Interaction Diagram

- As its name might suggest, an interaction diagram is a type of UML diagram that's used to capture the interactive behaviour of a system
- Interaction diagrams are models that describe how a group of objects collaborate in some behaviour - typically in a single use-case.
- The diagrams show a number of example objects and the messages that are passed between these objects within the use-case.
- Interaction diagrams should be used when you want to look at the behaviour of several objects within a single use case.
- They are good at showing the collaborations between the objects, they are not so good at precise definition of the behaviour.

The purpose of interaction diagram is −

- To capture the dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe the structural organization of the objects.
- To describe the interaction among objects.

Types of Interaction Diagram

- **Sequence Diagram** ***(Already Discussed. Note is with You)***
- **Communication Diagram or Collaboration Diagram**

Communication Diagram or Collaboration Diagram:

- A collaboration diagram, also known as a communication diagram, which represents the relationships and interactions among software objects in the Unified Modelling Language (UML).

- These diagrams can be used to portray the dynamic behaviour of a particular use case and define the role of each object.

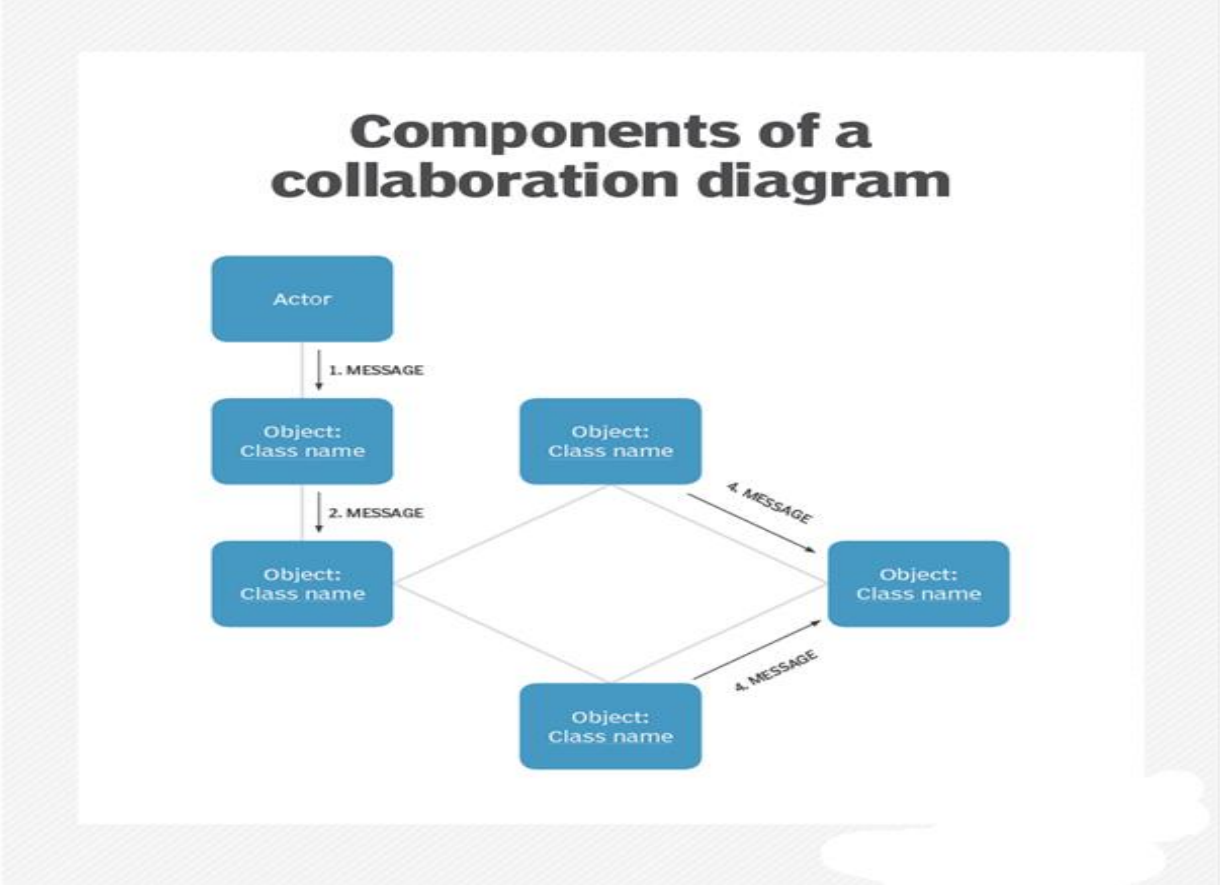
- These diagrams are used to show how objects interact to perform the behaviour of a particular use case, or a part of a use case.
- Along with sequence diagrams, collaboration are used by designers to define and clarify the roles of the objects that perform a particular flow of events of a use case.
- They are the primary source of information used to determining class responsibilities and interfaces.

