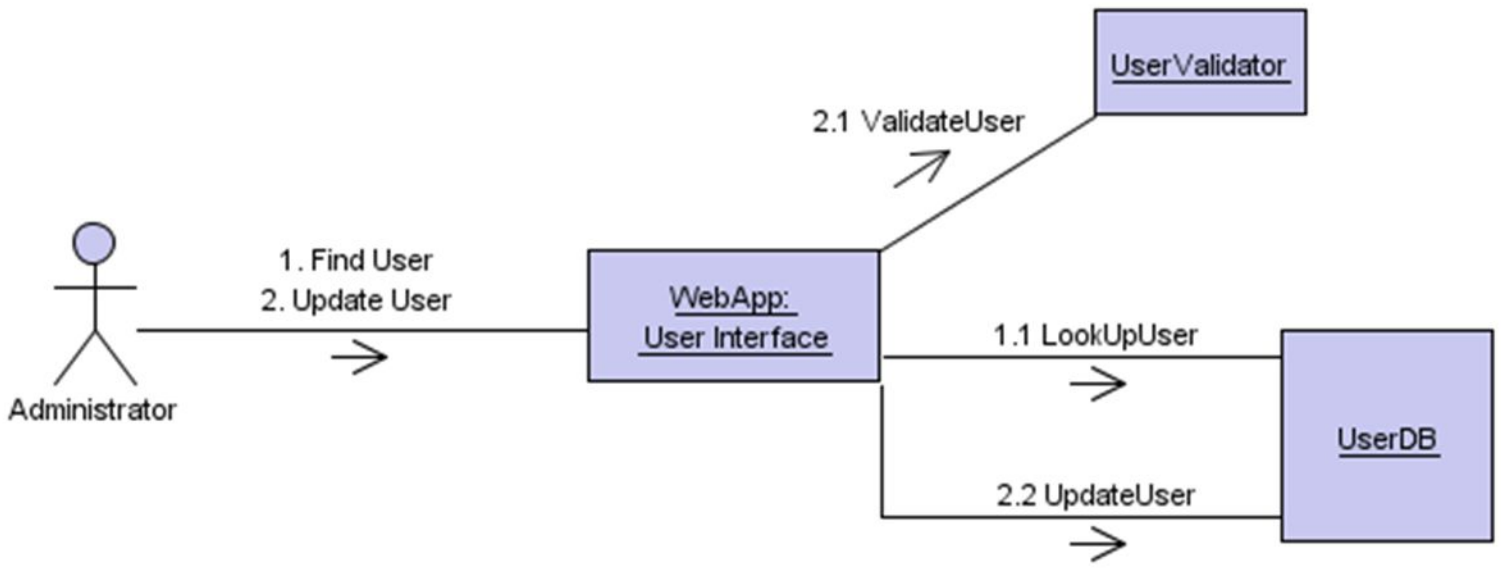
1.1.3 GetCheckedOutMedia

1.1.4 IsMediaOverdue

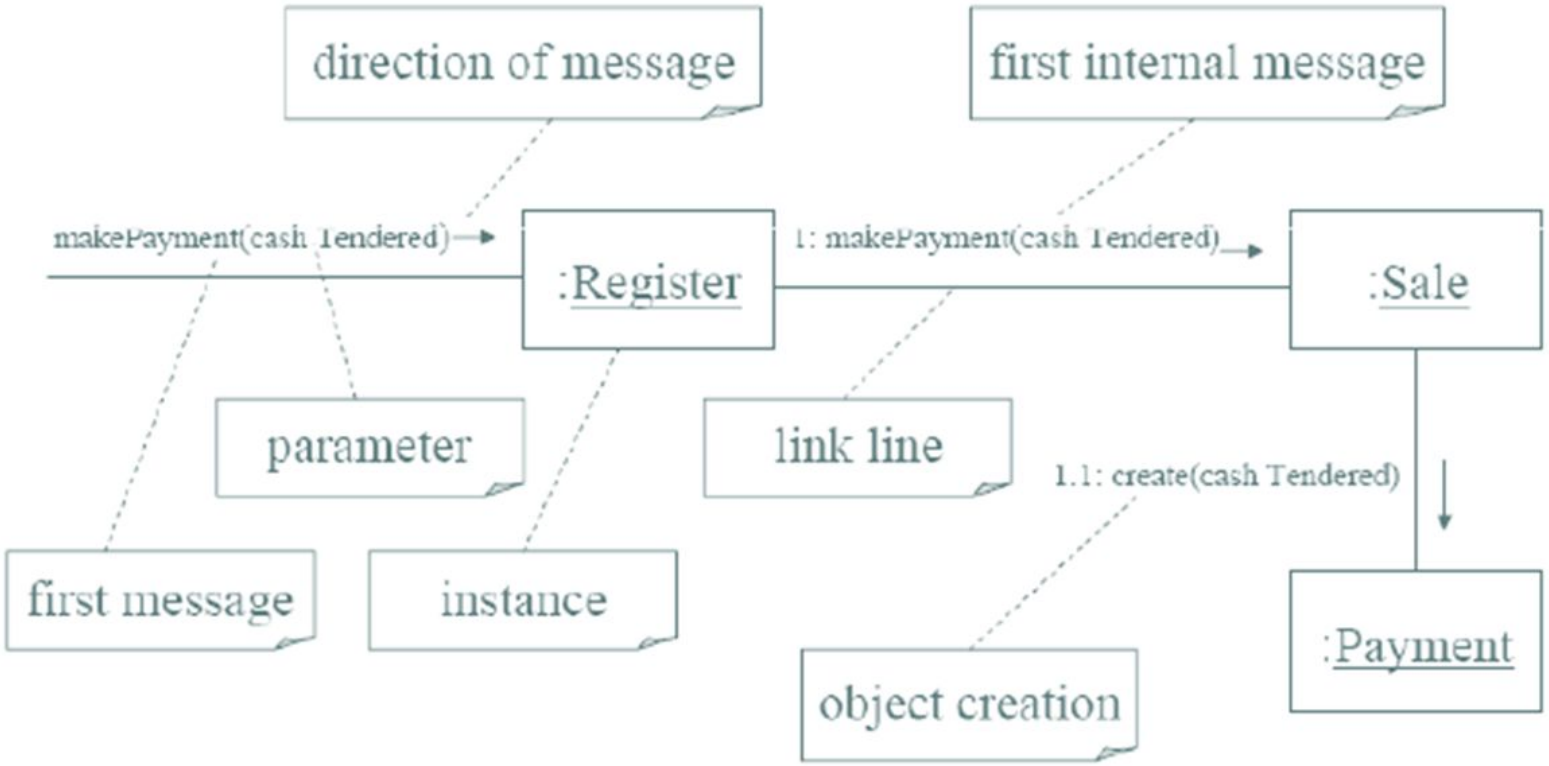
1.1.5 amount

