

Software Engineering (IT2020)

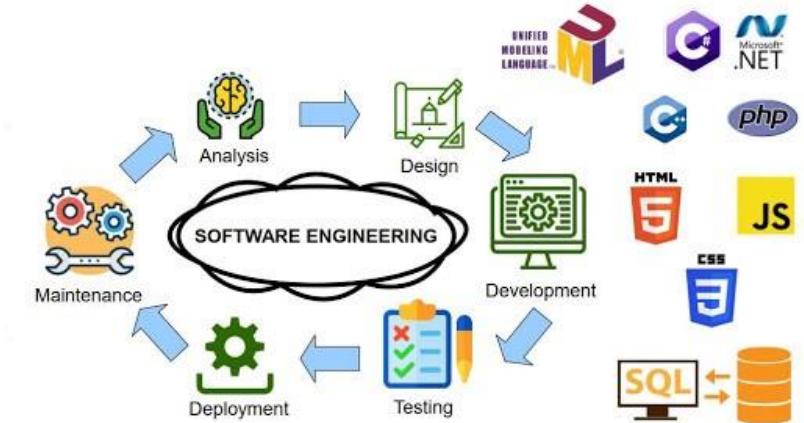
2022

Lecture 1 – Introduction & Object Diagram

Outline

1. What is Software Engineering ?
2. Software Development Life Cycle and phases
(Covered in previous semester)
3. UML Diagrams (Covered in previous semester)
4. Class Diagram (Covered in previous semester)
5. Object Diagrams

What is Software Engineering?



- IEEE Definition of Software Engineering:

The application of a **systematic, disciplined, quantifiable** approach for the development, operation, and maintenance of software.

Ref : IEEE Standard 610.12-1990, 1993.

- Software engineering** is defined as a process of analyzing user requirements and then designing, building, and testing software application which will satisfy those requirements.

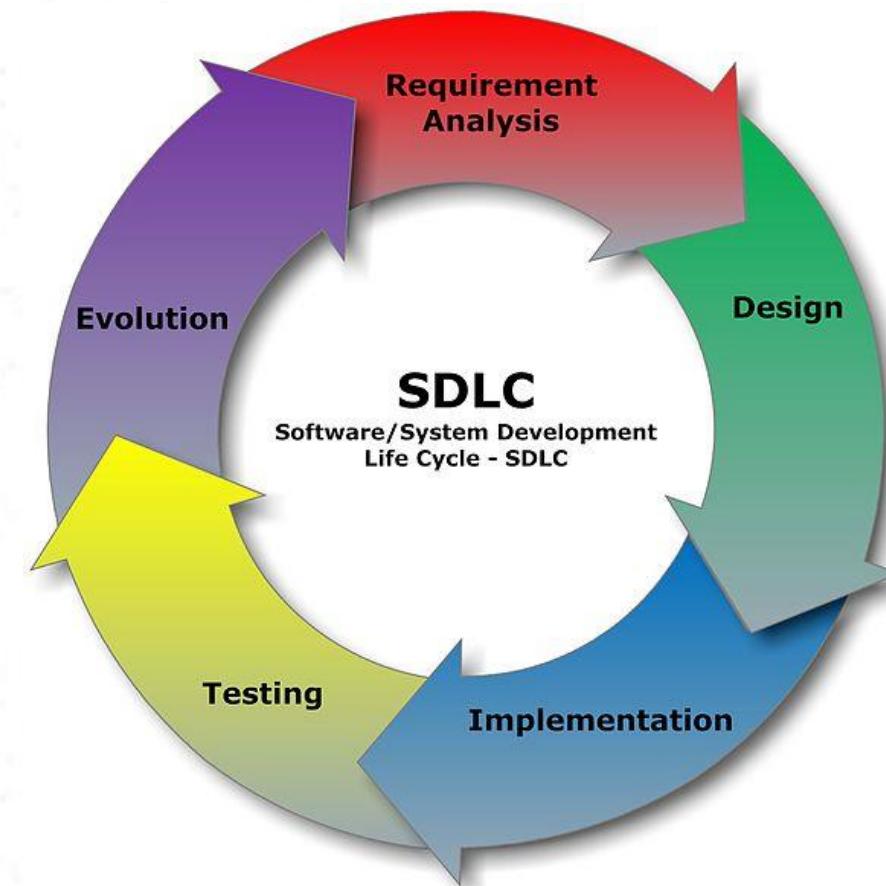
Software Development Process

- In Software Engineering, an Engineering process is followed to transform inputs into outputs/software products.
- The software development process consists of a set of activities and associated results that produce a Software.



Software Development Life Cycle

- Software Development Life Cycle (SDLC) is a framework that defines the phases/stages to be followed throughout the software development process.



Software Development Life Cycle

Phase 1 : Requirement Gathering and Analysis

Phase 2 : Design

Phase 3 : Implementation

Phase 4 : Testing

Phase 5 : Maintenance / Evolution

Design Phase

Software Design Methods

- **Function Oriented Software Design**
- **Object Oriented Software Design**

In SE Module, we are going to cover Object Oriented Software Design.

Object Oriented Design

- **Object Oriented Software Design:**
- Object-oriented design is the discipline of defining the objects and their interactions to solve a software problem.
- Object Oriented Concepts are the base for the object-oriented design.

Object Oriented Software Design Cont...

Design Model Types

- Structural Models
- Dynamic Models

Modeling Languages

A modeling language is any artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules.

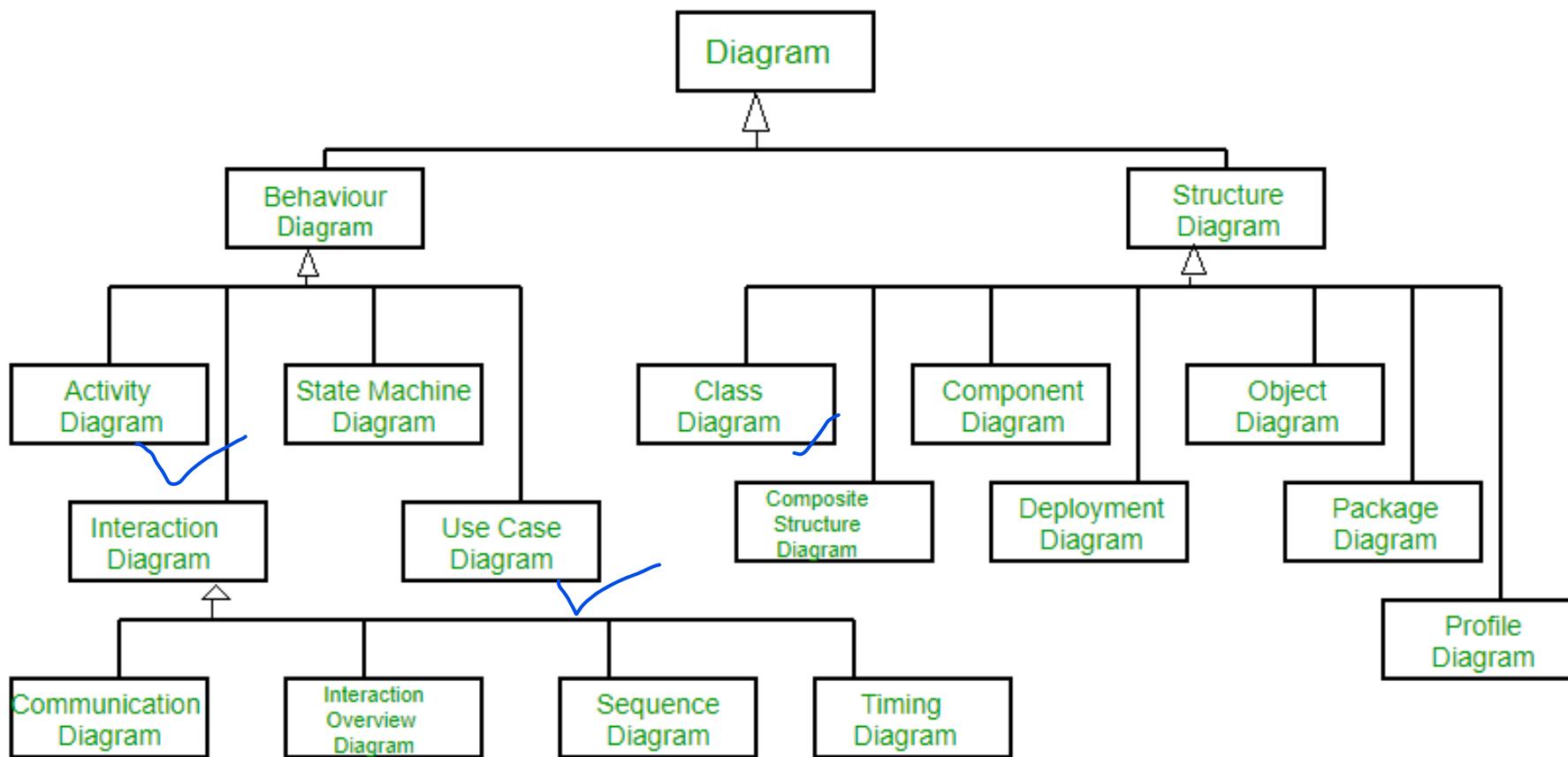
Unified Modeling Language (UML)

What Is the UML?

UML is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system.



UML Diagram Structure



Object Oriented Software Design Cont...

In SE module we are going to learn following UML Diagrams.

- **Class Diagram- Completed in OOC**
- **Object Diagram**
- **Sequence and Communication Diagrams –(Interaction Diagrams)**
- **State Diagram**
- **Component and Deployment Diagrams –(Physical Diagrams)**

Class Diagram Revision

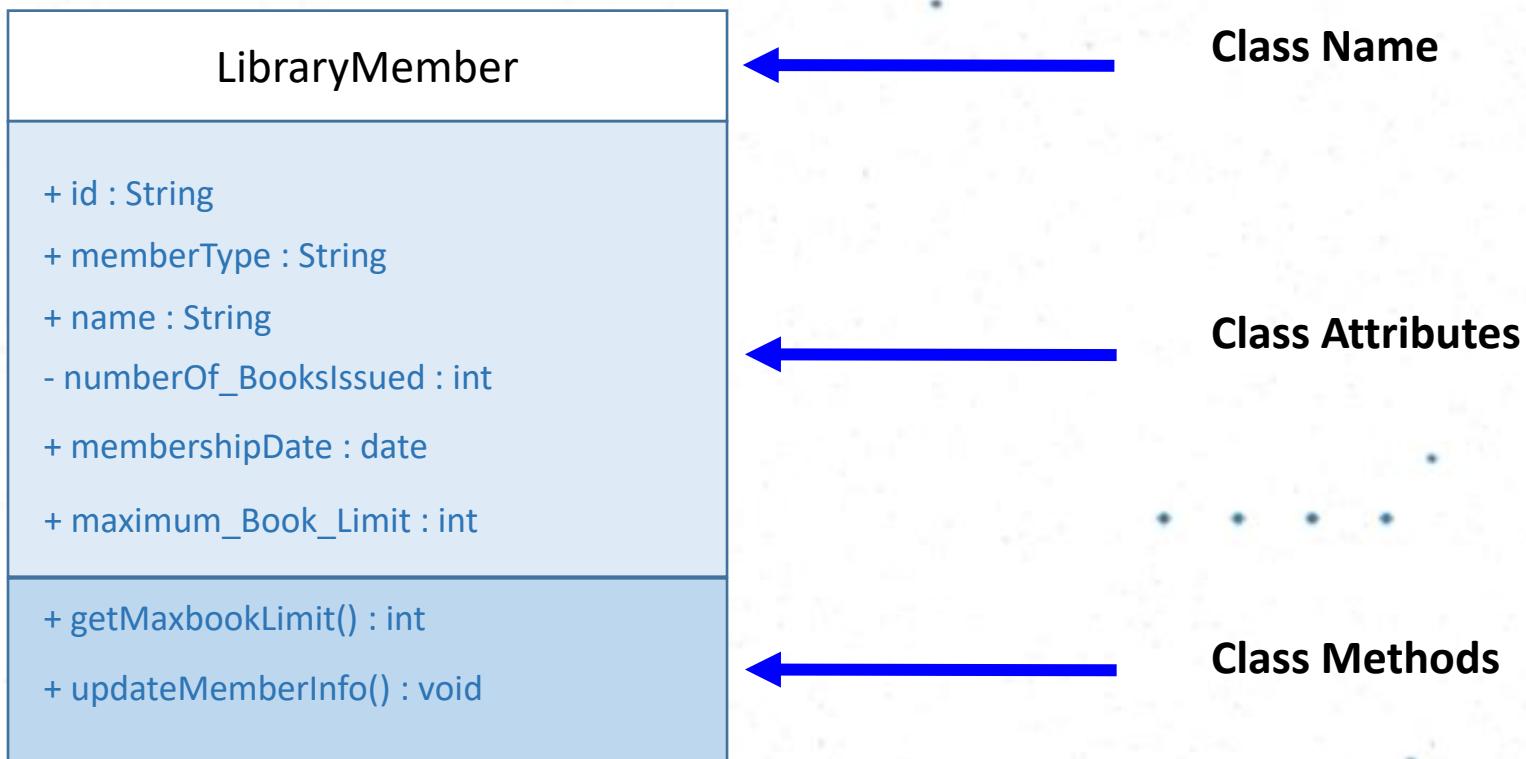
How to discover classes?

