G. PULLAIAH COLLEGE OF ENGINEERING AND TECHNOLOGY

Accredited by NAAC with 'A' Grade of UGC, Approved by AICTE, New Delhi
Permanently Affiliated to JNTUA, Ananthapuramu
(Recognized by UGC under 2(f) and 12(B) & ISO 9001:2008 Certified Institution)
Nandikotkur Road, Venkayapalli, Kurnool – 518452

Department of Computer Science and Engineering

Bridge Course
On
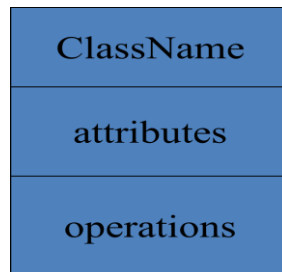# *Software Architecture & Design Patterns*

By

M.Sri Lakshmi

Table of Contents

Software Design

Static Modeling using Unified Modeling Language (UML) :

Classes:

A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

Class Names:

The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.
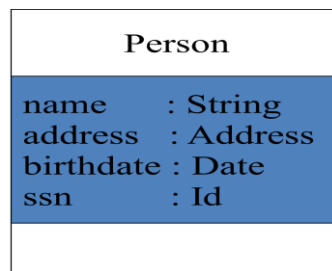
An *attribute* is a named property of a class that describes the object being modeled. In the class diagram, attributes appear in the second compartment just below the name-compartment.

Attributes are usually listed in the form:

attributeName : Type

A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist. For example, a Person's age can be computed from his birth date. A derived attribute is

designated by a preceding '/' as in: / age : Date

Attributes can be:

   + public

protected

- private

/ derived

Class Operations: *Operations* describe the class behavior and appear in the third compartment.

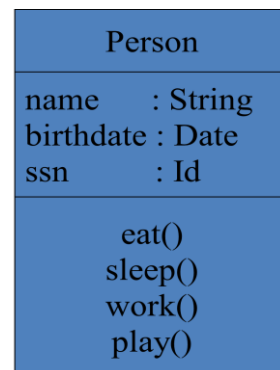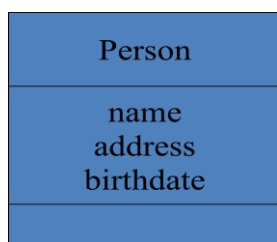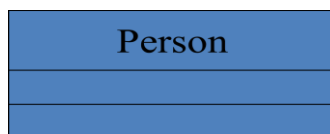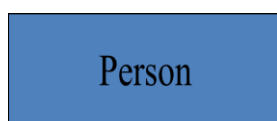| |
|---|
| **newEntry (n : Name, a : Address, p : PhoneNumber, d : Description)** |
| **getPhone (n : Name, a : Address) : PhoneNumber** |
| **PhoneBook** |

You can specify an operation by stating its signature: listing the name, type, and default value of all parameters, and, in the case of functions, a return type.

Depicting Classes:

When drawing a class, you needn't show attributes and operation in every diagram.

Person

Person

Person
name
address
birthdate

eat
play

Person
name   : String birthdate : Date ssn   : Id
eat() sleep() work() play()

Class Responsibilities:

A class may also include its responsibilities in a class diagram.

A responsibility is a contract or obligation of a class to perform a particular service.
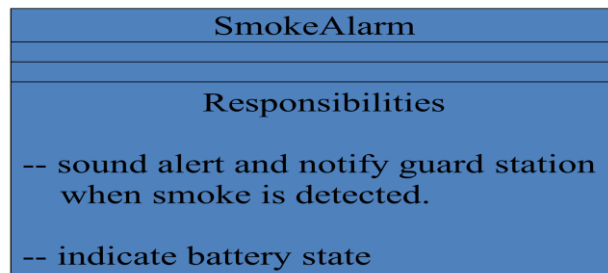
Relationships:

In UML, object interconnections (logical or physical), are modeled as relationships.

There are three kinds of relationships in UML:

- dependencies
- generalizations
- Associations

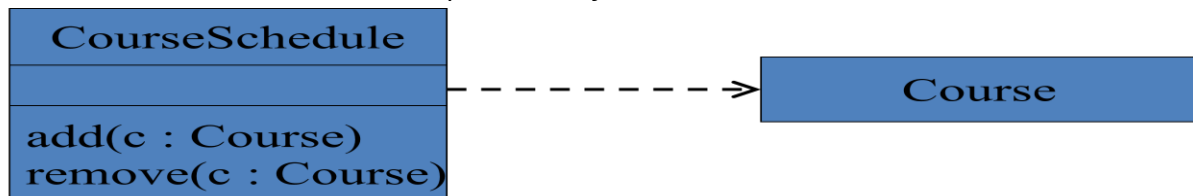***Dependency Relationships****: A dependency indicates a semantic relationship between two or more elements. The dependency from Course Schedule to Course exists because Course is used in both the **add** and **remove** operations of Course Schedule.*

**Generalization Relationships:**
A generalization connects a subclass to its super class. It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass. UML permits a class to inherit from multiple superclasses, although some programming languages (e.g., Java) do not permit multiple inheritance.
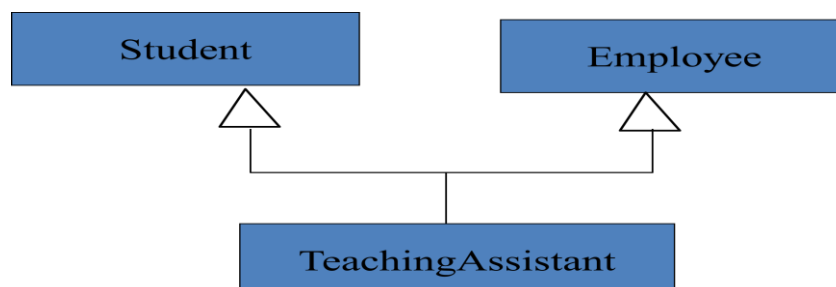
Association Relationships:

If two classes in a model need to communicate with each other, there must be link between them.

An *association* denotes that link.