1.2 DisplayInvalidMsg

Collaboration diagram – an example

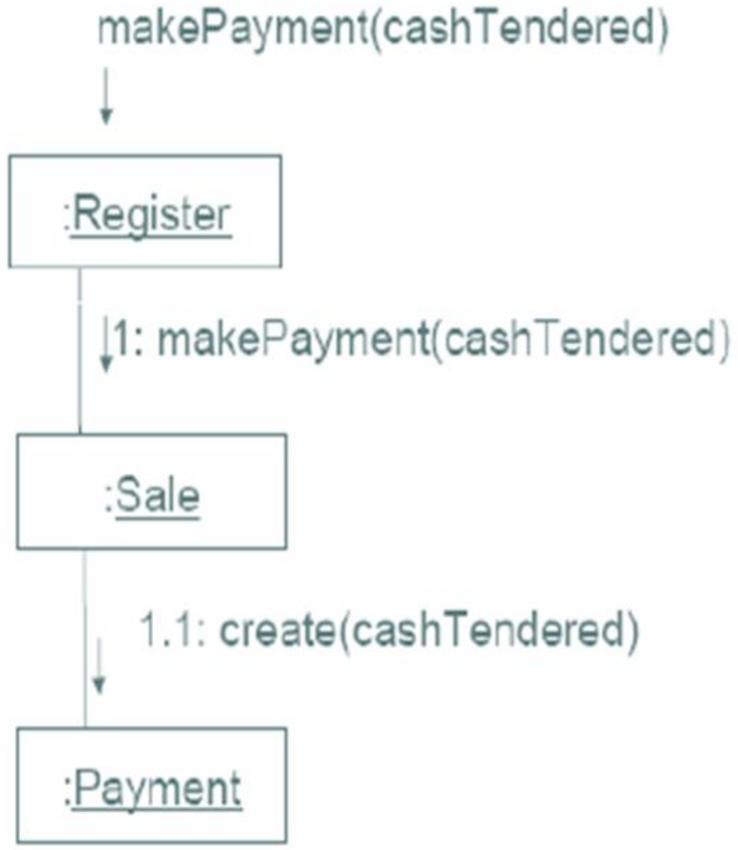


Example of Collaboration Diagram "Make Payment"