Noun/ Verb Analysis

- Through Noun/Verb Analysis, we can identify objects in our problem statement by looking for **nouns** and **noun phrases**.
- Each of these can be underlined and becomes a candidate for an object in our solution.
- Then write **Class Responsibility and Collaboration (CRC)** Cards for final set of classes.

Class Diagram

Class Structure



Class Structure Cont...

Class Attributes

- Attributes of a class can be fully specified as below.

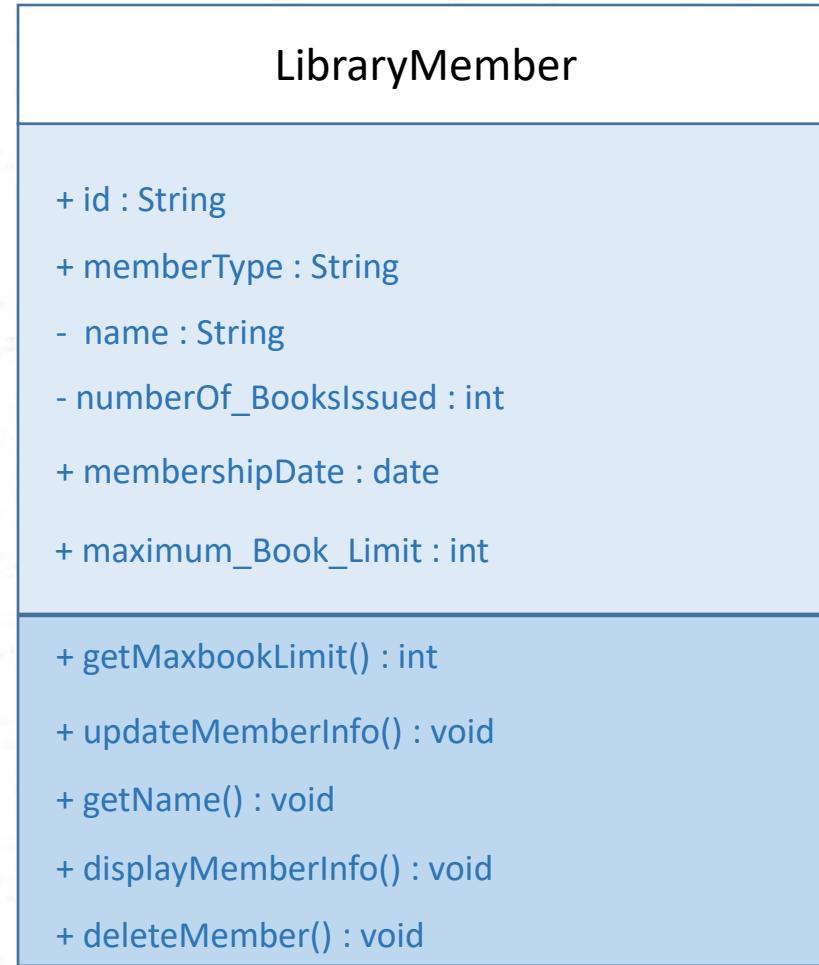
```
visibility name multiplicity : type = initial value {property}
```

Class Methods

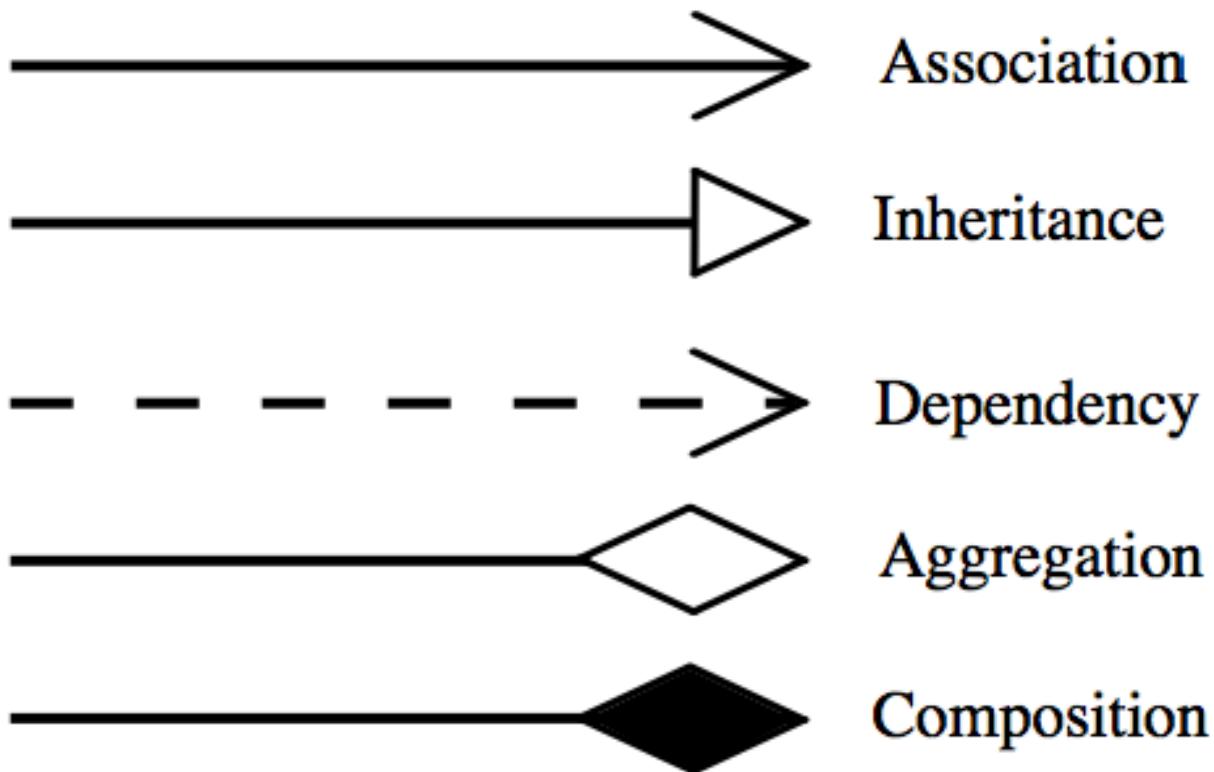
- Methods of a class can be fully specified as below.

```
name (parameters) : type
```

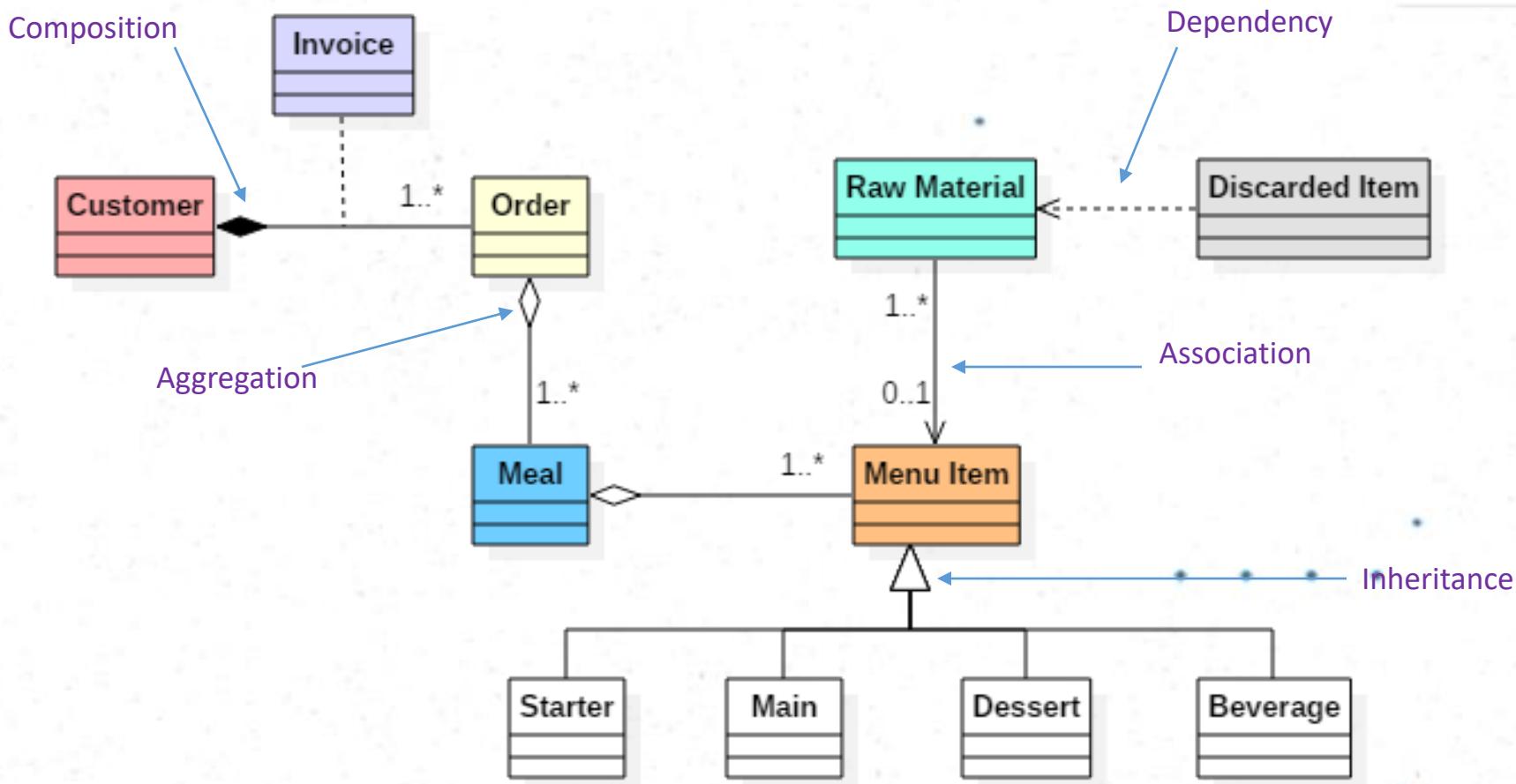
Class Structure Example



Class Relationships



Class Relationships Cont.



Exercise 1

As discuss in OOC, draw a class diagram for SLIIT Library Management System.