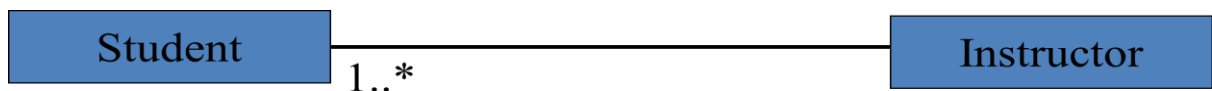| Student | ——————————— | Instructor |
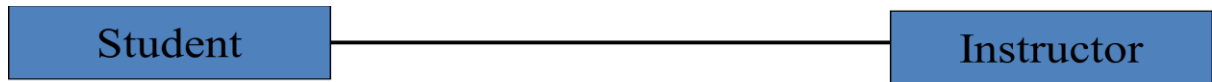
We can indicate the *multiplicity* of an association by adding *multiplicity adornments* to the line denoting the association.

The example indicates that a *Student* has one or more *Instructors*:

| Student | ——————————— 1..* | Instructor |
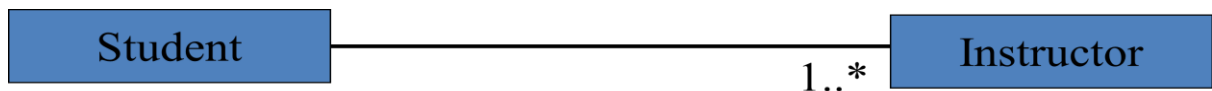
The example indicates that every *Instructor* has one or more *Students*:

| Student | 1..* ——————————— | Instructor |

We can also indicate the behavior of an object in an association (*i.e.,* the *role* of an object) using *role names.*

| Student | teaches   learns from 1..*   1..* | Instructor |
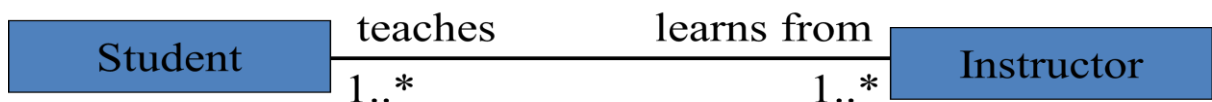
We can also name the association.

| Student | membership 1..*   1..* | Team |
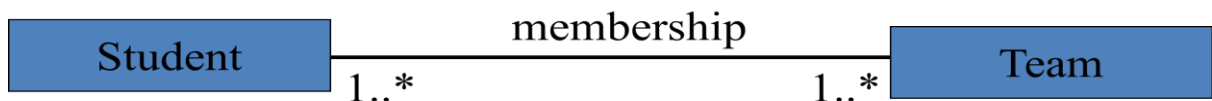
We can specify dual associations.

| Student | member of 1..*   1..*
 1   president of   1..* | Team |
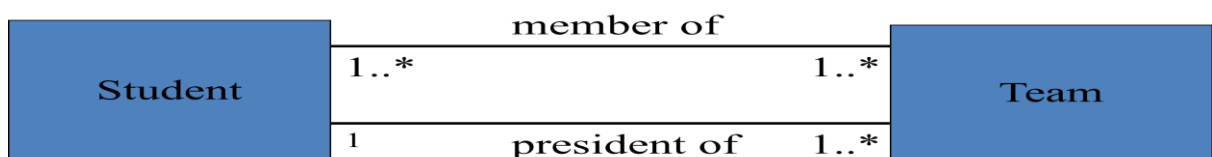
We can constrain the association relationship by defining the *navigability* of the association. Here, a *Router* object requests services from a *DNS* object by sending

messages to (invoking the operations of) the server. The direction of the association indicates that the server has no knowledge of the *Router*.

Associations can also be objects themselves, called *link classes* or an *association classes*.
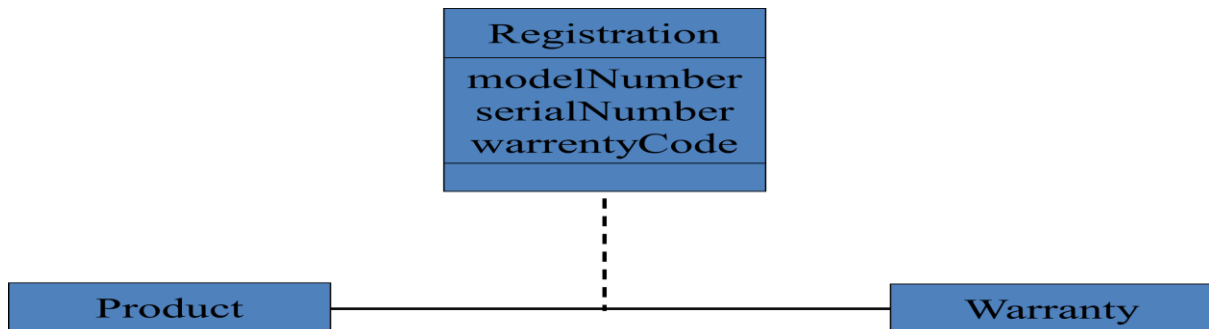
A class can have a *self association*.
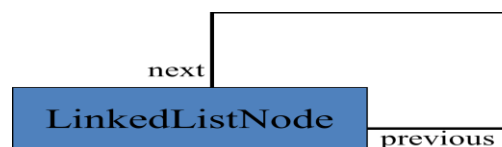
We can model objects that contain other objects by way of special associations called *aggregations* and *compositions.*

An *aggregation* specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment on the association.
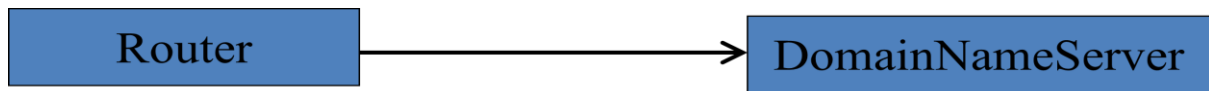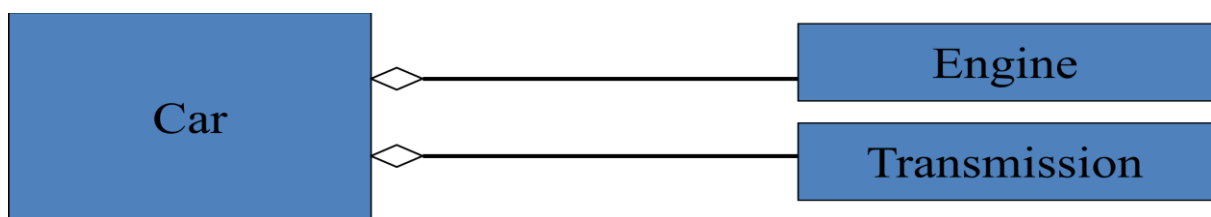