How to read “Make Payment” Collaboration Diagram

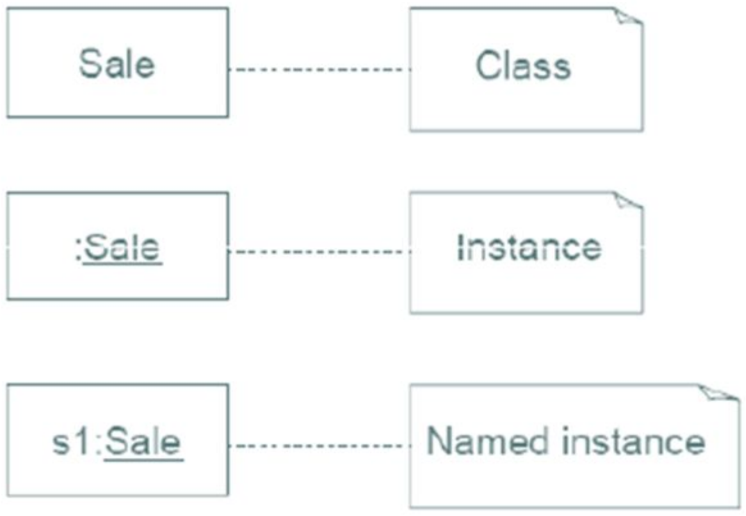
- The message makePayment is sent to an instance of **Register**. The sender is not identified.
- The **Register** instance sends the makePayment message to a **Sale** instance.
- The **Sale** instance creates an instance of a Payment.


makePayment(cashTendered)
:Register
1: makePayment(cashTendered)
:Sale
1.1: create(cashTendered)
:Payment

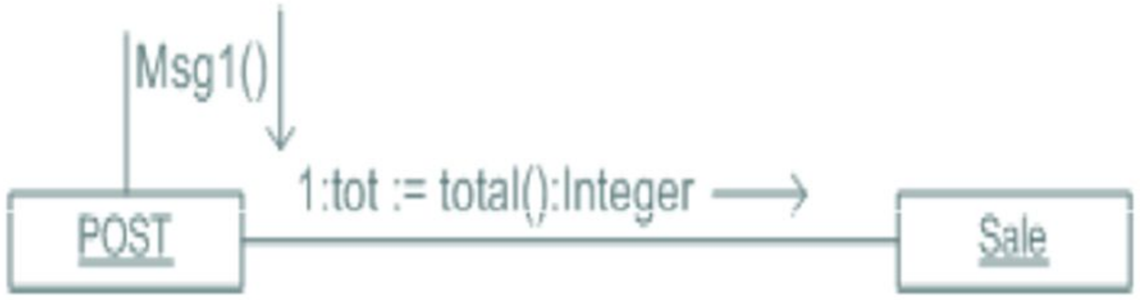
Common Interaction Diagram Notation Illustrating Classes and Instances:

- To show an instance of a class, the regular class box graphic symbol is used, but the name is underlined. Additionally a class name should be preceded by a colon.

- An instance name can be used to uniquely identify the instance.



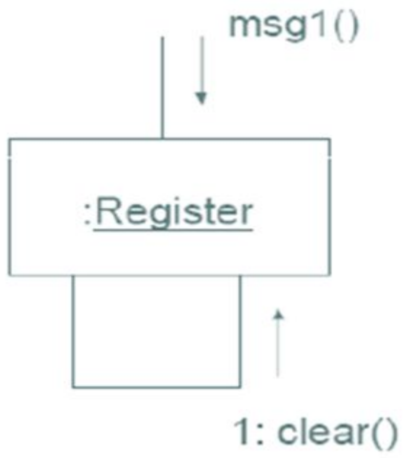
Collaboration Diagram Notation: Links, Messages and Return Value



Collaboration Diagram Notation: Messages to “Self” or “This”