Object Diagram

Object Diagram

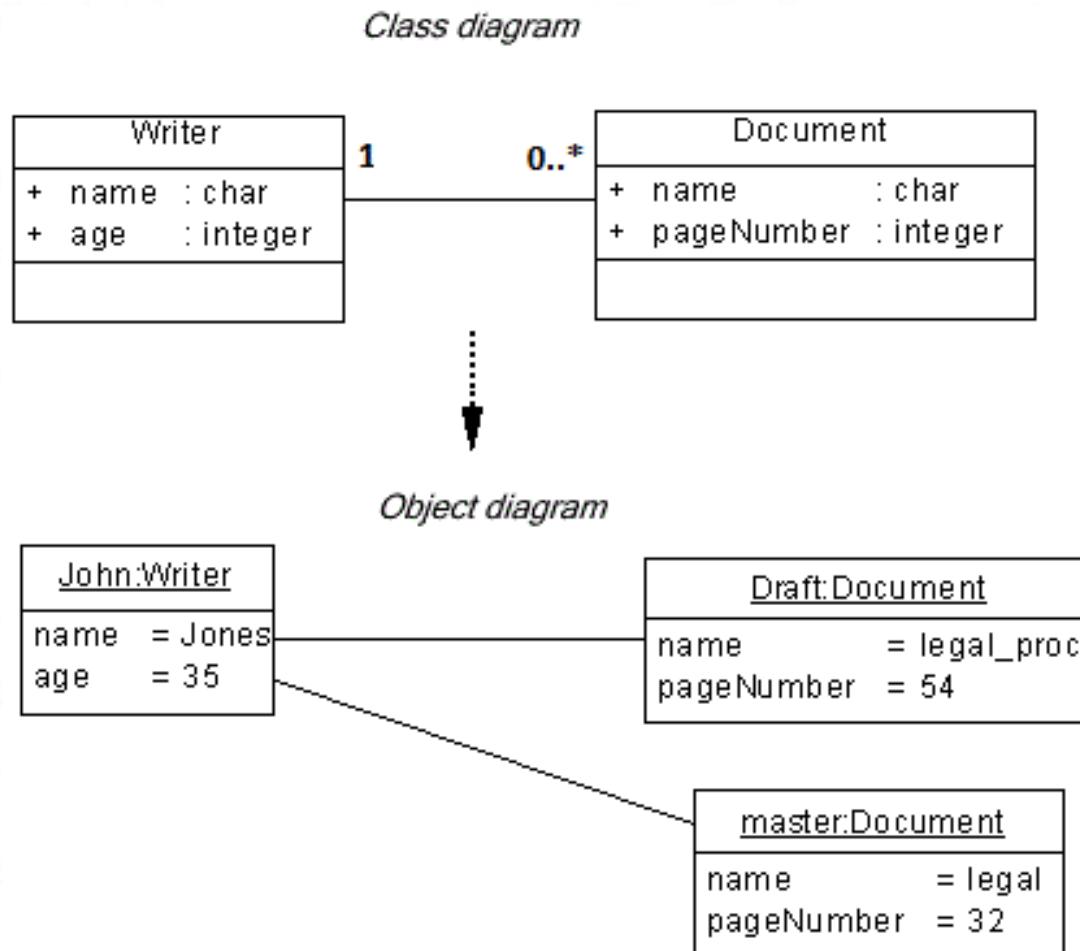
UML Specification:

“An object diagram is a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time.”

Object Diagram

- Object diagrams are derived from class diagrams, so object diagrams are dependent upon class diagrams.
- Both Class and Object diagrams are meant to visualize the structure of a system. Hence categorize under **Structural** diagram.
- Object diagrams represent an instance of a class diagram.
- The attributes identified by the class now have values associated with it.
- The purpose is to capture the static view of a system at a particular moment.

Object Diagram Example



Object Notation

objectname:Classname

Attributename1 : Type = value

Attributename2 : Type = value

- Top compartment contains object name and class name.
- Bottom compartment contains list of attribute names and values assigned.
- No need to show the operations (they are the same for all objects of a class)

Different Notation Types

Named Object :

Object name and the class name both should be there.

Objectname : Classname

Anonymous Object :

The name of the object may be omitted (optional), but the colon should be kept with the class name.

:Classname

Shorter form of Notation

Object With Attributes :

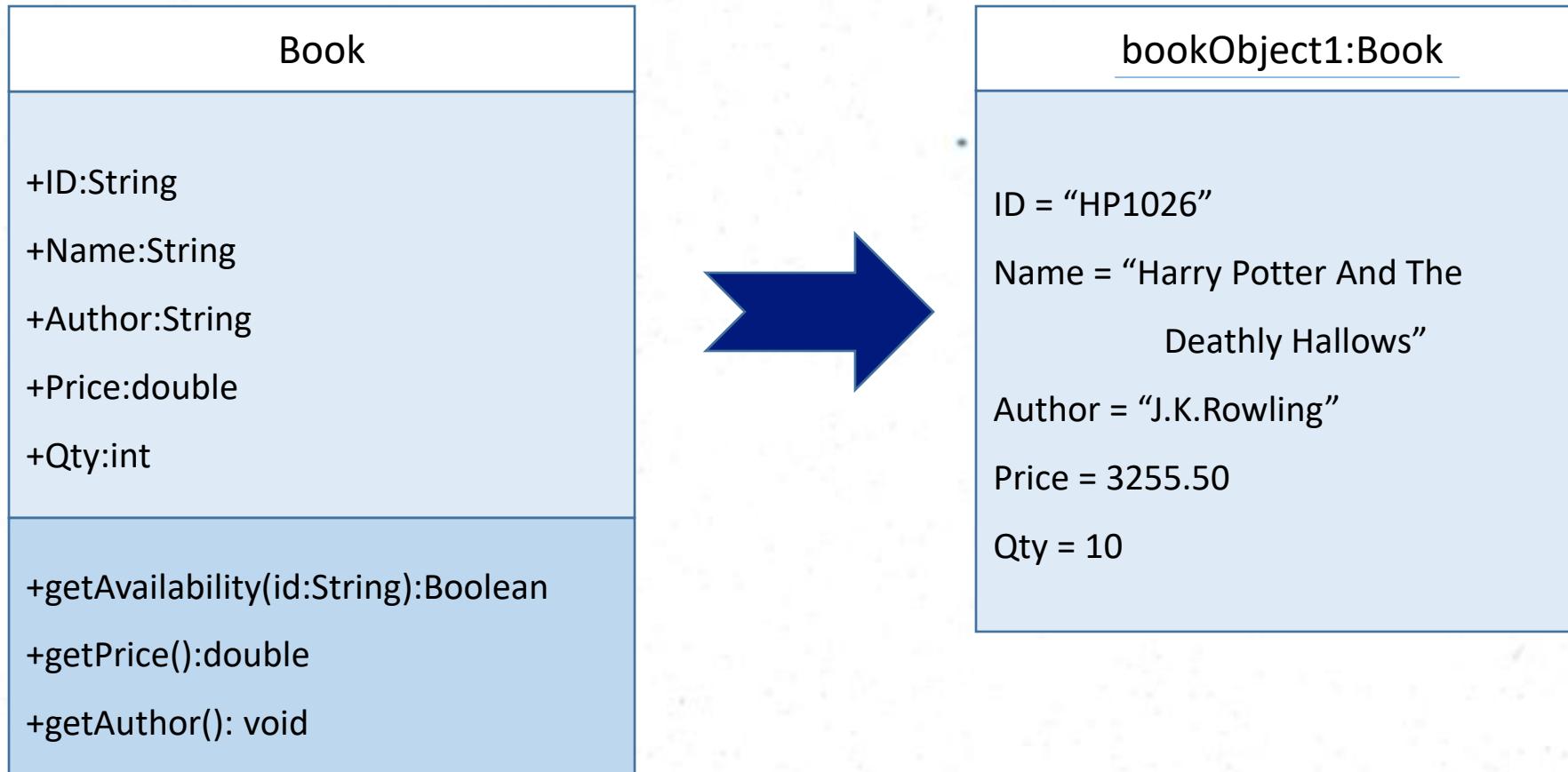
objectname:Classname

Attributename1 = “value”

Attributename2 = value

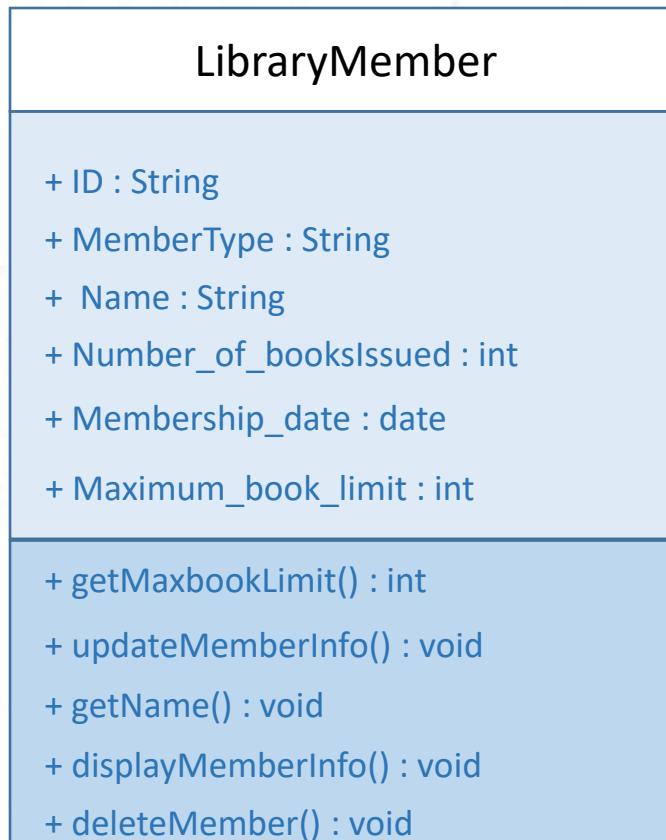
Note: Double quotes (“ ”)are used for String values.

Sample Object Diagram in UML



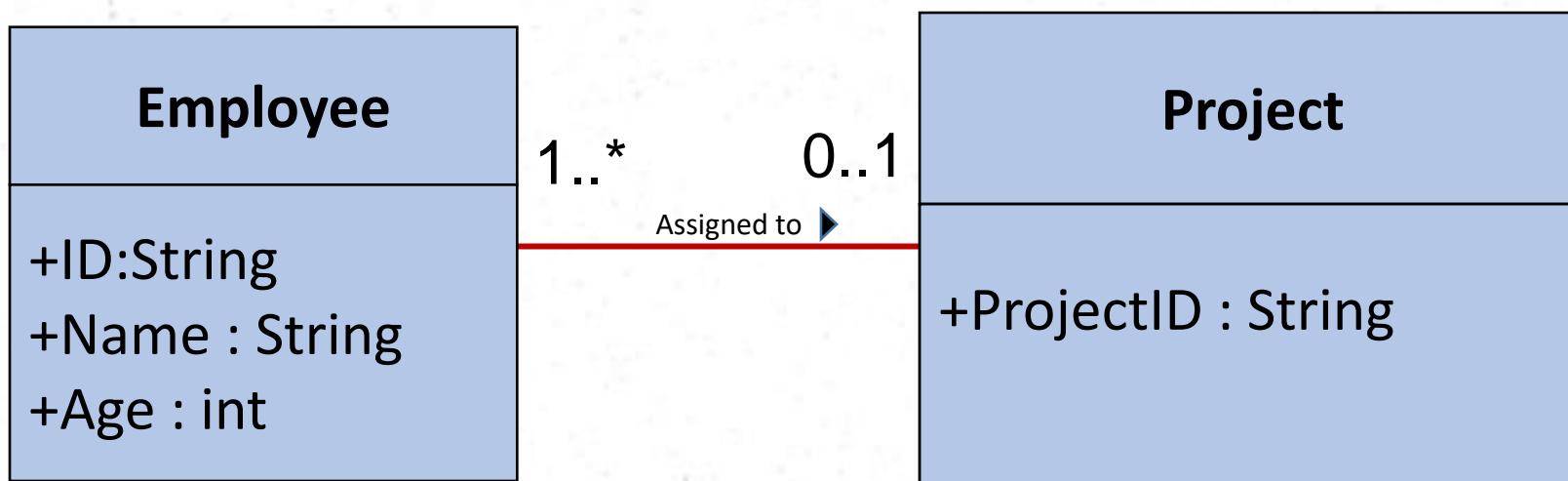
Exercise 2

Assume that you are the Library Member. Draw your Library Member object.