Notations of a collaboration diagram

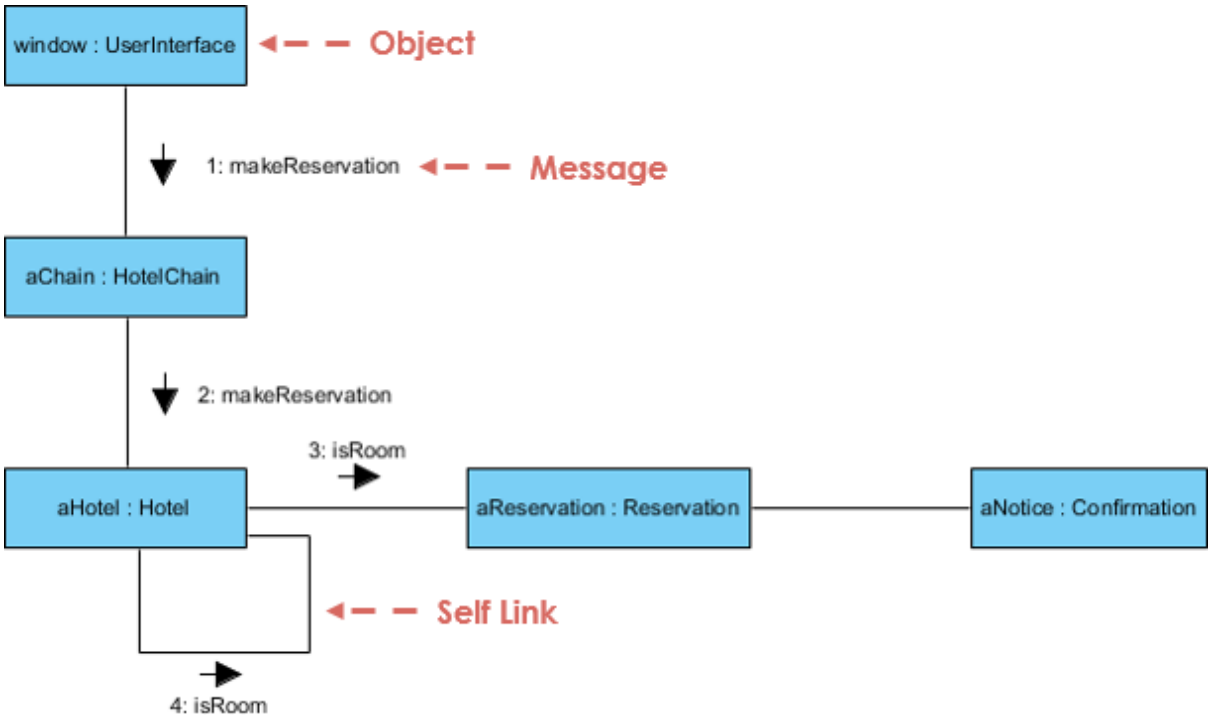
A collaboration diagram shows the roles, functionality and behaviour of individual objects as well as the overall operation of the system in real time.

The four major components of a collaboration diagram are:

1. Objects- Objects are shown as rectangles with naming labels inside. The naming label follows the convention of object name: class name. If an object has a property or state that specifically influences the collaboration, this should also be noted.
2. Actors- Actors are instances that invoke the interaction in the diagram. Each actor has a name and a role, with one actor initiating the entire use case.
3. Links- Links connect objects with actors and are depicted using a solid line between two elements. Each link is an instance where messages can be sent.
4. messages- Messages between objects are shown as a labelled arrow placed near a link. These messages are communications between objects that convey information about the activity and can include the sequence number.

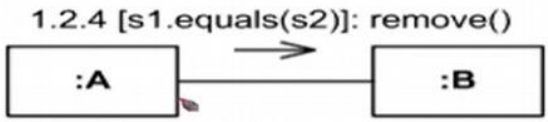


Collaboration Diagram Example is given below:



Messages in Communication (or Collaboration) Diagram

- Message in Communication Diagram is shown as a line with sequence expression and **arrow** above the line
- The arrow indicates direction of the communication

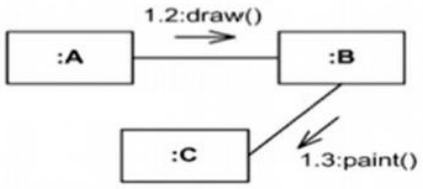


Instance of class A sends remove() message to instance of B if s1 is equal to s2

- 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
- Example: **3b.2.2:m5** : Sequence expression **3b.2.2** and message name **m5**
- Each Sequence term
  sequence-term ::= [integer [name]] [recurrence]
- The **integer** represents the **sequential order** of the message within the next higher level of procedural calling

Sequential order messages

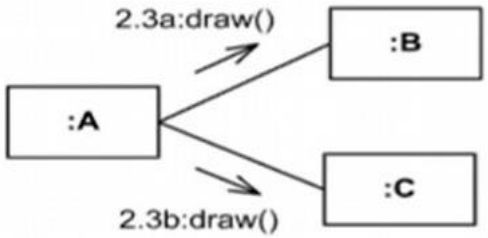
- **Example**:
  - message with sequence 2 follows message with sequence 1
  - 2.1 follows 2
  - 5.3 follows 5.2 within activation 5
  - 1.2.4 follows message 1.2.3 within activation 1.2.



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

Concurrent Messages

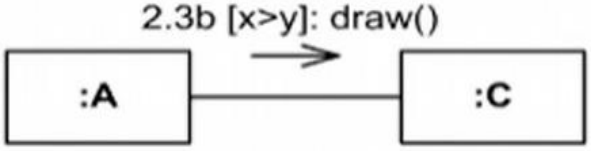
- The **name** represents a **concurrent thread** of control
- **Example**:
  - messages 2.3a and 2.3b are concurrent within activation 2.3
  - 1.1 follows 1a and 1b
  - 3a.2.1 and 3b.2.1 follow 3.2



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

Guard :

- A **guard** specifies condition for the message to be sent (executed) at the given nesting depth
- **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

Recurrence and Iteration

- The **recurrence** defines conditional or iterative execution of zero or more messages that are executed depending on the specified condition
  recurrence ::= branch | loop , branch ::= '[' guard ']'
- An **iteration** specifies a sequence of messages at the given nesting depth
- **Notation**:
  - * : *Messages Executed Sequentially*
  - *|| : *Messages Executed Concurrently*

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

e.g. Online Book Shop Component diagram