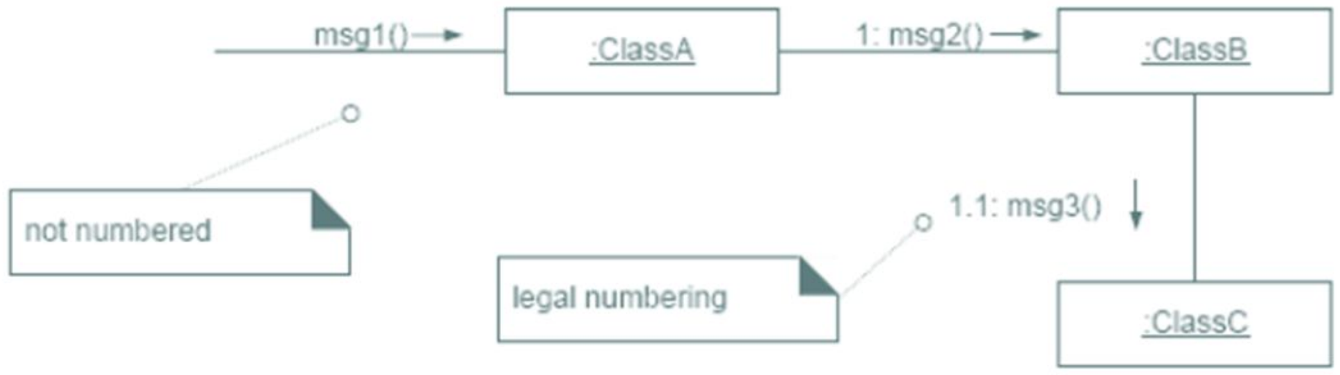
- A message can be sent from an object to itself.

- This is illustrated by a link to itself, with messages flowing along the link.