A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.,* they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.
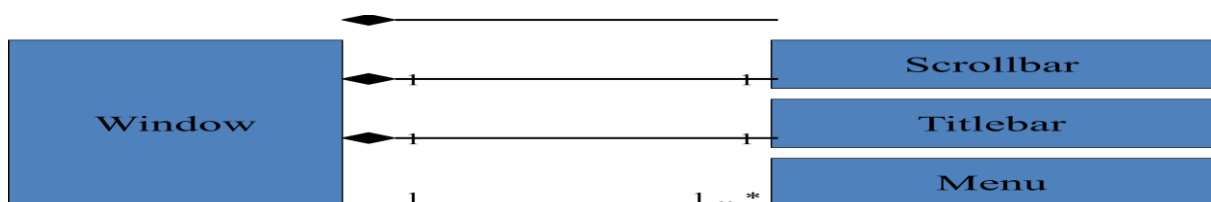
Interfaces :

An *interface* is a named set of operations that specifies the behavior of objects without showing their inner structure. It can be rendered in the model by a one- or two-compartment rectangle, with the *stereotype* <<interface>> above the interface name.

Interface Services:
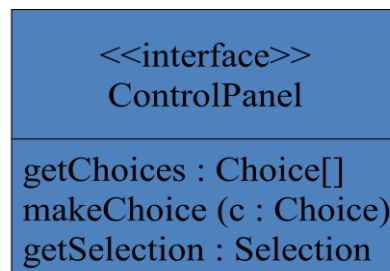
Interfaces do not get instantiated. They have no attributes or state. Rather, they specify the services offered by a related class.

Interface Realization Relationship:

A realization relationship connects a class with an interface that supplies its behavioral specification. It is rendered by a dashed line with a hollow triangle towards the specifier.

A class' interface can also be rendered by a circle connected to a class by a solid line.
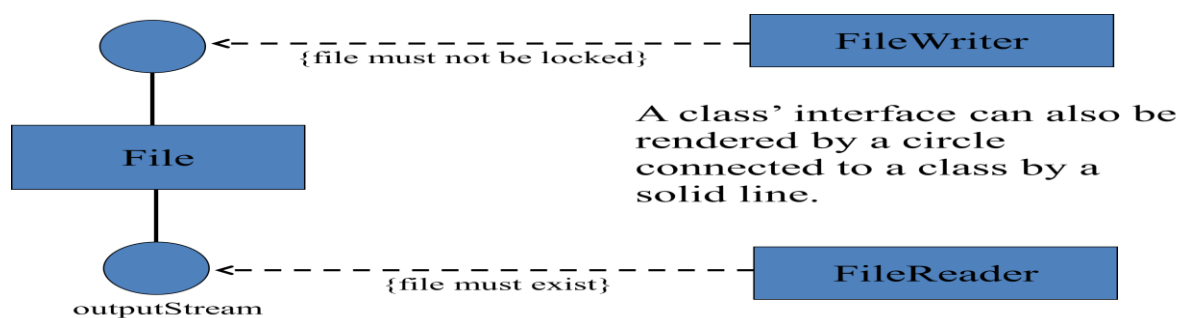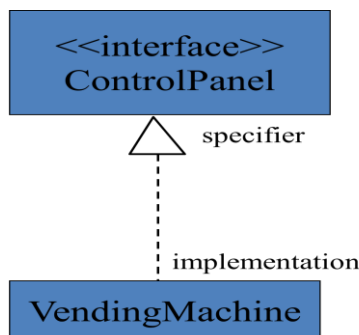
Parameterized Class:
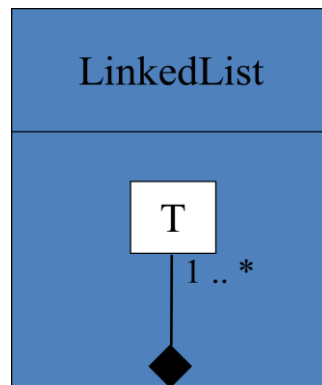
A *parameterized class* or *template* defines a family of potential elements. To use it, the parameter must be bound.

A *template* is rendered by a small dashed rectangle superimposed on the upper-right corner of the class rectangle. The dashed rectangle contains a list of formal parameters for the class.

Binding is done with the <<bind>> stereotype and a parameter to supply to the template. These are adornments to the dashed arrow denoting the realization relationship.

Here we create a linked-list of names for the Dean's List.
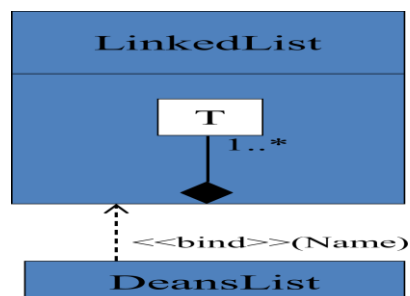
Enumeration :
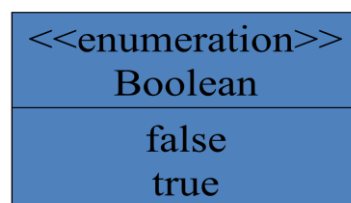
An *enumeration* is a user-defined data type that consists of a name and an ordered list of enumeration literals.

Exceptions :

Exceptions can be modeled just like any other class.

Notice the <<exception>> stereotype in the name compartment.
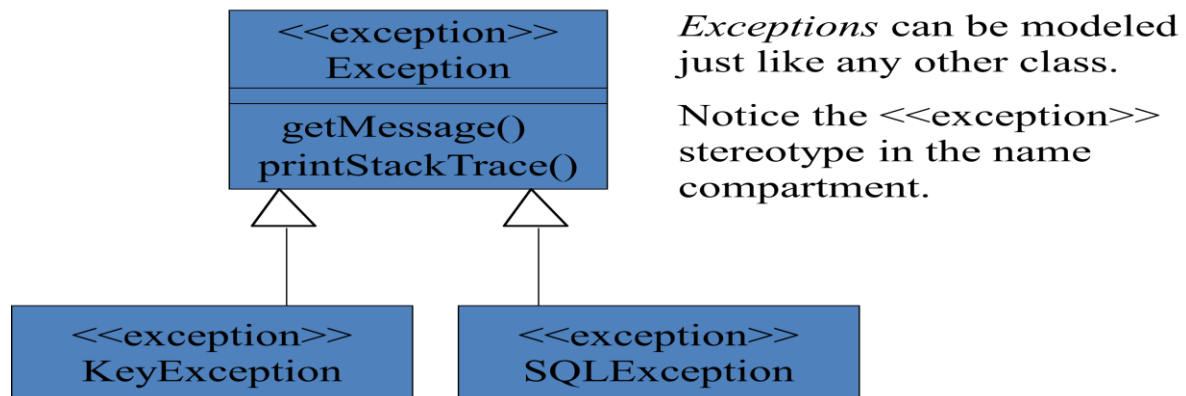
Packages: *A package is a container-like element for organizing other elements into groups.*

A package can contain classes and other packages and diagrams.