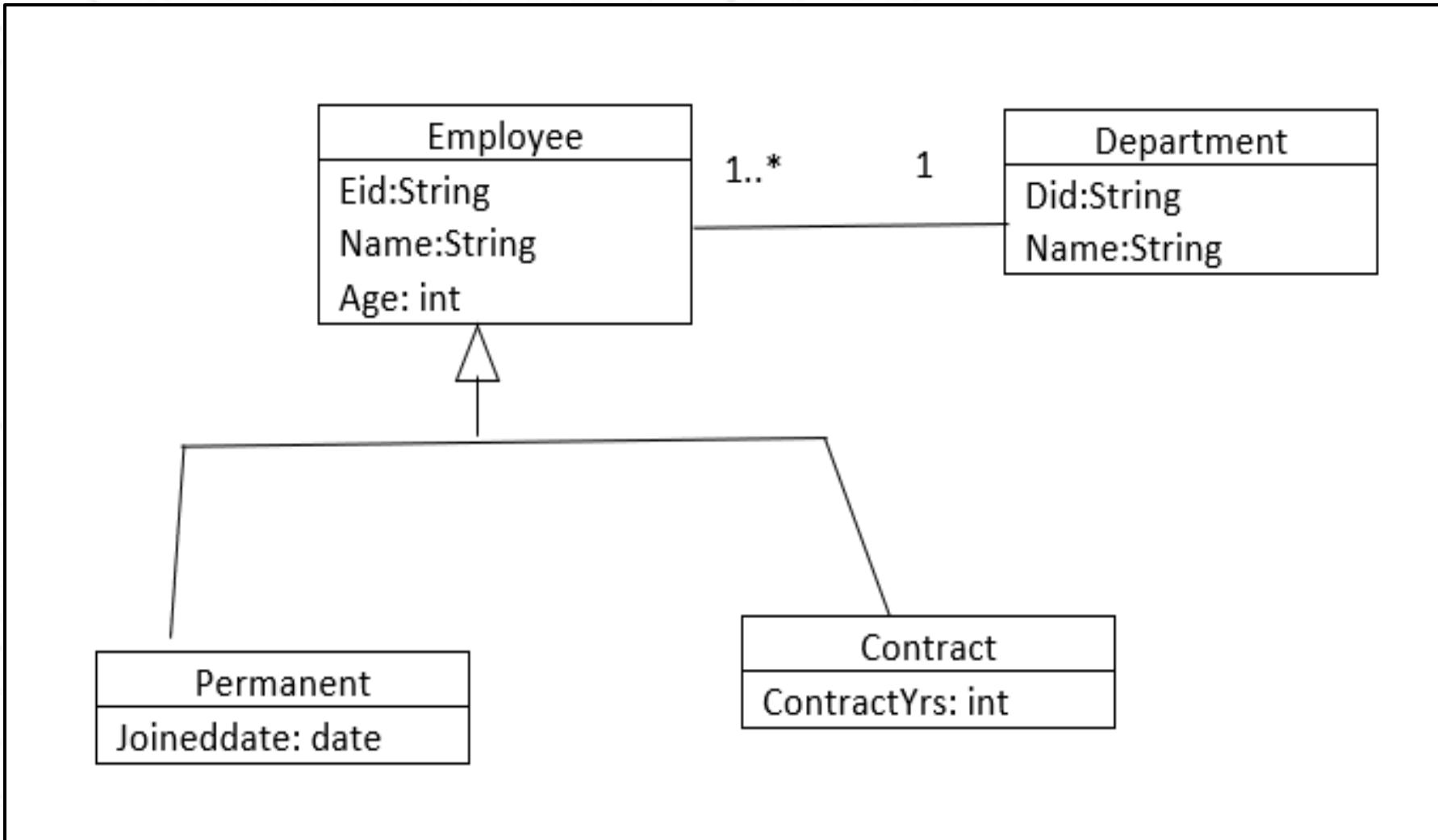
Exercise 3

Draw an Object Diagram



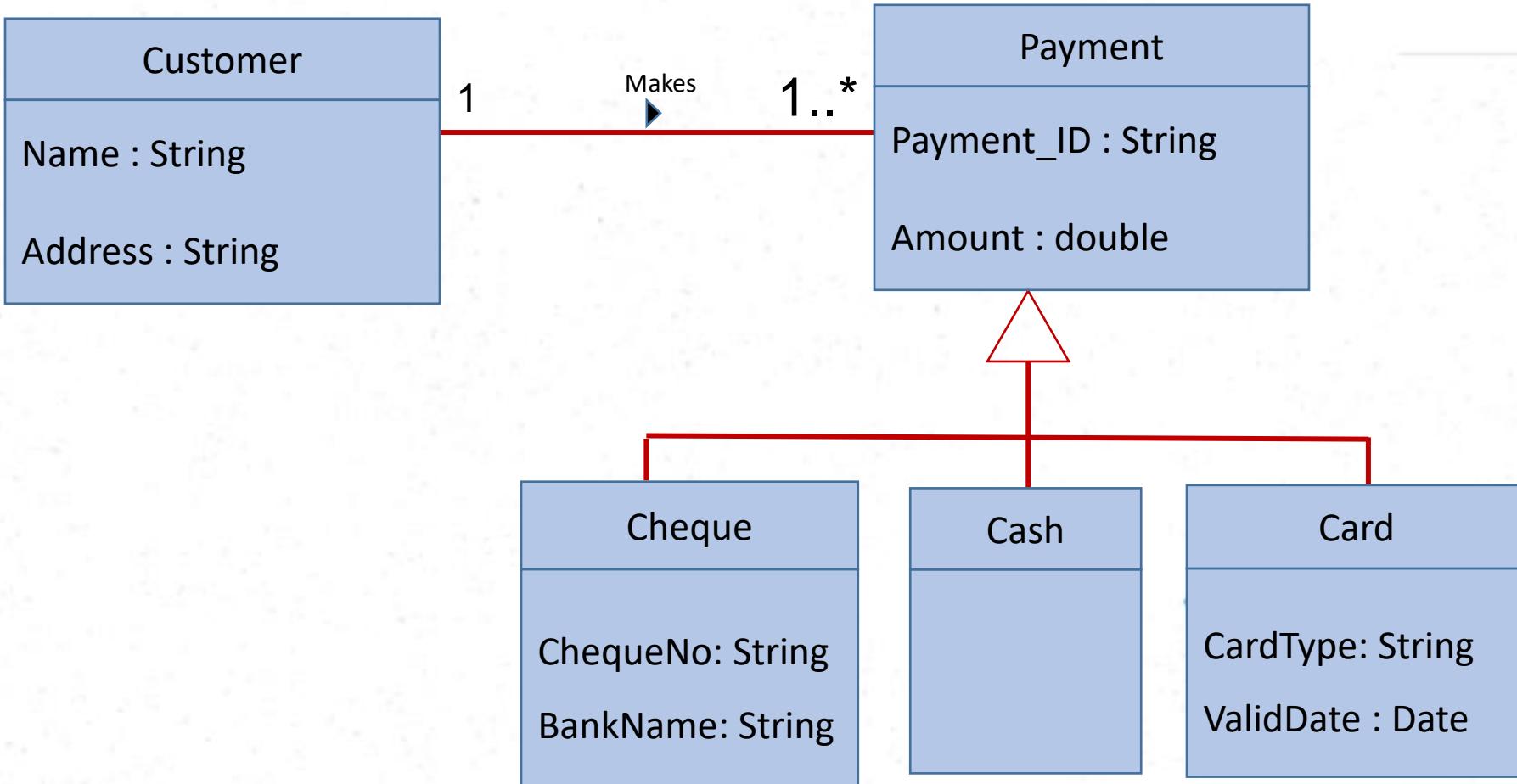
Exercise 4

Draw an Object Diagram for the given partial class diagram.



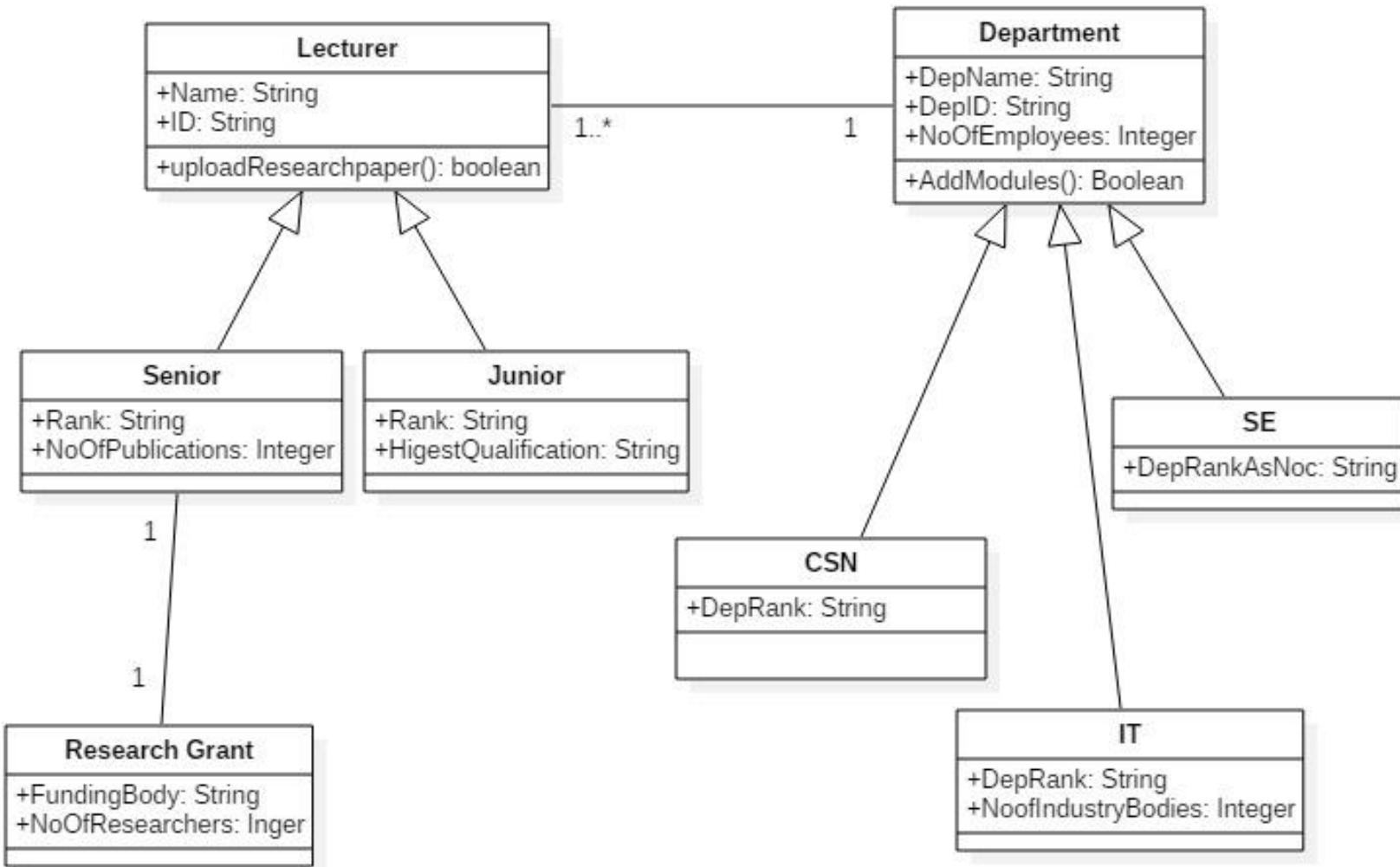
Exercise 5

Draw an Object Diagram for the given partial class diagram.



Exercise 06 – Self-study Question

Draw an Object Diagram for the given partial class diagram.



References

- IEEE Standard 610.12-1990, 1993.
- Software Engineering, I.Sommerville, 10th ed. , Pearson Education. (p. 21)
- Grady Booch, eta (2008), Object Oriented Analysis and Design with Applications 3rd Edition, pg 44,52)

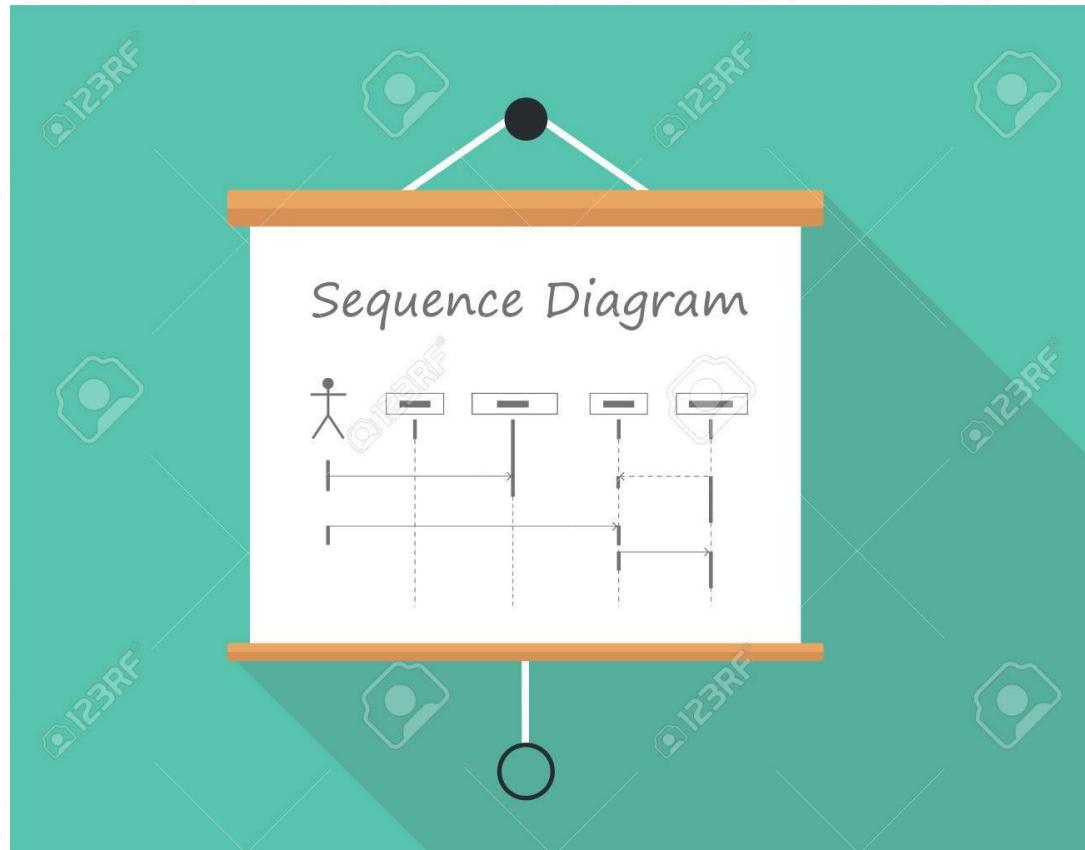
Thank you...

