Interactions Diagrams

Object Interactions
Sequence Diagrams
Communication (Collaboration) Diagrams
Examples
## Use Case Realization

| Use-Case Realization | A **use-case realization** describes how a particular use case is realized within the design model, in terms of collaborating objects. |

A Use-Case Realization represents the Design perspective of a Use Case. The reason for separating the Use-Case Realization from its Use Case is that doing so allows the Use Cases to be managed separately from their realizations.
For each Use Case – One Realization

For each use case in the use-case model, there is a use-case realization in the design model with a dependency (stereotyped <<realizes>>) to the use case.

Use case realization is an organization model element used to group a number of artifacts related to the design of a use case:

- **Class diagrams** of participating classes and subsystems, and
- **Interaction diagrams** which illustrate the flow of events of a use case, performed by a set of class and subsystem instances.
Interaction Diagrams

For each use-case realization there is one or more interaction diagrams depicting its participating objects and their interactions. There are two types of interaction diagrams:

1. **Sequence diagrams** - show the explicit sequence of messages and are better for real-time specifications and for complex scenarios;

2. **Communications (prior UML 2.0 – collaboration) diagrams** - show the communication links between objects and are better for understanding all of the effects on a given object and for algorithm design.
Kept in a Picture

Use Case Realization

Class Diagrams

Interaction Diagrams

Sequence Diagrams

Communication Diagrams
# Models and views in UML 1.5

<table>
<thead>
<tr>
<th></th>
<th>Use case view</th>
<th>Logical view</th>
<th>Implementation view</th>
<th>Process view</th>
<th>Deployment view</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use case diagram</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class diagram</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence diagram</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collaboration diagram</td>
<td>YES</td>
<td></td>
<td></td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Statechart diagram</td>
<td>YES</td>
<td></td>
<td></td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Component diagram</td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Deployment diagram</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
</tr>
</tbody>
</table>
A sequence diagram describes a pattern of interaction among objects, arranged in a chronological order.
Properties of Sequence Diagrams

- Sequence diagrams show the objects participating in the interaction by their "lifelines" and the messages that they send to each other, i.e. how objects interact to perform the behavior of a use case.

- Sequence diagrams are particularly important to designers because they clarify the roles of objects in a flow and thus provide basic input for determining class responsibilities and interfaces.

- Unlike a communication (before UML 2.0 – collaboration) diagram, a sequence diagram includes chronological sequences (explicit sequence of messages), but does not include object relationships.
Contents of Sequence Diagrams

You can have:

*objects* (i.e. class instances)

and

*actor instances*

in sequence diagrams, together with *messages* describing how they interact.

The diagram describes what takes place in the participating objects, in terms of activations, and how the objects communicate by sending messages.
A sequence diagram describing part of the flow of events of the use case **Place Local Call** in a simple **Phone Switch**.
Contents of Sequence Diagrams - 2

*Objects* - shown as a vertical dashed line called the "lifeline". The lifeline represents the existence of the object at a particular time. An object symbol shows the name of the object and its class underlined:

```
objectname  [selector] : classname ref decomposition
```

*Actors* - try keeping them either at the left-most, or the right-most lifelines
**Messages** - communications between objects that conveys information with the expectation that activity will ensue; shown as a horizontal solid arrow from the lifeline of one object to the lifeline of another object. The arrow is labeled with the name of the message, and its parameters, or with a sequence number.

**Scripts** - describe the flow of events textually in a sequence diagram.
## Object/class naming convention

<table>
<thead>
<tr>
<th>Example participant name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>admin</td>
<td>A part is named admin, but at this point in time the part has not been assigned a class.</td>
</tr>
<tr>
<td>: ContentManagementSystem</td>
<td>The class of the participant is ContentManagementSystem, but the part currently does not have its own name.</td>
</tr>
<tr>
<td>admin : Administrator</td>
<td>There is a part that has a name of admin and is of the class Administrator.</td>
</tr>
<tr>
<td>eventHandlers [2] : EventHandler</td>
<td>There is a part that is accessed within an array at element 2, and it is of the class EventHandler.</td>
</tr>
<tr>
<td>: ContentManagementSystem ref cmsInteraction</td>
<td>The participant is of the class ContentManagementSystem, and there is another interaction diagram called cmsInteraction that shows how the participant works internally.</td>
</tr>
</tbody>
</table>
Interactions on a sequence diagram are shown as messages between participants (from Learning UML 2.0, by K. Hamilton, R. Miles)
The Message Signature