Packages can be used to provide controlled access between classes in different packages.
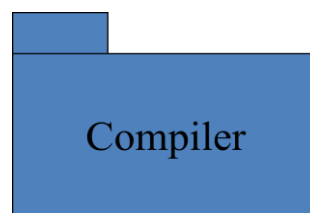
Classes in the *FrontEnd* package and classes in the *BackEnd* package cannot access each other in this diagram.
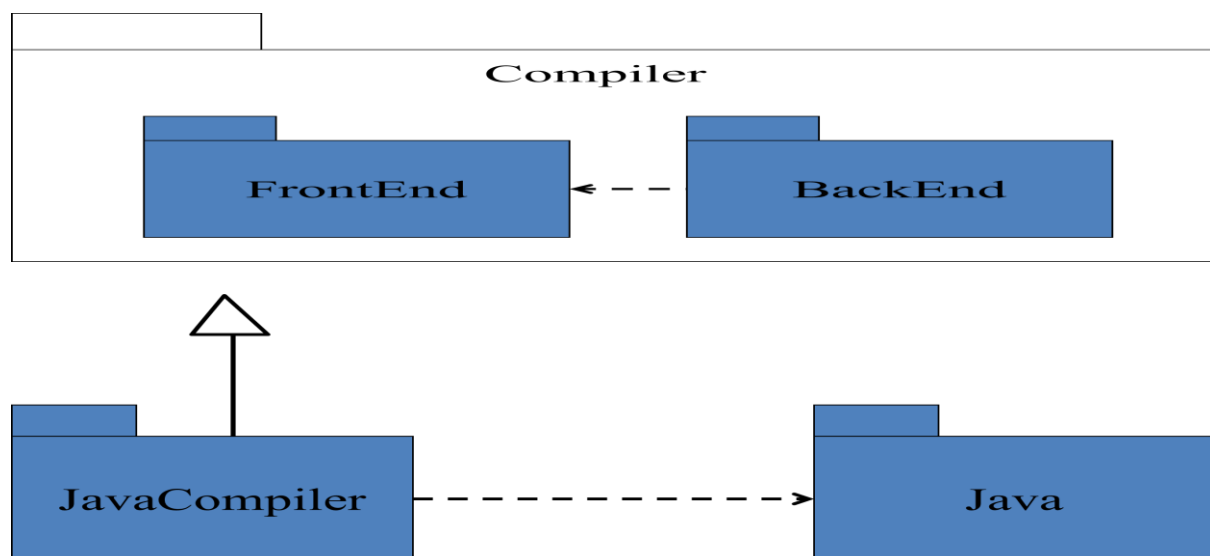
We can model generalizations and dependencies between packages.

Component Diagram:

Component diagrams are one of the two kinds of diagrams found in modeling the physical aspects of an object-oriented system. They show the organization and dependencies between a set of components.

Use component diagrams to model the *static implementation view* of a system. This involves modeling the physical things that reside on a node, such as executables, libraries, tables, files, and documents.

Component Diagram: *Here's an example of a component model of an executable release.*
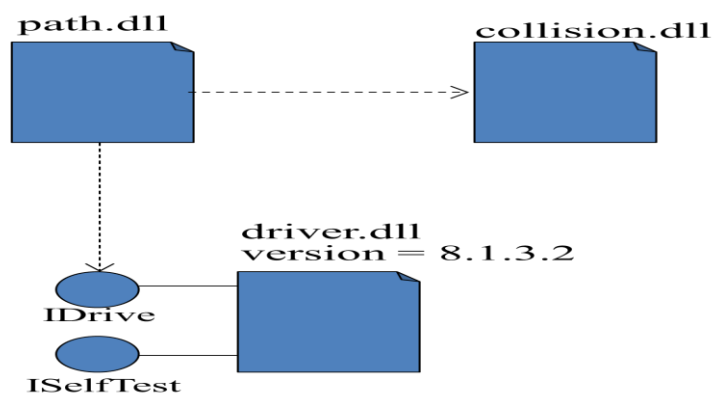
Component Diagram :
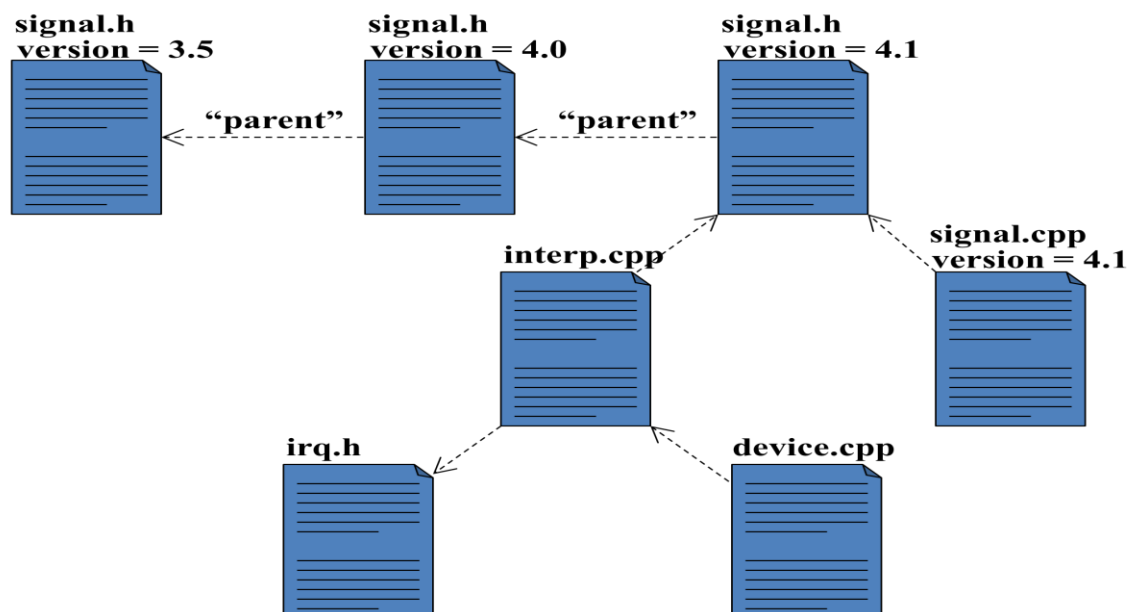
Deployment Diagram :

Deployment diagrams are one of the two kinds of diagrams found in modeling the physical aspects of an object-oriented system. They show the configuration of *run-time processing* nodes and the components that live on them.