Software Engineering (IT2020)

2022

Lecture 2 - Interaction Diagrams – Part I

Interaction Diagrams

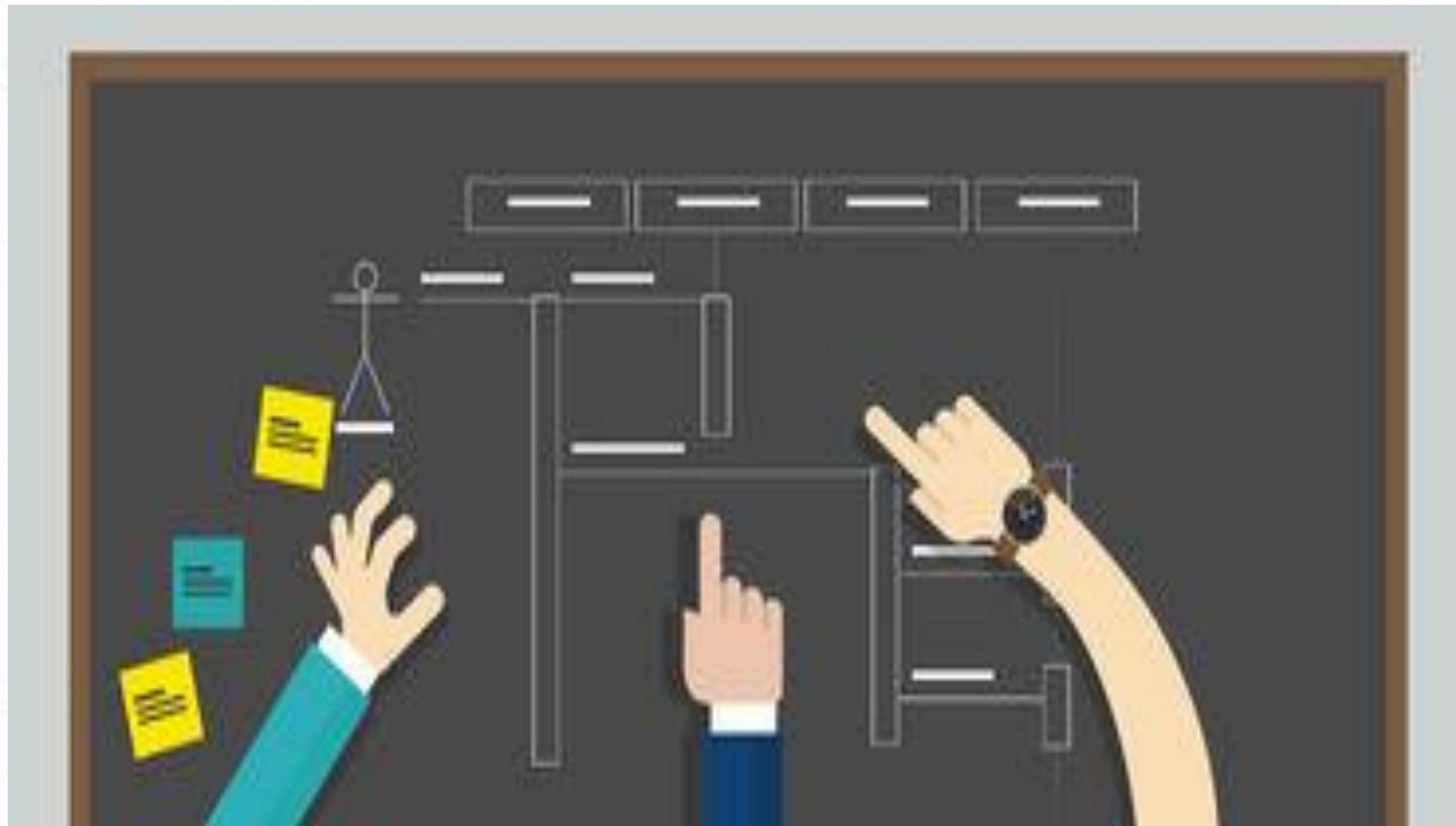


- An interaction diagram represents an interaction, which contains a set of objects and the relationships between them including the messages exchanged between the objects.

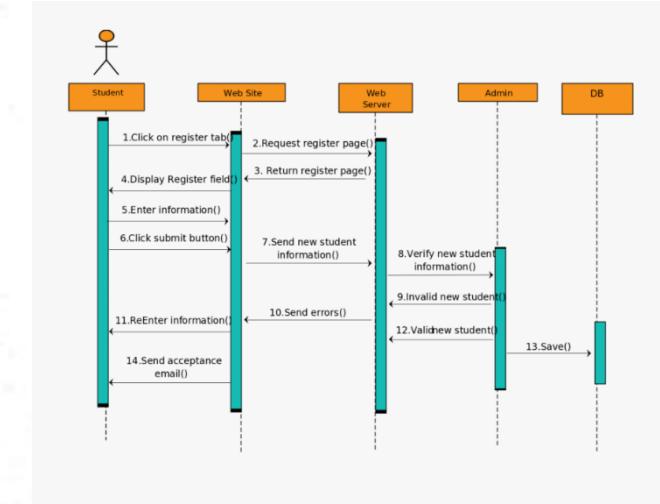
Interaction Diagrams

- Describes how groups of *objects* interact in some behavior.
- Typically, captures the behavior of a single use case.
- Two kinds of interaction diagrams:
 - Sequence diagrams
 - Communication diagrams-(Collaboration diagram)

Sequence Diagram



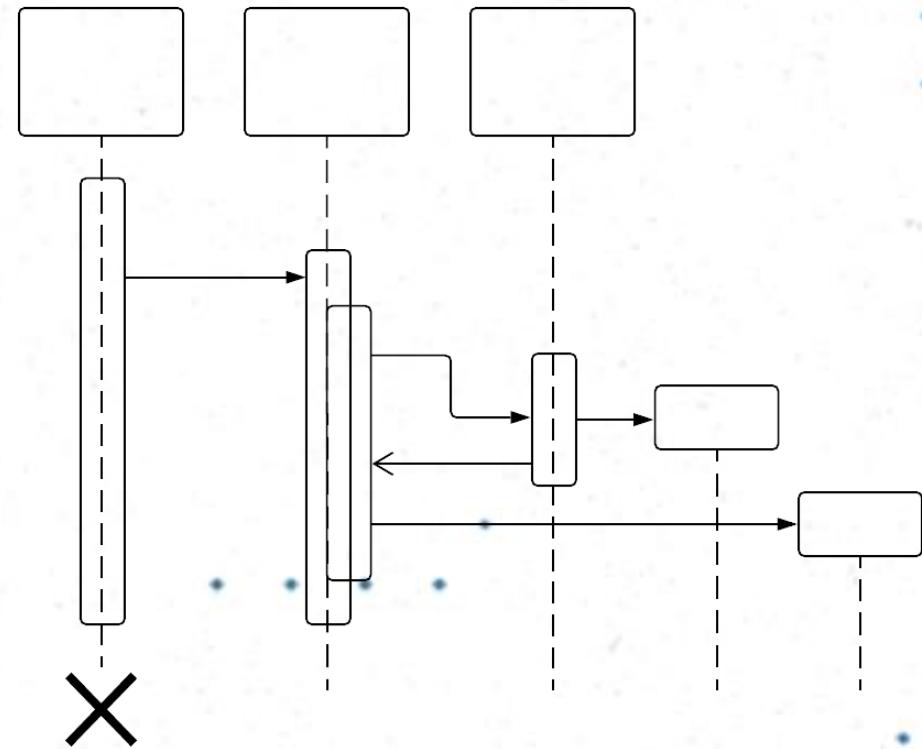
What is a Sequence Diagram?



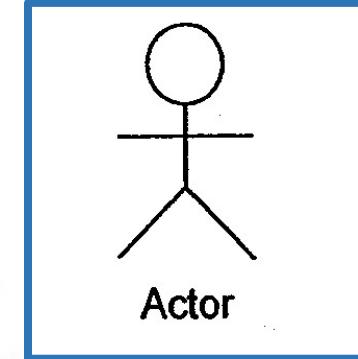
- A sequence diagram is one of the two interaction diagrams.
- The sequence diagram emphasizes on the time ordering of messages.
- In a sequence diagram, the objects that participate in the interaction are arranged at the top along the x-axis.

Elements of a Sequence Diagram

- Frame
 - Actor
 - Objects
 - Stereotypes
 - Lifeline
 - Execution bar (Activation bar)
 - Messages
 - Complex Interactions (Tags)



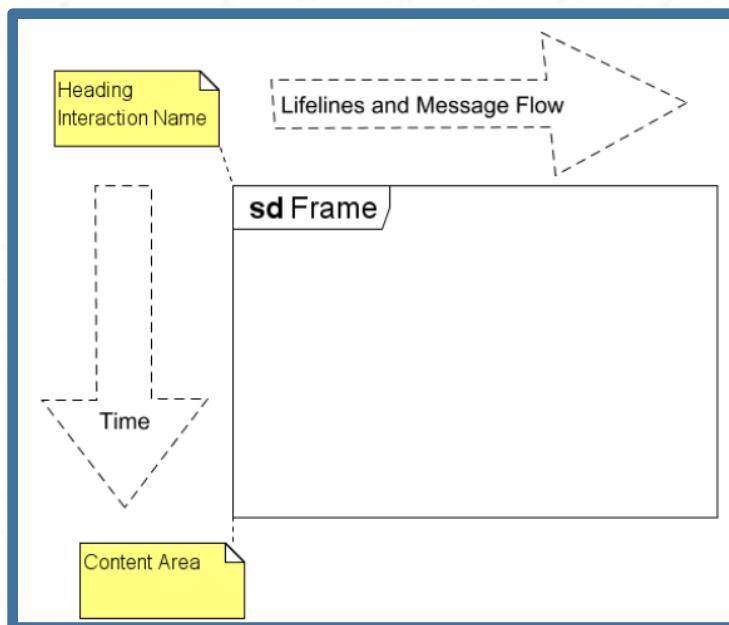
Actor



- Actors are the participants in a sequence sending and/or receiving messages.
- External to the subject.
- Represent roles played by human users, external hardware, or other subjects.