- **Signature:**
  \[ \text{attribute} = \text{signal\_or\_message\_name} (\text{arguments}) : \text{return\_type} \]

- **Arguments:**
  \[ \text{name:} \text{type}, \ldots \]

- Only **signal\_or\_message\_name** is not optional

- **Example:**
  \[ \text{myVar} = \text{sendSignal}() : \text{ReturnClass} \]

  The message's name is **sendSignal**; no arguments; returns an object of class **ReturnClass** that is assigned to the **myVar** attribute of the **message caller**.
Five main types of messages

- A Synchronous Message
- An Asynchronous Message
- A Return Message
- A Participant Creation Message
- A Participant Destruction Message
Message Synchronization

- Synchronous message call -> method invocation
- Asynchronous message call -> method call in another thread:

```java
public void operation1() {
    // Receive the message and trigger off the thread
    Thread myWorker = new Thread(this);
    myWorker.start();
    // This call starts a new thread, calling the run() thread method
    // As soon as the thread has been started, the call returns.
}
```
Example

- The first message is a synchronous message (denoted by the solid arrowhead) complete with an implicit return message;
- The second message is asynchronous (denoted by line arrowhead), and the third is the asynchronous return message (denoted by the dashed line).
A lifeline may be created or destroyed during the timescale represented by a sequence diagram. In the latter case, the lifeline is terminated by a stop symbol, represented as a cross. In the former case, the symbol at the head of the lifeline is shown at a lower level down the page than the symbol of the object that caused the creation.
Create and Destroy Messages
(from Learning UML 2.0, by K. Hamilton, R. Miles)
Nested and Self Messages
Self Message

- A self message can represent a recursive call of an operation, or one method calling another method belonging to the same object. It is shown as creating a nested focus of control in the lifeline’s execution occurrence.
Lost and Found Messages (UML 2.0)

- Lost messages are those that are either sent but do not arrive at the intended recipient, or which go to a recipient not shown on the current diagram. Found messages are those that arrive from an unknown sender, or from a sender not shown on the current diagram. They are denoted going to or coming from an endpoint element.
Duration and Time Constraints (UML 2.0)

- By default, a message is shown as a horizontal line. Since the lifeline represents the passage of time down the screen, when modelling a real-time system, or even a time-bound business process, it can be important to consider the length of time it takes to perform actions. By setting a duration constraint for a message, the message will be shown as a sloping line.
UML 2.0 Fragments

SystemRegisterSequence

User
UserInterface
UserWorkflow
UsersEntity

Tools
Point Eraser
Sweeper
Gesture Pen

Sequence
LifeLine
Message
Duration Message
Create Message
Self Message
Recursive Message
Found Message
Lost Message
Alternative Combined Fragment

1: register()
2: displayRegistrationForm()
3: enterRegistrationData()
4: saveRegistrationData()
5: saveRegistrationData()
6: confirmRegistration()
7: displayConfirmationMsg()
About fragments

- Fragments allow for adding a degree of procedural logic to diagrams and which come under the heading of combined fragments.

- A combined fragment is one or more processing sequence enclosed in a frame and executed under specific named circumstances. The fragments available are:
  - **Alternative fragment** (denoted “alt”) models if…then…else constructs.
  - **Option fragment** (denoted “opt”) models switch constructs.
  - **Break** fragment models an alternative sequence of events that is processed instead of the whole of the rest of the diagram.
Cont.