Use deployment diagrams to model the *static deployment view* of a system. This involves modeling the topology of the hardware on which the system executes.

A component is a physical unit of implementation with well-defined interfaces that is intended to be used as a replaceable part of a system. Well designed components do not depend directly on other components, but rather on interfaces that components support.
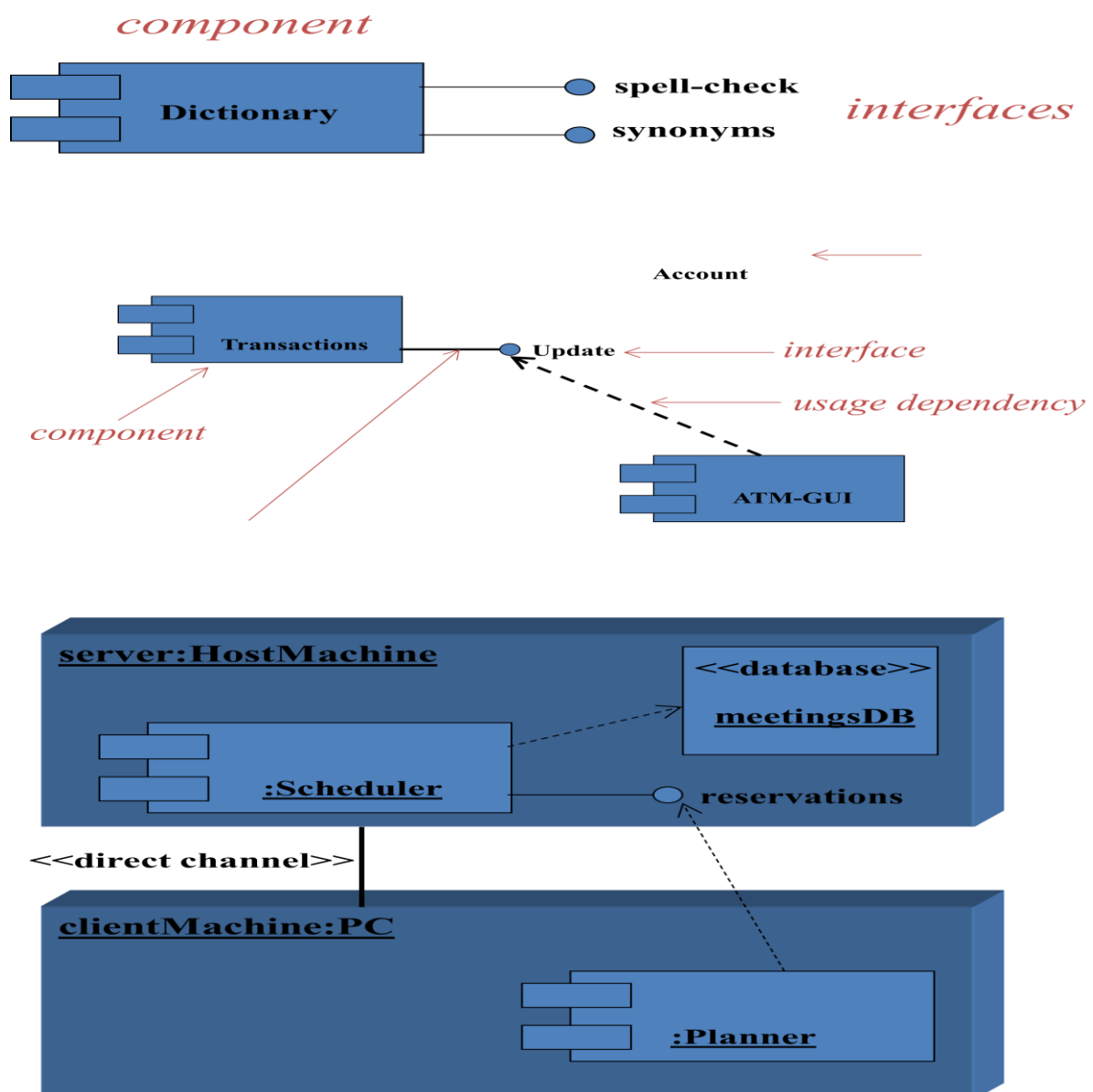
Use Case :

"A *use case* specifies the behavior of a system or a part of a system, and is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor."

"An *actor* is an idealization of an external person, process, or thing interacting with a system, subsystem, or class. An actor characterizes the interactions that outside users may have with the system."

A use case is rendered as an ellipse in a use case diagram. A use case is always labeled with its name.
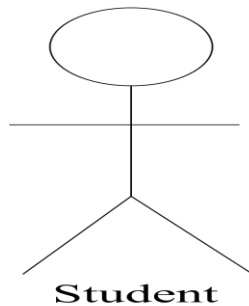
An actor is rendered as a stick figure in a use case diagram. Each actor participates in one or more use cases.