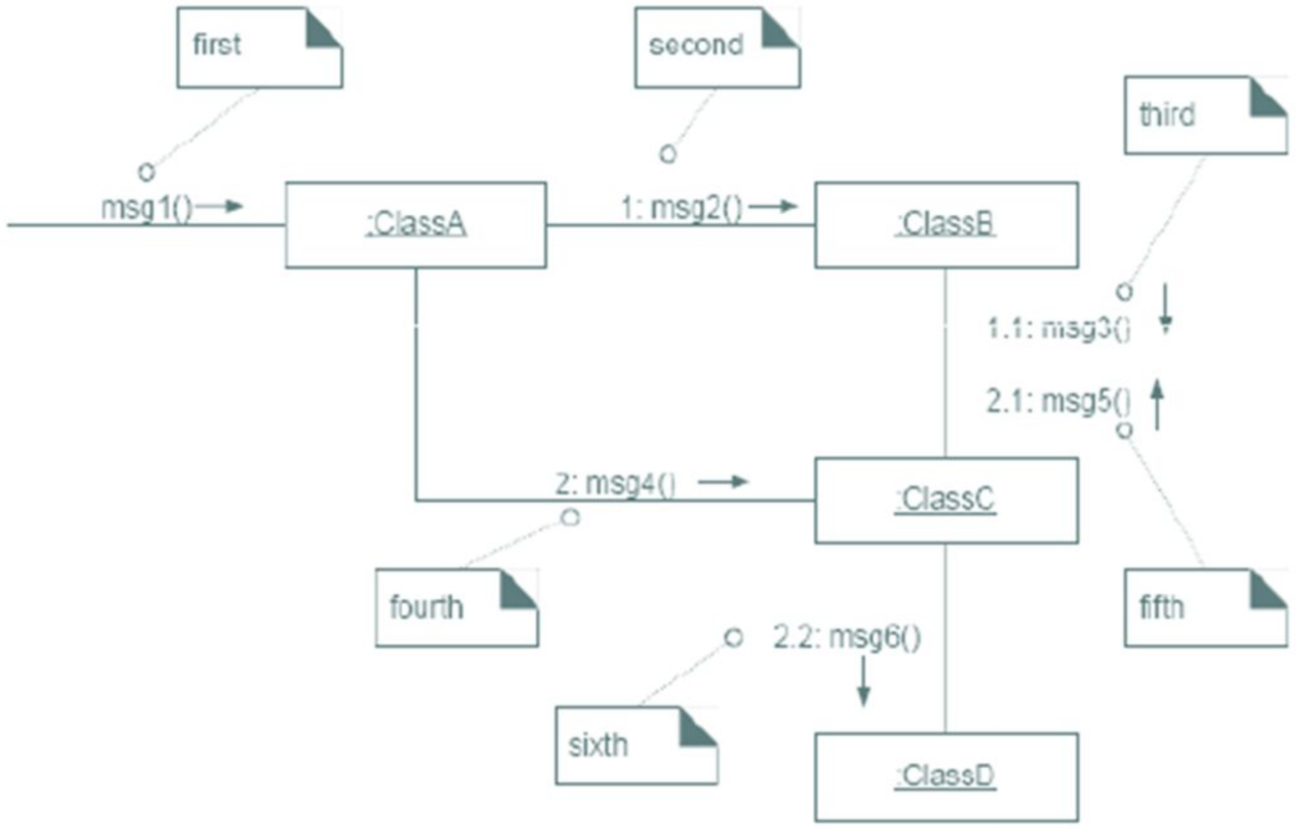


Collaboration Diagram Notation: Message Number Sequencing

- The first message is not numbered.
- The order and nesting of subsequent messages is shown with legal numbering scheme.



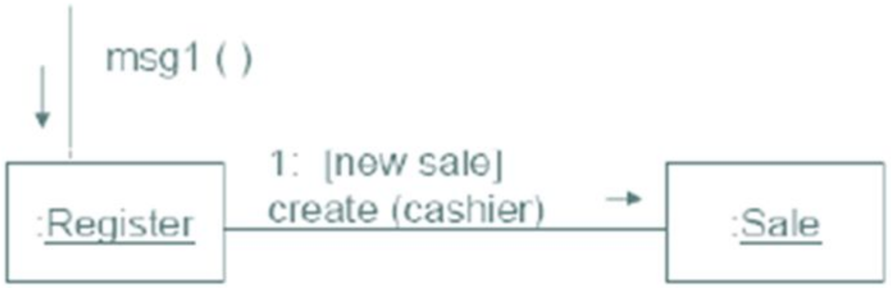
Continue…


first
second
third
msg1()
:ClassA
1: msg2()
:ClassB
1.1: msg3()
2.1: msg5()
2: msg4()
:ClassC
fourth
fifth
2.2: msg6()
sixth
:ClassD

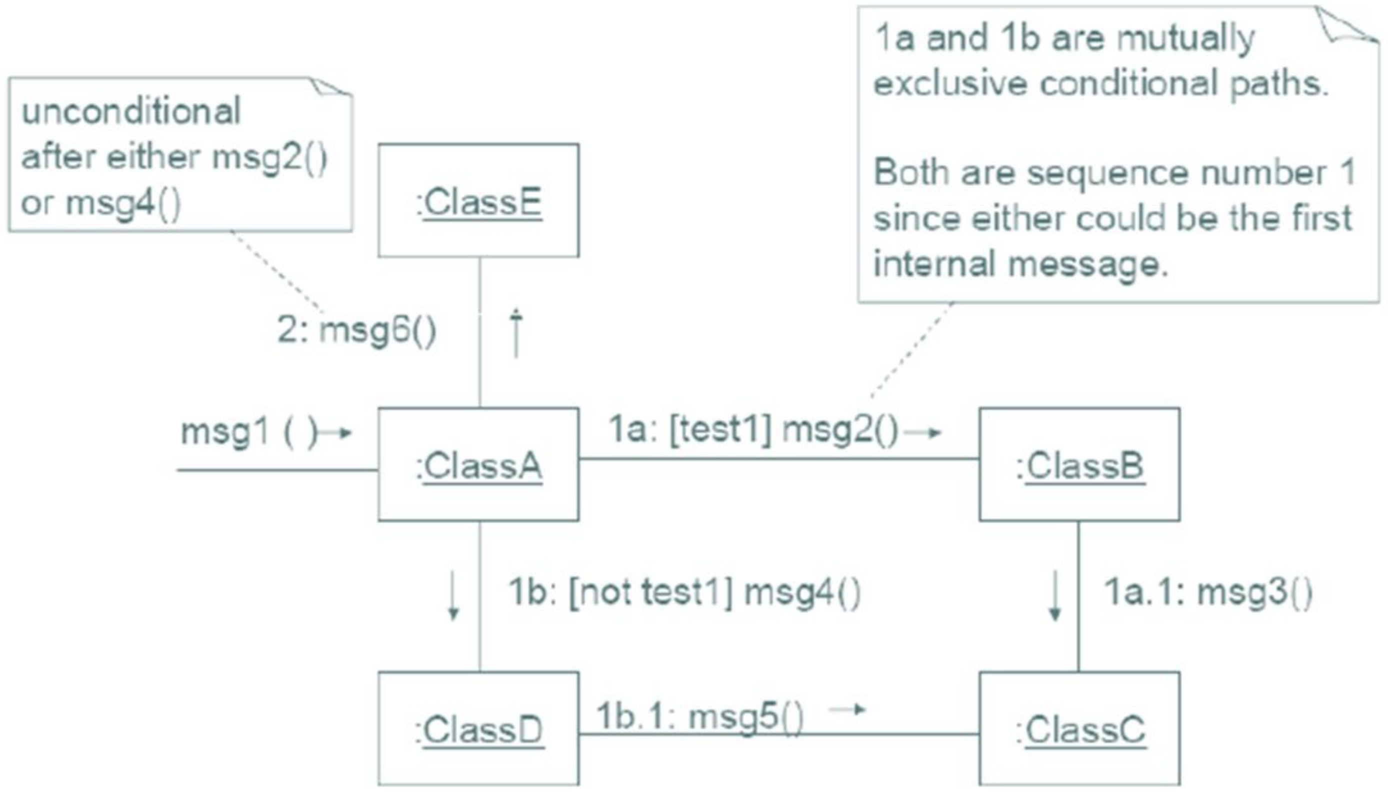
Collaboration Diagram Notation: Conditional Messages