Frame

- A frame represents an interaction, which is a unit of behavior that focuses on the observable exchange of information between different objects.
- **sd** → Sequence Diagram <sd name of the diagram>



Objects

- Objects are shown in the same way as in UML object diagrams.
- A **named** object
(Named instance of a class)

Object name : Class name

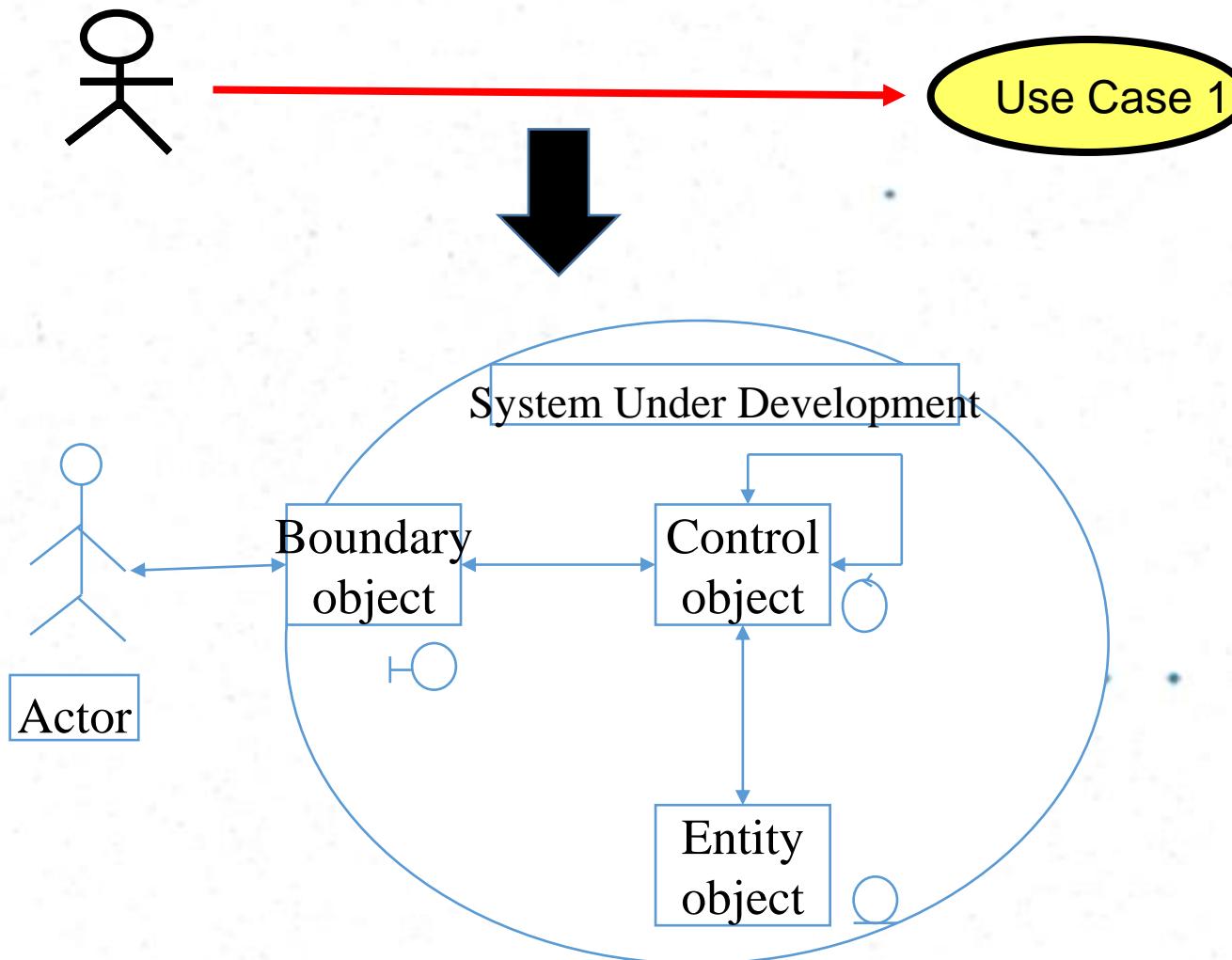
- An **anonymous** object
(Anonymous instance of a class)

: Class name

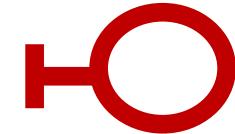
Class / Object Stereotypes

- Stereotypes differentiate the roles that classes / objects can play.
- Classes or objects can be classified into one of the following three stereotypes:
 - **Boundary** classes / objects - model interaction between the system and actors (and other systems)
 - **Entity** classes / objects - represent information and behaviour in the application domain
 - **Control** classes / objects - Co-ordinate and control other objects

Block Diagram: Class / Object Stereotypes

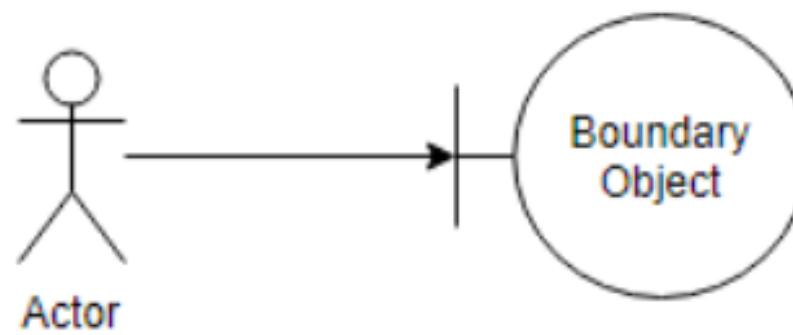
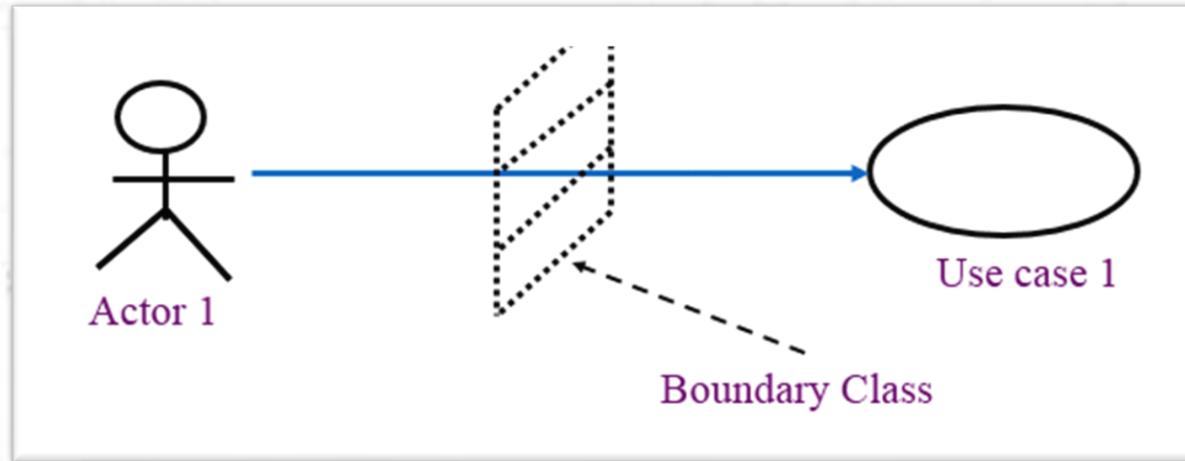


Boundary Classes / Objects

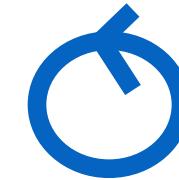


- Boundary is a stereotyped class or object that represents some system boundary,
 - e.g. a user interface screen, system interface .
- The interaction often involves inputs and outputs.
- It is often used in sequence diagrams which demonstrate user interactions with the system.
- This is a class / object used to model the interaction between the system and its actors.
- To find the Boundary classes, you can examine your use case diagram and scenario.
- At a minimum, **there must be at least one Boundary class / object for every actor-use case interaction.**

Boundary Classes / Objects cont...

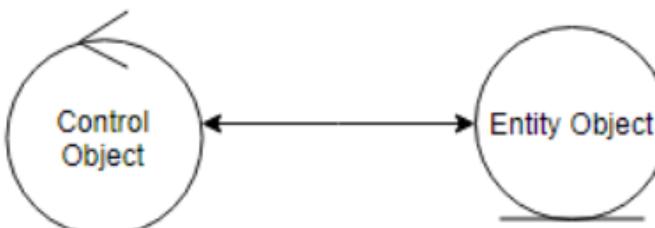
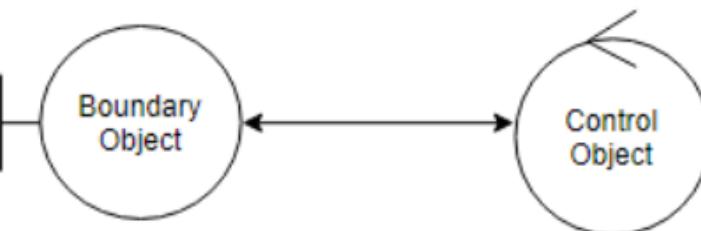


Control Classes / Objects

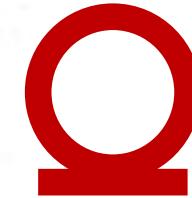


- Control classes represent coordination, sequencing, transactions and control of other objects
- It co-ordinates the events needed to realize the behaviour specified in the use case.
- Also, models complex computations and algorithms.
- Typically, they coordinate the movement (parameters) between boundary and entity classes.
- **There is at least one control class per use case**
 - Eg. Running or executing the use case.
- One or several control classes could describe use case realization.

Control Classes / Objects cont...



Entity Classes / Objects

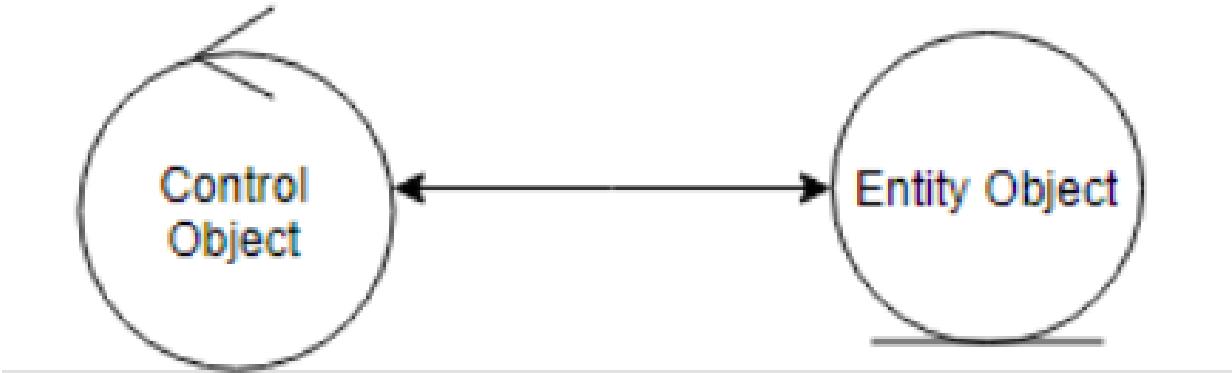


- Entity is a stereotyped class or object that represents some information or data.
- These classes are needed to perform tasks internal to the system.
- It reflects a real-world entity.

Eg:

- Account
- Student
- Lecturer, etc...

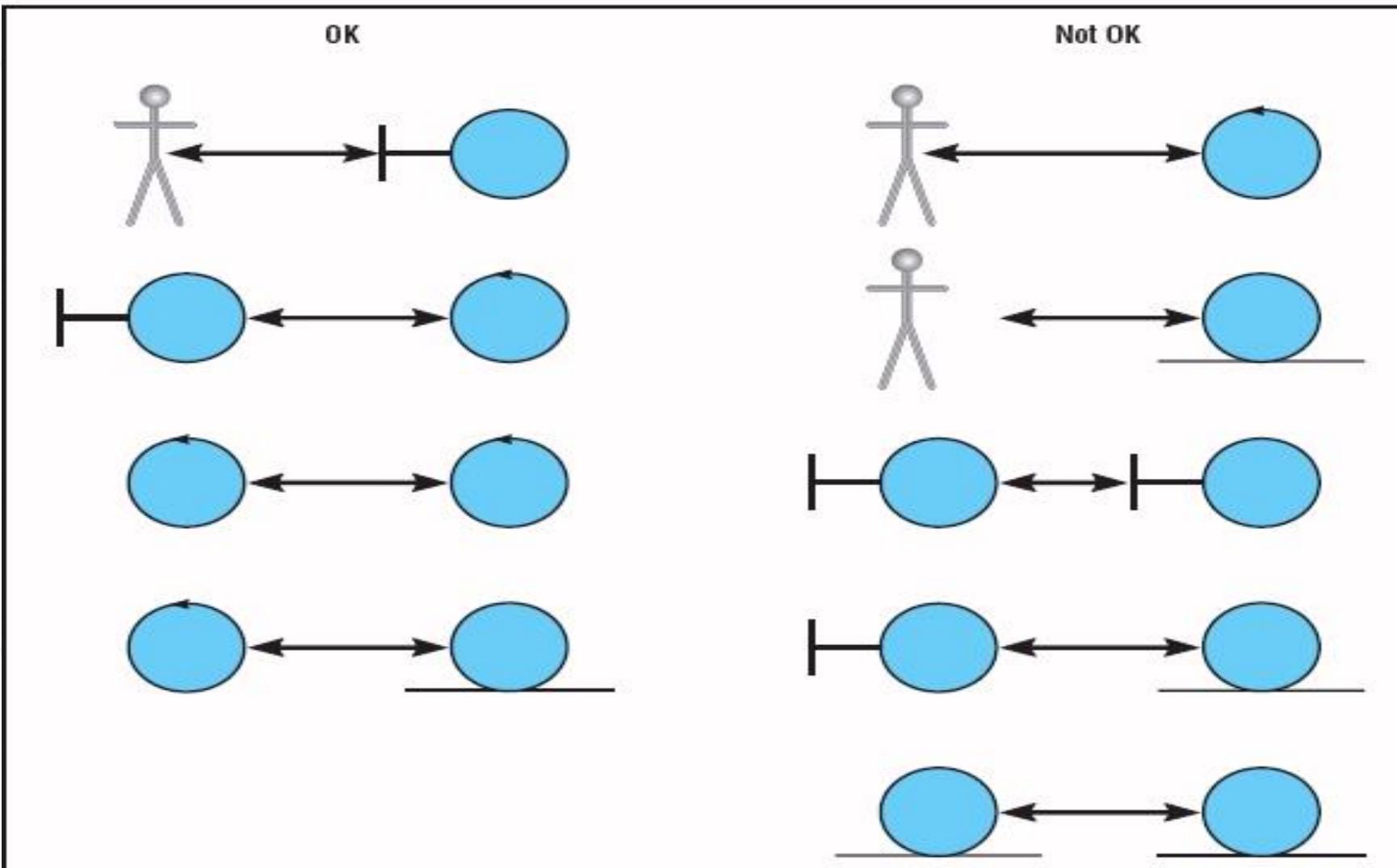
Entity Class / Objects cont...