- Parallel fragment (denoted “par”) models concurrent processing.
- Weak sequencing fragment (denoted “seq”) encloses a number of sequences for which all the messages must be processed in a preceding segment before the following segment can start, but which does not impose any sequencing within a segment on messages that don’t share a lifeline.
- Negative fragment (denoted “neg”) encloses an invalid series of messages.
- Critical fragment encloses a critical section.
- Ignore fragment declares a message or message to be of no interest if it appears in the current context.
- Consider fragment is in effect the opposite of the ignore fragment: any message not included in the consider fragment should be ignored.
Loop fragment encloses a series of messages which are repeated.
Gate (UML 2.0)

- A gate is a connection point for connecting a message inside a fragment with a message outside a fragment. EA shows a gate as a small square on a fragment frame. Diagram gates act as off-page connectors for sequence diagrams, representing the source of incoming messages or the target of outgoing messages.
Part Decomposition (UML 2.0)

- An object can have more than one lifeline coming from it.
- This allows for inter- and intra-object messages to be displayed on the same diagram.
State Invariant (UML 2.0)

- A state invariant is a constraint placed on a lifeline that must be true at run-time.
- It is shown as a rectangle with semi-circular ends.
Sequence diagram for ReportEmergency [Bruege&Dutoit]
Figura 5-9  Diagrama de secuencia para el caso de uso ReportEmergency-2 [Bruege&Dutoit]
Sequence diagram for ReportEmergency-3 [Bruege&Dutoit]
Centralized control of a flow of events means that a few objects steer the flow by sending messages to, and receiving messages from other objects. These controlling objects decide the order in which other objects will be activated in the use case. Interaction among the rest of the objects is very minor or does not exist.

Main advantage: each object does not have to keep track of the next object's tally. To change the order of the sub-event phases, you merely make the change in the control object. Another advantage to this structure is that you can easily reuse the various sub-event phases in other use cases because the order of behavior is not built into the objects.
Decentralized control arises when the participating objects communicate directly with one another, not through one or more controlling objects.
Centralized Control - Example

In the **Recycling-Machine System**, the use case **Print Daily Report** keeps track of - among other things - the number and type of returned objects, and writes the tally on a receipt. The **Report Generator** control object decides the order in which the sums will be extracted and written.
Decentralized Control - Example

In the use case **Send Letter** someone mails a letter to another country through a post office. The letter is first sent to the country of the addressee. In the country, the letter is sent to a specific city. The city, in turn, sends the letter to the home of the addressee.

The sub-event phases belong together. The sender of the letter speaks of "sending a letter to someone."
Control Flow – How to be used?

A decentralized structure is appropriate:

• If the sub-event phases are tightly coupled:
  • Form a part-of or consists-of hierarchy, such as Country - State - City;
  • Form an information hierarchy, such as CEO - Division Manager - Section Manager;
  • Represent a fixed chronological progression (the sequence of sub-event phases will always be performed in the same order), such as Advertisement - Order - Invoice - Delivery - Payment; or
  • Form a conceptual inheritance hierarchy, such as Animal - Mammal - Cat.
• If you want to encapsulate, and thereby make abstractions of, functionality.
A *centralized* structure is appropriate:

• If the order in which the sub-event phases will be performed is likely to change.
• If you expect to insert new sub-event phases.
• If you want to keep parts of the functionality reusable as separate pieces.
A communication (collaboration) diagram describes a pattern of objects interaction; it shows the objects participating in the interaction by their links to each other and the messages sent to each other.

Unlike a sequence diagram, a collaboration diagram shows the *relationships* among the objects.

You can have **objects** and **actor instances** in collaboration diagrams, together with links and **messages** describing how they are related and how they interact. The diagram describes what takes place in the participating objects, in terms of how the objects communicate by sending messages to one another.
Communication vs Sequence Diagrams

- Communication diagrams are especially good at showing which links are needed between participants to pass an interaction's messages.
- On a sequence diagram, the links between participants are implied by the fact that a message is passed between them. Communication diagrams provide an intuitive way to show the links between participants that are required for the events that make up an interaction (the order of the events involved in an interaction is secondary).
- Sequence and communication diagrams are so similar that most UML tools can automatically convert from one diagram type to the other.
• A link is a relationship among objects across which messages can be sent (shown as a solid line between two objects).

• An object interacts with, or navigates to, other objects through its links to them.