- A conditional message is shown by following a sequence number with a conditional clause in square brackets, similar to the iteration clause.

- The message is sent only if the clause evaluates to true

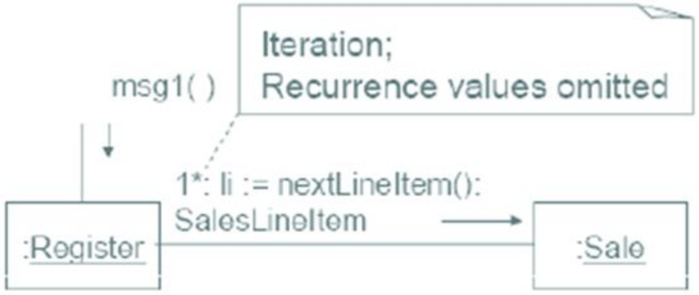


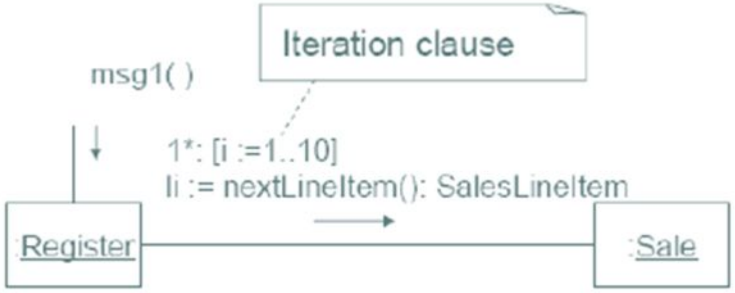
Collaboration Diagram Notation: Mutually Exclusive Conditional Paths


unconditional after either msg2() or msg4()
1a and 1b are mutually exclusive conditional paths.
Both are sequence number 1 since either could be the first internal message.
:ClassE
2: msg6()
msg1 ( )
:ClassA
1a: [test1] msg2()
:ClassB
1b: [not test1] msg4()
1a.1: msg3()
:ClassD
1b.1: msg5()
:ClassC

Collaboration Diagram Notation: Iteration or Looping

- Iteration is indicated by following the sequence number with a star *.
- This expresses that the message is being sent repeatedly, in a loop, to the receiver.
- It is also possible to include an iteration clause indicating the recurrence values.