Actors can participate in a generalization relation with other actors.
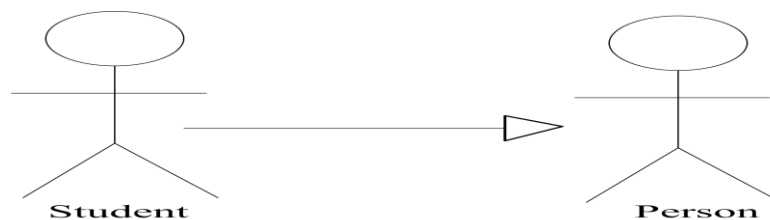
Actors may be connected to use cases only by associations.
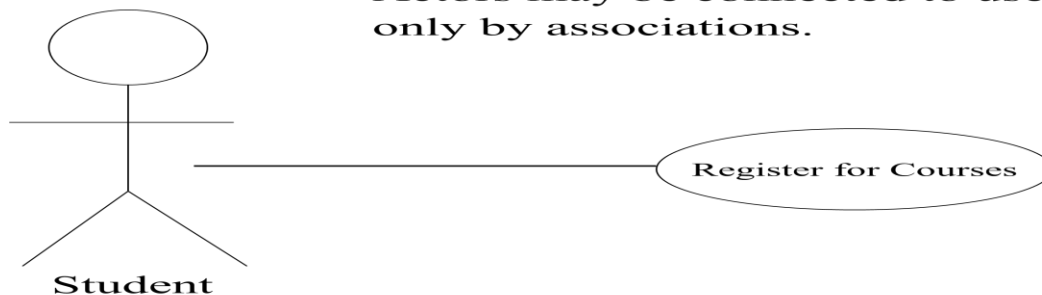
Here we have a *Student* interacting with the *Registrar* and the *Billing System* via a "*Register for Courses*" use case.
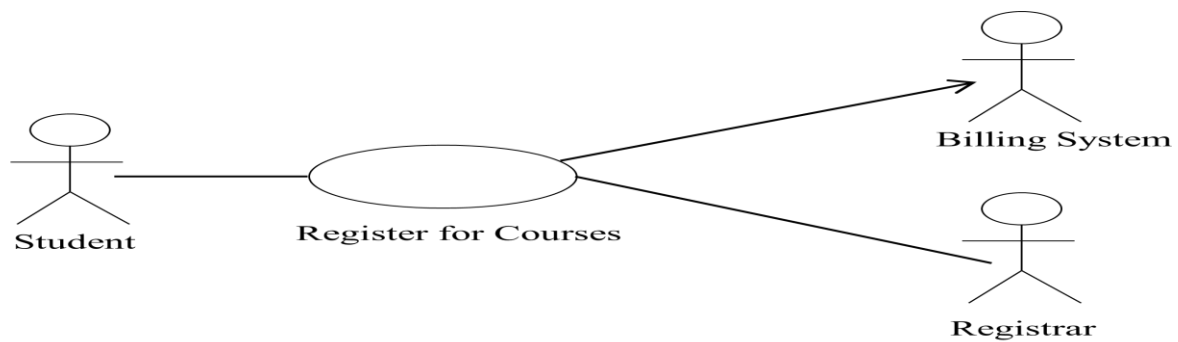
State Machine :

"The state machine view describes the dynamic behavior of objects over time by modeling the lifecycles of objects of each class. Each object is treated as an isolated entity that communicates with the rest of the world by detecting events and responding to them. Events represent the kinds of changes that objects can detect... Anything that can affect an object can be characterized as an event."

An object must be in some specific state at any given time during its lifecycle. An object transitions from one state to another as the result of some event that affects it. You may create a state diagram for any class, collaboration, operation, or use case in a UML model .

There can be only one start state in a state diagram, but there may be many intermediate and final states.
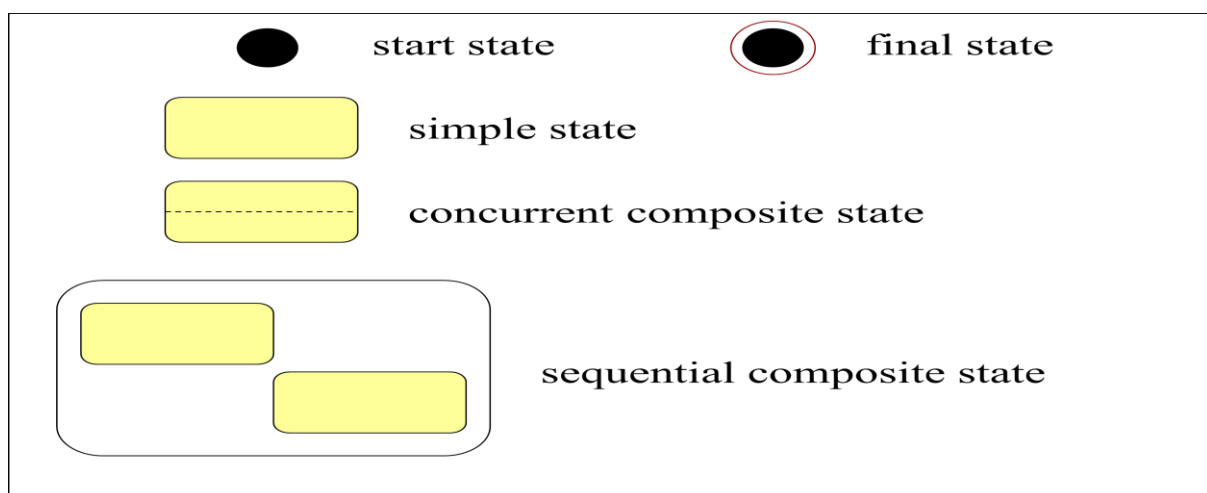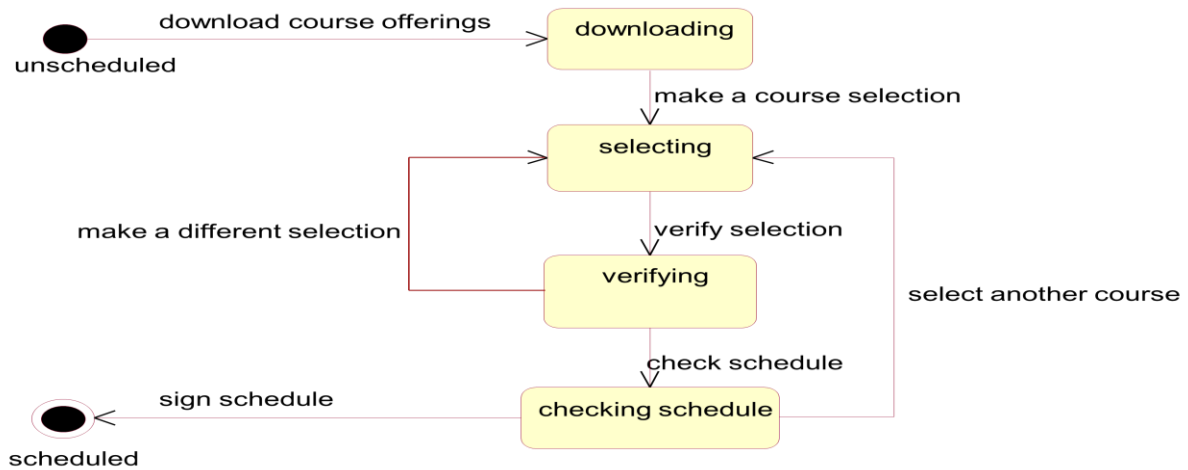
Sequence Diagram :

A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. It shows a set of objects and the messages sent and received by those objects.

Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis.
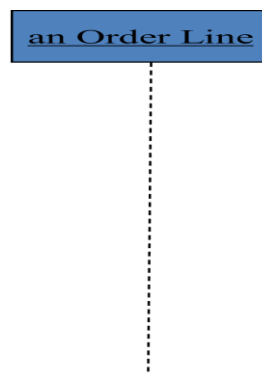
An object in a sequence diagram is rendered as a box with a dashed line descending from it.

The line is called the *object lifeline*, and it represents the existence of an object over a period of time.
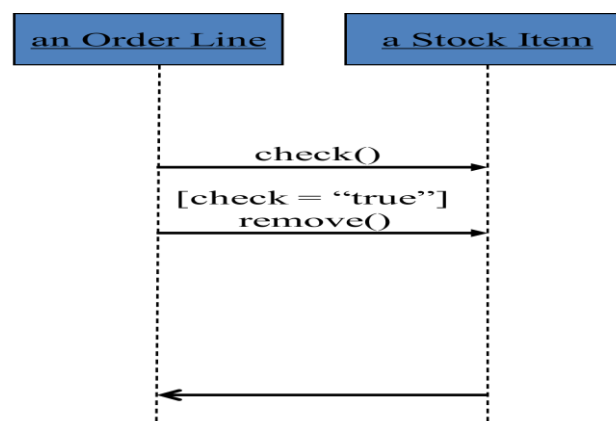
Messages are rendered as horizontal arrows being passed from object to object as time advances down the object lifelines. Conditions (such as [check = "true"]) indicate when a message gets passed.

```
  ┌─────────────────┐      ┌─────────────────┐
  │  an Order Line  │      │  a Stock Item   │
  └─────────────────┘      └─────────────────┘
          ┆                        ┆
          ┆        check()         ┆
          ┆──────────────────────▶ ┆
          ┆   [check = "true"]     ┆
          ┆        remove()        ┆
          ┆──────────────────────▶ ┆
          ┆                        ┆
          ┆◀────────────────────── ┆
          ┆                        ┆
```
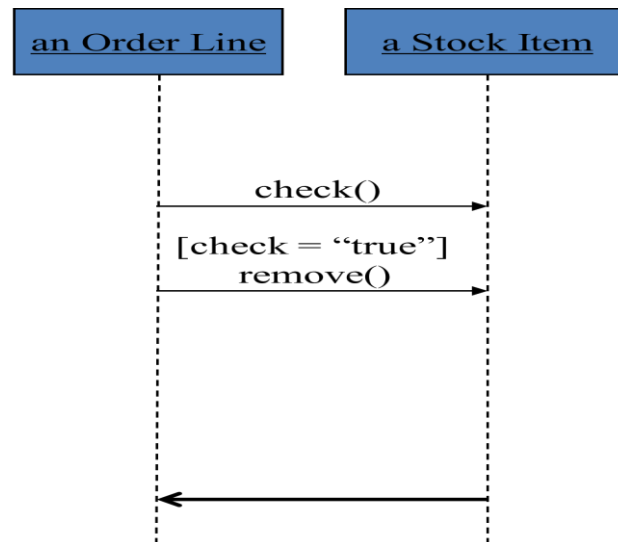
Notice that the bottom arrow is different. The arrow head is not solid, and there is no accompanying message. This arrow indicates a **return** from a previous message, not a new message.

```
  ┌─────────────────┐      ┌─────────────────┐
  │    an Order     │      │  a Order Line   │
  └─────────────────┘      └─────────────────┘
          ┆                        ┆
          ┆   *    prepare()       ┆
          ┆──────────────────────▶ ┆
          ┆    ↑                   ┆
          ┆  Iteration             ┆
          ┆  marker                ┆
          ┆                        ┆
```
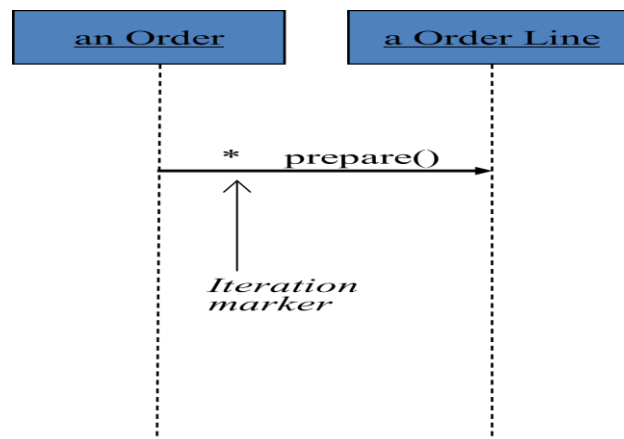
An iteration marker, such as * (as shown), or *[i = 1..n] , indicates that a message will be repeated as indicated.