• A link can be an instance of an association, or it can be anonymous, meaning that its association is unspecified.
Example of Collaboration Diagram

Events’ flow of the use case *Receive Deposit Item* in the *Recycling-Machine System*
Contents of Collaboration Diagrams - Messages

• A message is a communication between objects that conveys information with the expectation that activity will ensue.

• In collaboration diagrams, a message is shown as a labeled arrow placed near a link. This means that the link is used to transport, or otherwise implement the delivery of the message to the target object.

• The arrow points along the link in the direction of the target object (the one that receives the message).

• The arrow is labeled with the name of the message, and its parameters. The arrow may also be labeled with a sequence number to show the sequence of the message in the overall interaction.

• A message can be unassigned, meaning that its name is a temporary string that describes the overall meaning of the message. You can later assign the message by specifying the operation of the message's destination object.
Boundary Objects in Interaction Diagr.

To illustrate the boundary objects participating in the use-case storyboard (a logical and conceptual description of how a use case is provided by the user interface), and their interactions with the user, we use collaboration or sequence diagrams. This is useful for use cases with complex sequences or flows of events.

Example: Class diagram including the Mail User actor and the boundary classes Mail Box, Mail Message, and Attachment, realizing *Manage Incoming Mail Messages use case.*
Boundary Object Collaboration Diagrams - Example

Collaboration diagram including the Mail User actor and boundary objects of Mail Box, Mail Message, and Attachment, participating in a use-case storyboard realizing the *Manage Incoming Mail Messages* use case:

1: start by showing mail messages
2: arrange by criteria (sender, subject)
7: quit managing incoming mail messages

3: read the mail message
4: save the mail message
5: save attachments

6: save attachment
Message Sending based on Conditions

- 1. `messageA()` will be invoked 10 times before the other
- If `condition == true` returns `false`, then 2a. `messageB()` is not invoked
Comparing sequence and communication diagrams
(from Learning UML 2.0, by K. Hamilton, R. Miles)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Sequence diagrams</th>
<th>Communication diagrams</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shows participants effectively</td>
<td>Participants are mostly arranged along the top of page, unless the drop-box participant creation notation is used. It is easy to gather the participants involved in a particular interaction.</td>
<td>Participants as well as links are the focus, so they are shown clearly as rectangles.</td>
</tr>
<tr>
<td>Showing the links between participants</td>
<td>Links are implied. If a message is passed from one participant to another, then it is implied that a link must exist between those participants.</td>
<td>Explicitly shows the links between participants. In fact, this is the primary purpose of these types of diagram.</td>
</tr>
<tr>
<td>Showing message signatures</td>
<td>Message signatures can be fully described.</td>
<td>Message signatures can be fully described.</td>
</tr>
<tr>
<td>Supports parallel messages</td>
<td>With the introduction of sequence fragments, sequence diagrams are much better.</td>
<td>Shown using the number-letter notation on message sequences.</td>
</tr>
<tr>
<td>Supports asynchronous messages (fire and forget)</td>
<td>Achieved using the asynchronous arrow.</td>
<td>Communication diagrams have no concept of the asynchronous message since its focus is not on message ordering.</td>
</tr>
<tr>
<td>Easy to read message ordering</td>
<td>This is a sequence diagram's forte. Sequence diagrams clearly show message ordering using the vertical placement of messages down the diagram's page.</td>
<td>Shown using the number-point-nested notation.</td>
</tr>
<tr>
<td>Easy to create and maintain the diagram</td>
<td>Creating a sequence diagram is fairly simple. However, maintaining sequence diagrams can be a nightmare unless a helpful UML tool is being used.</td>
<td>Communication diagrams are simple enough to create; however, maintenance, especially if message numbering needs to be changed, still ideally needs the support of a helpful UML tool.</td>
</tr>
</tbody>
</table>
The University Course Registration (UCR) Case Study

Object Name
History 101-Section2

Object Name & Class
History 101-
Section7:CourseOffering

Class Name
:CourseOffering

Naming Objects in a Sequence Diagram
Sequence Diagram (over) versus Collaboration Diagram (below)
UCR - 3

Sequence Diagram with Objects Assigned to Classes

1: set course info
2: process
3: add course
4: new course
UCR - 4

Collaboration Diagram

1: set course info
2: process
3: add course
4: new course