How to Identify Class / Object Stereotypes

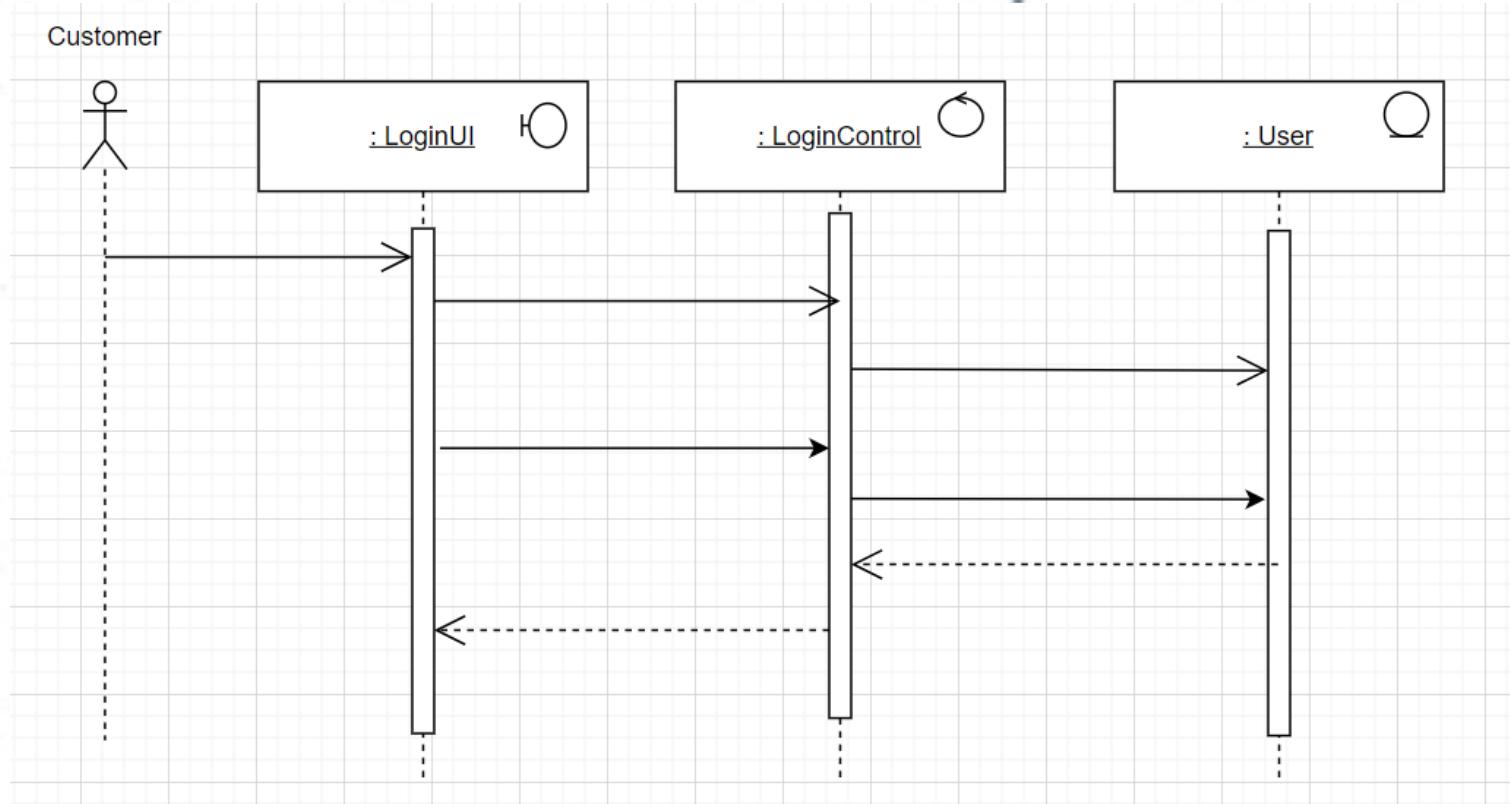
- Examine the models of a Use Case and identify **Entity classes** by examining the **data(information) storage requirements** of the Use Case.
- **Boundary classes** can identify by examining the **point of interaction of the actor with the system**.
 - One central **boundary class** for each actor. (primary window).
- **Control classes** can be identified by examining the **business process of the use case**.
 - One or more control class/es responsible for handling the control of the Use Case.

Rules of Stereotype Usage



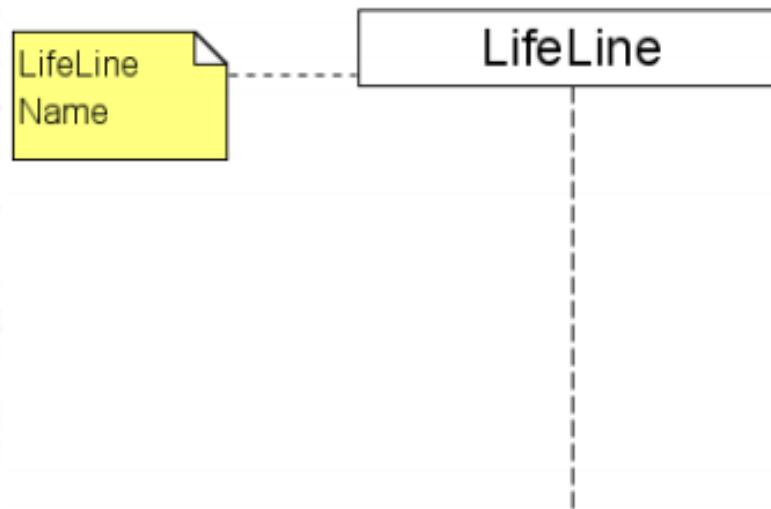
Stereotype Representation in Sequence Diagrams

- Each object in the sequence diagram better to indicate its stereotype and the objects should communicate as specified in the stereotype rules.



Lifelines

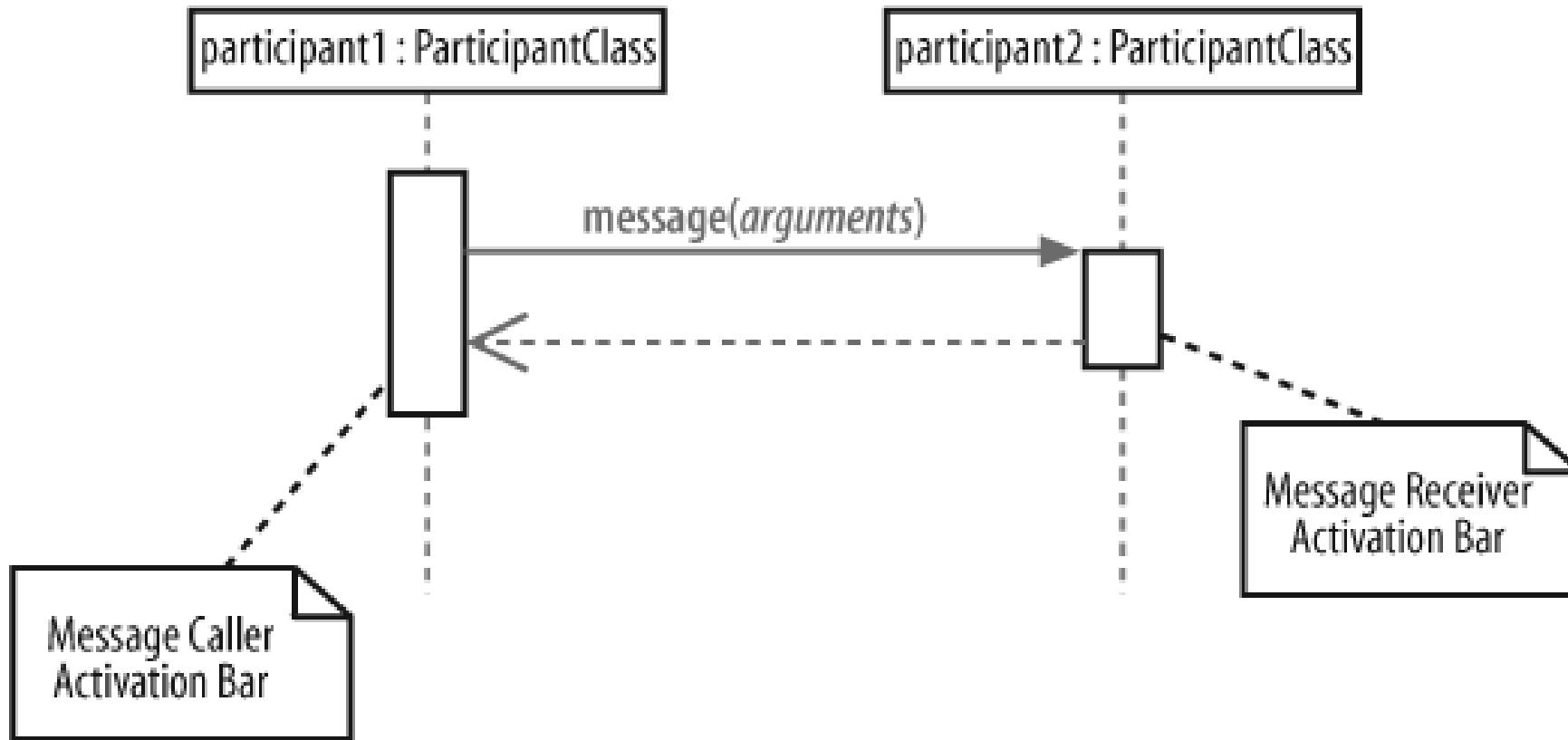
- Sequence diagrams are organized according to the time.
- Each participant/object has a corresponding lifeline.
- Each vertical dotted line is a lifeline, representing the time that an object exists.



Execution/Activation Bar

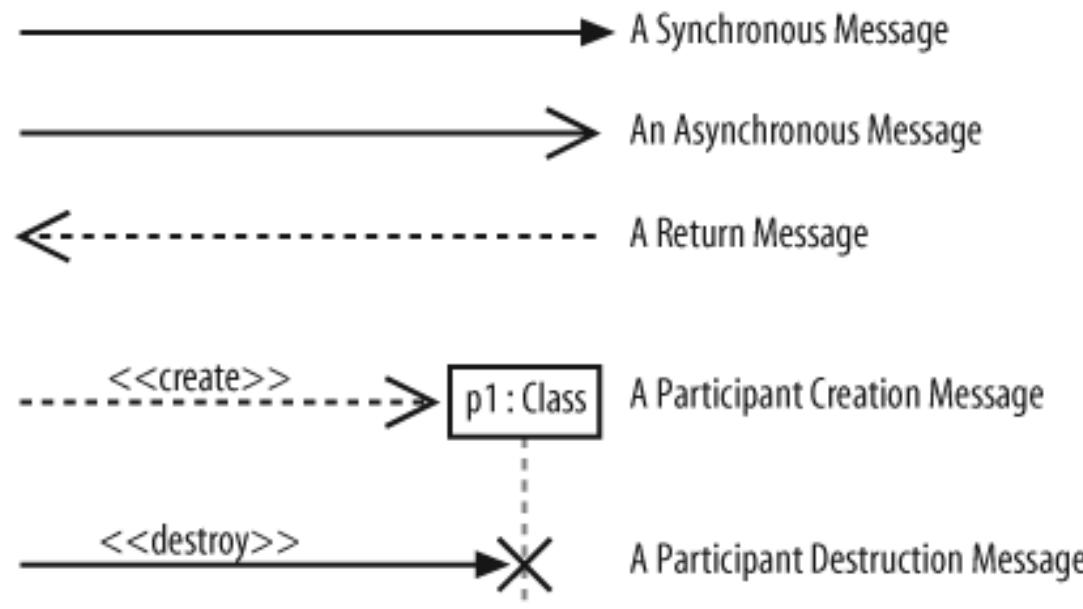
- When a message is passed to a participant it triggers, or invokes, the receiving participant into doing something; at this point, the receiving participant is said to be active.
- To show that a participant is active, i.e., doing something, you can use an activation bar, as shown below.