[Fowler,97]
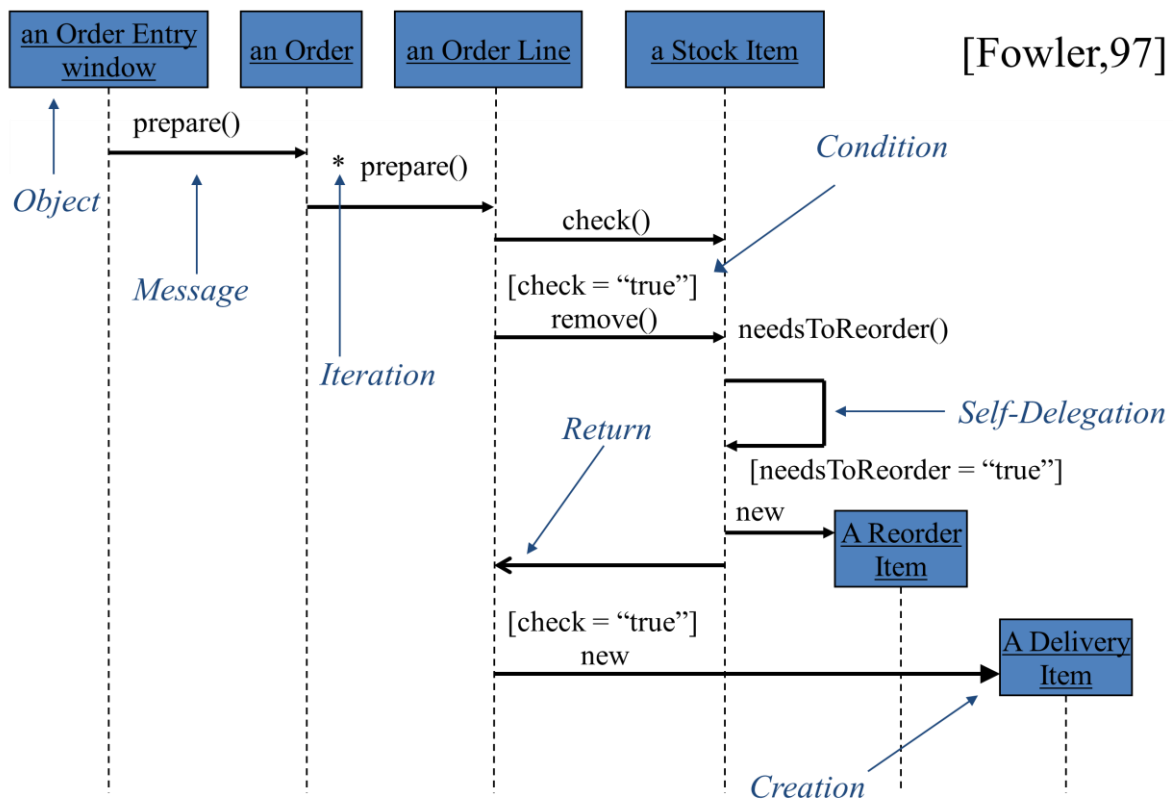
Collaboration Diagram :

A collaboration diagram emphasizes the relationship of the objects that participate in an interaction. Unlike a sequence diagram, you don't have to show the lifeline of an object explicitly in a collaboration diagram. The sequence of events are indicated by sequence numbers preceding messages.

Object identifiers are of the form *objectName : className,* and either the objectName or the className can be omitted, and the placement of the colon indicates either an objectName: , or a :className.
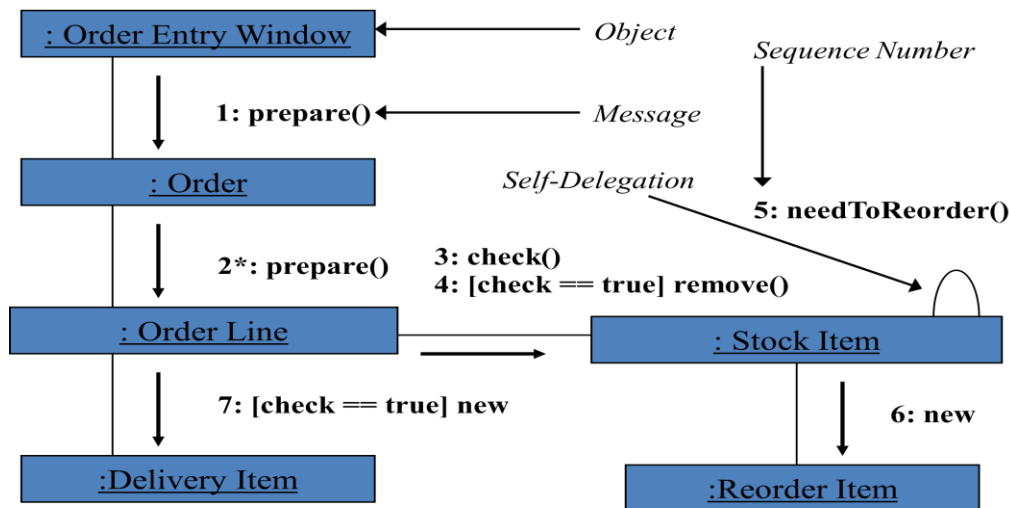
[Fowler,97]

Collaboration Diagram Sequence Diagram :

Both a collaboration diagram and a sequence diagram derive from the same information in the UML's metamodel, so you can take a diagram in one form and convert it into the other. They are semantically equivalent.

Activity Diagram :

An activity diagram is essentially a flowchart, showing the flow of control from activity to activity.

Use activity diagrams to specify, construct, and document the dynamics of a society of objects, or to model the flow of control of an operation. Whereas interaction diagrams emphasize the flow of control from object to object, activity diagrams emphasize the flow of control from activity to activity. *An activity is an ongoing non-atomic execution within a state machine.*

```
                                    ┌──────────┐
               ●──────────────────▶ │ Receive  │
                                    │  Order   │
                                    └────┬─────┘
                                         │                      Multiple Trigger
                                    ▬▬▬▬▬▬▬▬▬▬
                            ┌────────┘      └─────────┐◀──────  for each line
                            │                     *   │          item on order
                            ▼                         ▼
   ┌────────┐        ┌────────────┐           ┌──────────┐
   │ Cancel │ ◀───── │  Authorize │           │  Check   │
   │ Order  │        │  Payment   │           │  Line    │
   └────────┘ [failed]└────────────┘          │  Item    │
                            │                 └────┬─────┘
                       [succeeded]              [in stock]
                            │                       │
                            │                       ▼
                            │                 ┌──────────┐
                            │                 │ Assign to│
                            │                 │  Order   │
                            │                 └────┬─────┘
    Synchronization Condition                      │
              │                                ▬▬▬▬▬▬▬▬▬▬
              ▼                          ┌────────┘    └─────────┐
                                         │         [need to      │
   [stock assigned to                    │          reorder]     ▼
    all line items and ▬▬▬▬▬▬▬▬▬▬        │                 ┌──────────┐
    payment authorized]     │            │                 │ Reorder  │
                            ▼                              │  Item    │
                      ┌──────────┐                         └──────────┘
                      │ Dispatch │
                      │  Order   │
                      └──────────┘
```