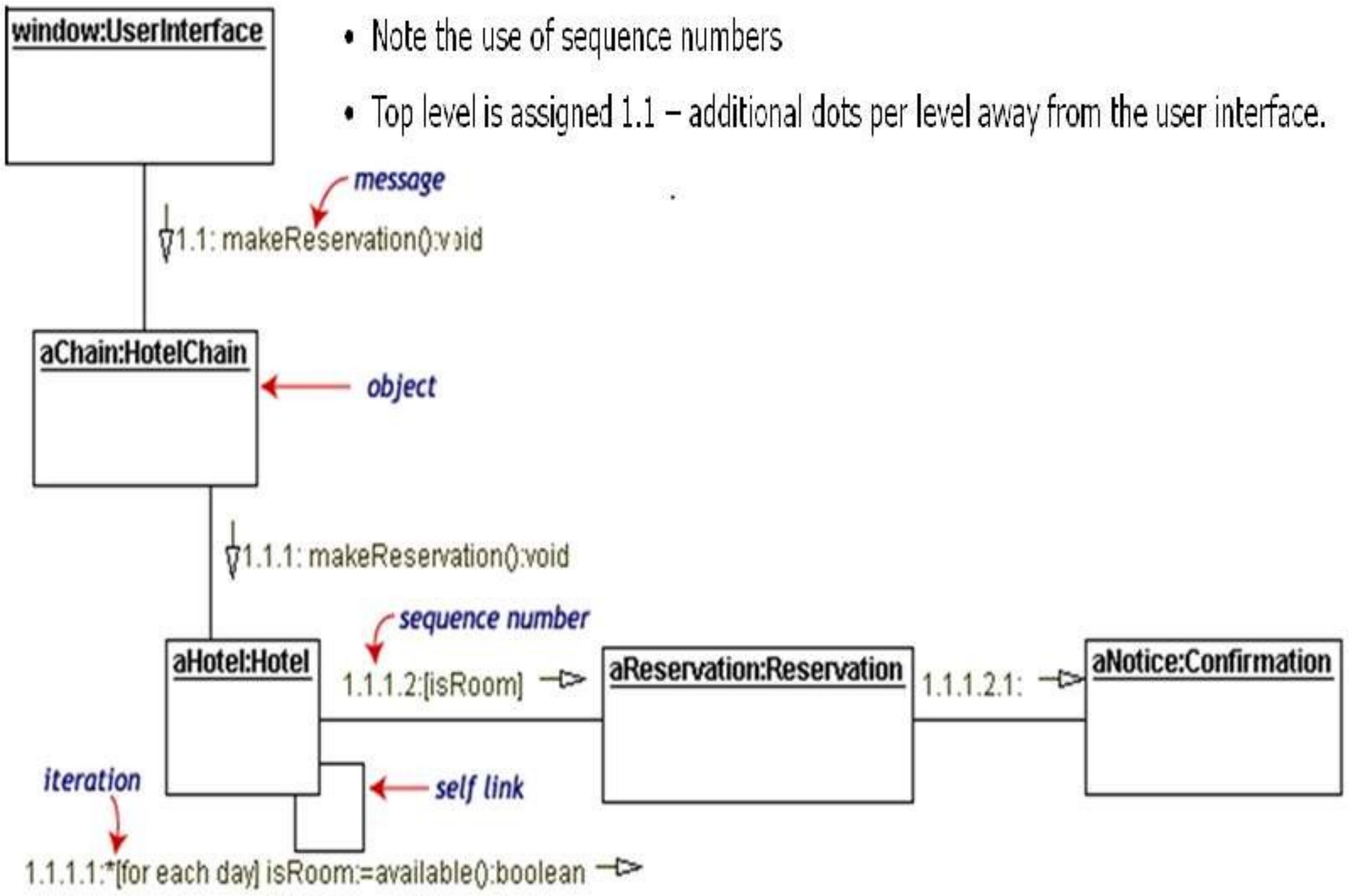


Collaboration Diagrams

- CollaborationsDiagrams show transient links that exists between objects.
  - <<self>> - A message from object to itself
  - << local>> - A message sent due to the object begin defined as a local variable in the method.
  - <<parameter>> - The object reference was sent as a parameter to the method.
  - <<global>> The object is global.

Collaboration – Hotel reservation system

- Note the use of sequence numbers
- Top level is assigned 1.1 – additional dots per level away from the user interface.



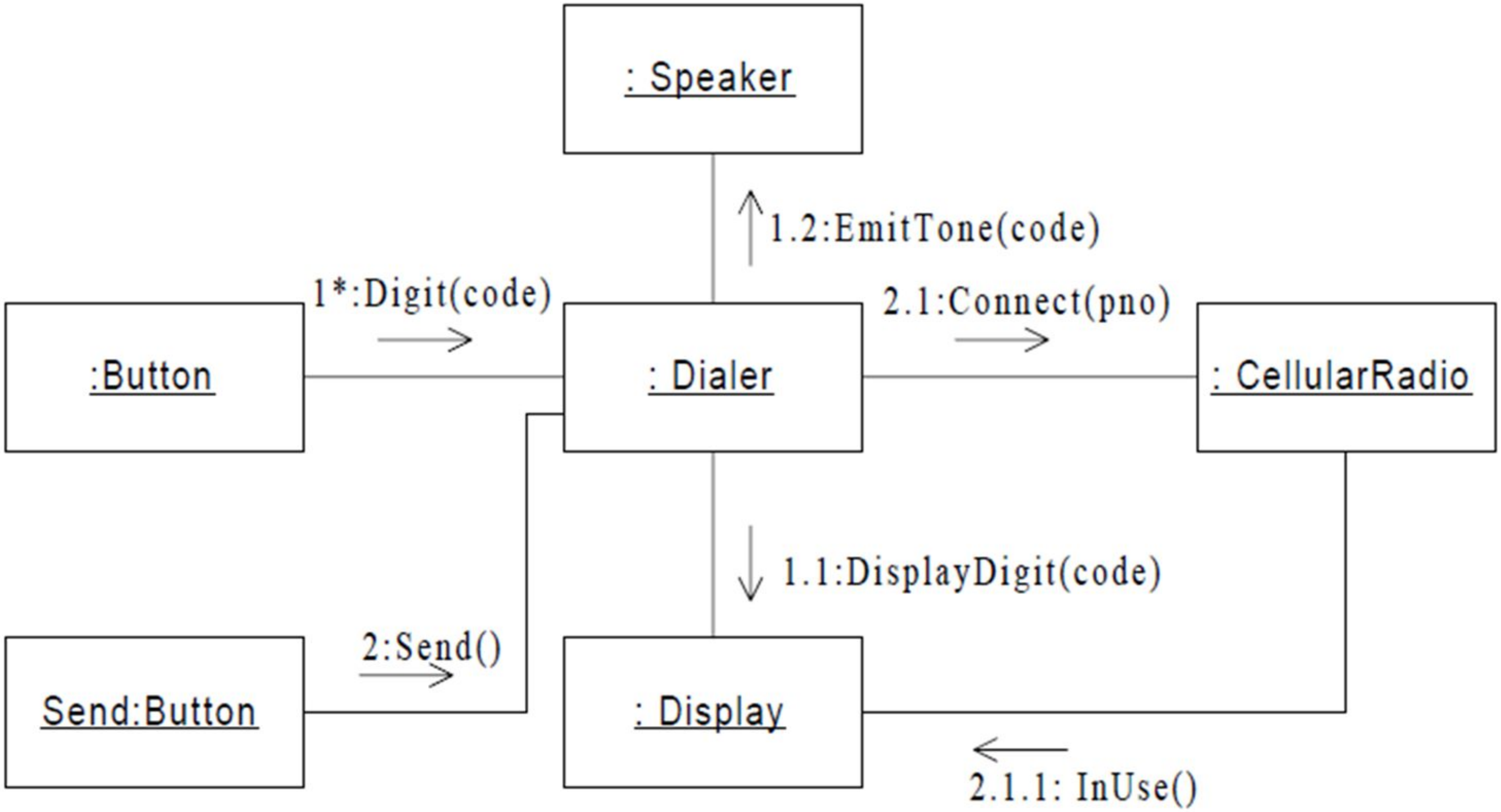
Example: A Cellular Phone

- Consider the software that controls a very simple cellular telephone. Such a phone has buttons for dialing digits, and a "send" button for initiating a call. It has "dialer" hardware and software that gathers the digits to be dialed and emits the appropriate tones. It has a cellular radio that deals with the connection to the cellular network. It has a microphone, a speaker, and a display

Use case: Make Phone Call

1.User presses the digit buttons to enter the phone number.
2. For each digit, the display is updated to add the digit to the phone number.
3. For each digit, the dialer generates the corresponding tone and emits it from the speaker.
4. User presses “Send”
5. The “in use” indicator is illuminated on the display
6. The cellular radio establishes a connection to the network.
7.The accumulated digits are sent to the network.
8. The connection is made to the called party.

Collaboration diagrams of **Make Phone Call** use case.



Collaboration diagrams - strenghts and weaknesses

- One of the greatest strengths of collaboration diagrams is their simplicity.

- The principal weakness, however, is that although they are good at describing behavior, they do not define it. They typically do not show all the iteration and control that is needed to give an computationally complete description.