Execution/Activation Bar cont...



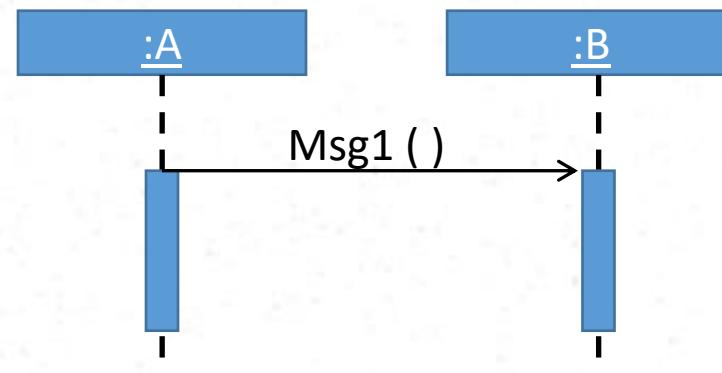
Messages

- Messages are used to illustrate communication between different active objects of a sequence diagram.



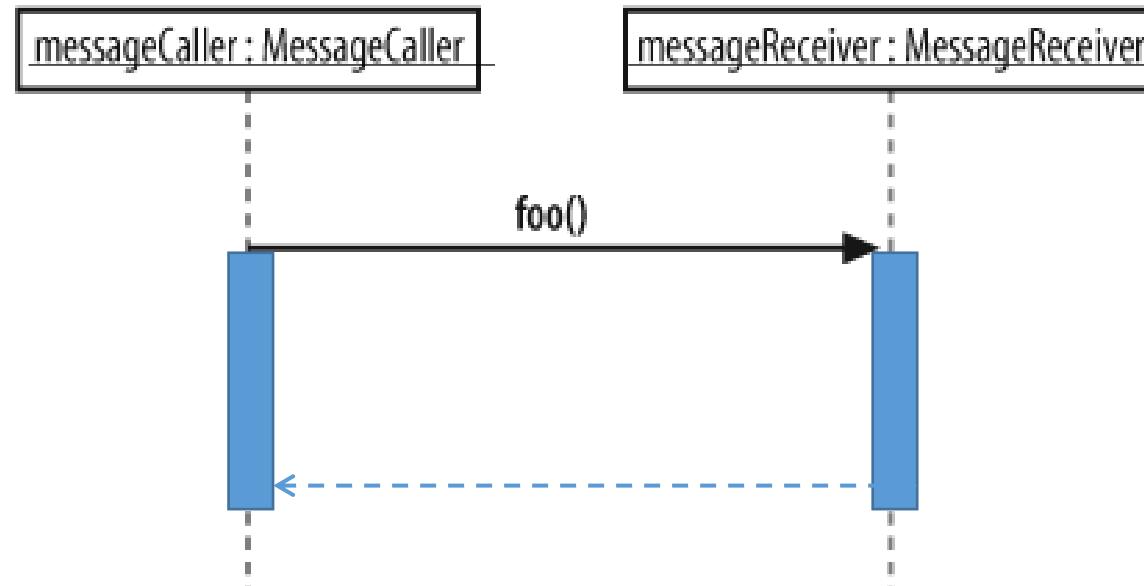
Message Passing

- Message passing is the invocation of a method in one object, by another method that belongs to a different object.
- When a message is sent from object A to object B:
 - object A asks object B to execute one of its methods (i.e. invoke the method);
 - object B receives the message and executes the method;



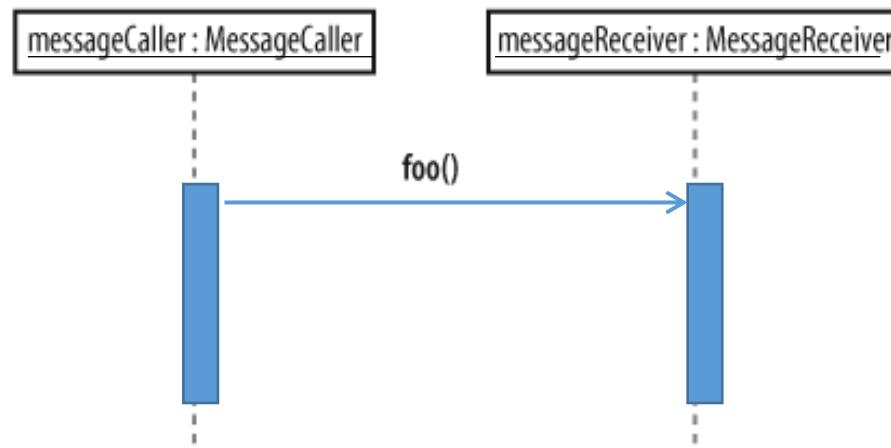
Synchronous Call

- Synchronous message is used when the sender waits until the receiver has finished processing the message.



Asynchronous Call

- **Asynchronous call** - The sender does not wait for the receiver to finish processing the message (for any return values), it continues immediately.
- An open arrowhead is used to asynchronous call.



Create and Delete Messages

- **Create Message**

- Create message to instantiate a new object in the sequence diagram.
- There are situations when a particular message call requires the creation of an object.
- It is represented with a dotted arrow and create word labelled on it to specify that it is the create Message symbol.



Create and Delete Messages cont...

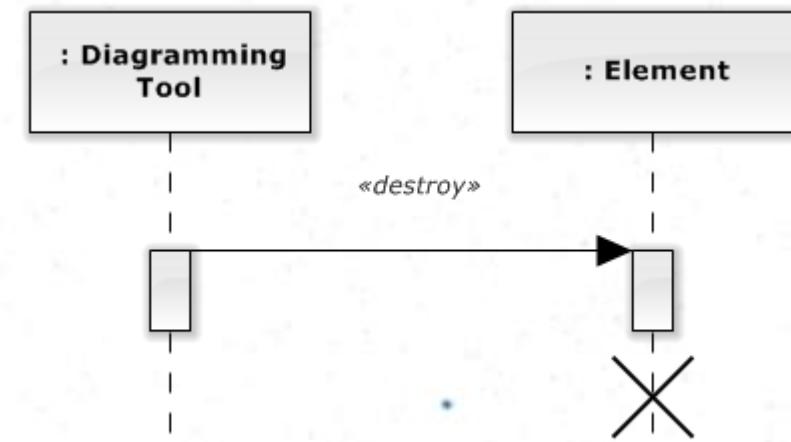
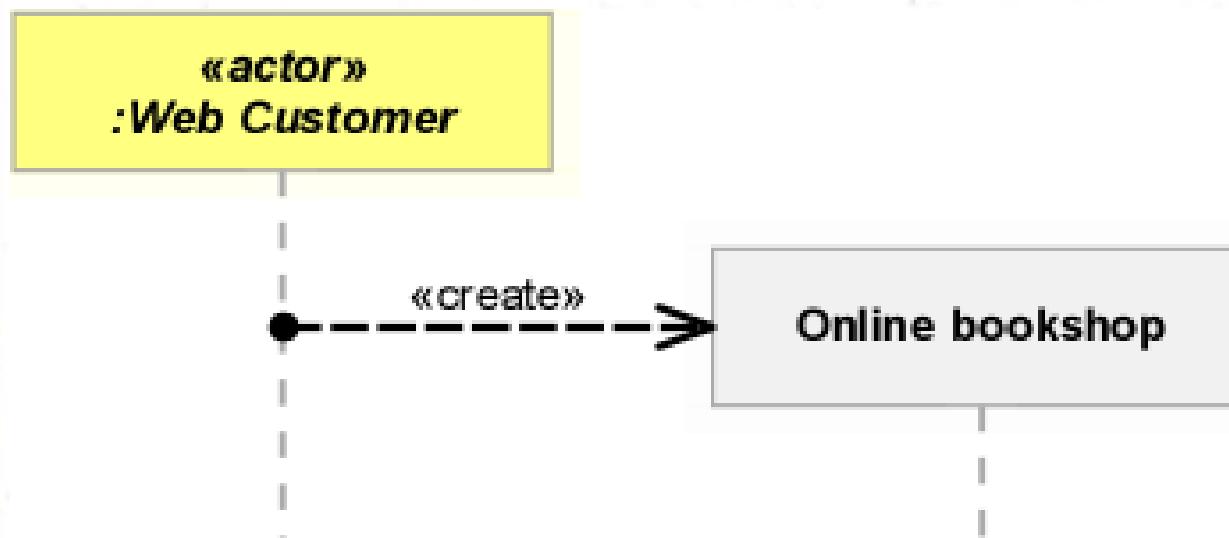
- **Delete Message**

- Use to remove / delete unwanted objects.
- In a garbage-collected environment, you don't delete objects directly, but it's still worth indicating when an object is no longer needed and is ready to be collected.

<<destroy>>



Create and Delete Messages cont...



Lost and Found messages

- **Lost Message :**

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



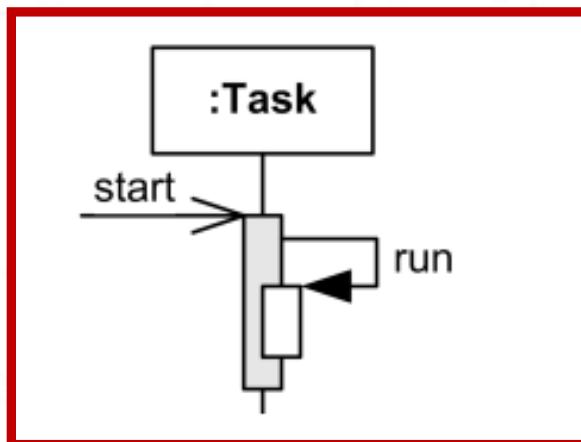
- **Found Message:**

- Found message indicates that although the receiver of the message is known in the current interaction fragment, the sender of the message is unknown.



Self Calls

- A **self message** is one method calling another method belonging to the same object. (message to the same lifeline)
- It is shown as creating a nested focus of control in the lifeline's execution occurrence.
- Represented by overlapping rectangles on the same lifeline.



Activity 1

- Draw a sequence diagram for a login process of an online ordering system.
- Note: You may use suitable boundary, control and entity classes.
 - Customer login using his/her username and password.
 - System request user details from the user DB and validates the username and password by comparing entered data with the user details in the user DB.
 - Then the system will display status of the user validation to the customer.

Activity 2

- The sequence of steps carried out in the "**Manage course information**" flow are given below. Draw a sequence diagram for the given description.
 - The system administrator wants to do some changes to a course. He clicks on the Manage Course icon in Manage Course UI which in turn invokes the ManageCourse functionality of CourseControl class.
 - The ManageCourse function of the CourseControl class first modifies the topic of the course through TopicModification function of the Topic class.
 - Then, ManageCourse functionality invokes CourseModification functionality of a Course.
 - Finally, the system administrator invokes AssignTutor function of the Tutor, to assign a tutor to the selected course.

Note:

- Assume all are Asynchronous messages.
- Use suitable boundary, control and entity classes.

Activity 3

Draw a sequence diagram for the given “Inquire book status” scenario by identifying stereotypes.

Name: Inquire book status

Brief Description: Handle from the borrower about availability of a book.

Actor: Librarian

Flow of Events:

1. Library member clicks “SerachBook” option in the Library UI by giving the Book ID.
2. Then LibraryController class will call SearchBook method in the Book class.
3. Then the system will send book status to the library member.
4. Next, the library member click reserve option in the UI and the LibraryController will call ReserveBook function of Book class.
5. Finally, Book class will send the reserve status to the library member.

References

- UML 2 Bible
 - Chapters 8 & 9
- Learning UML 2.0 by Kim Hamilton, Russ Miles
- Applying UML and Patterns by Craig Larman
- Chapter 15UML 2 Bible
 - Chapters 8 & 9
- TheElements of UML 2 Style
 - Chapter 7

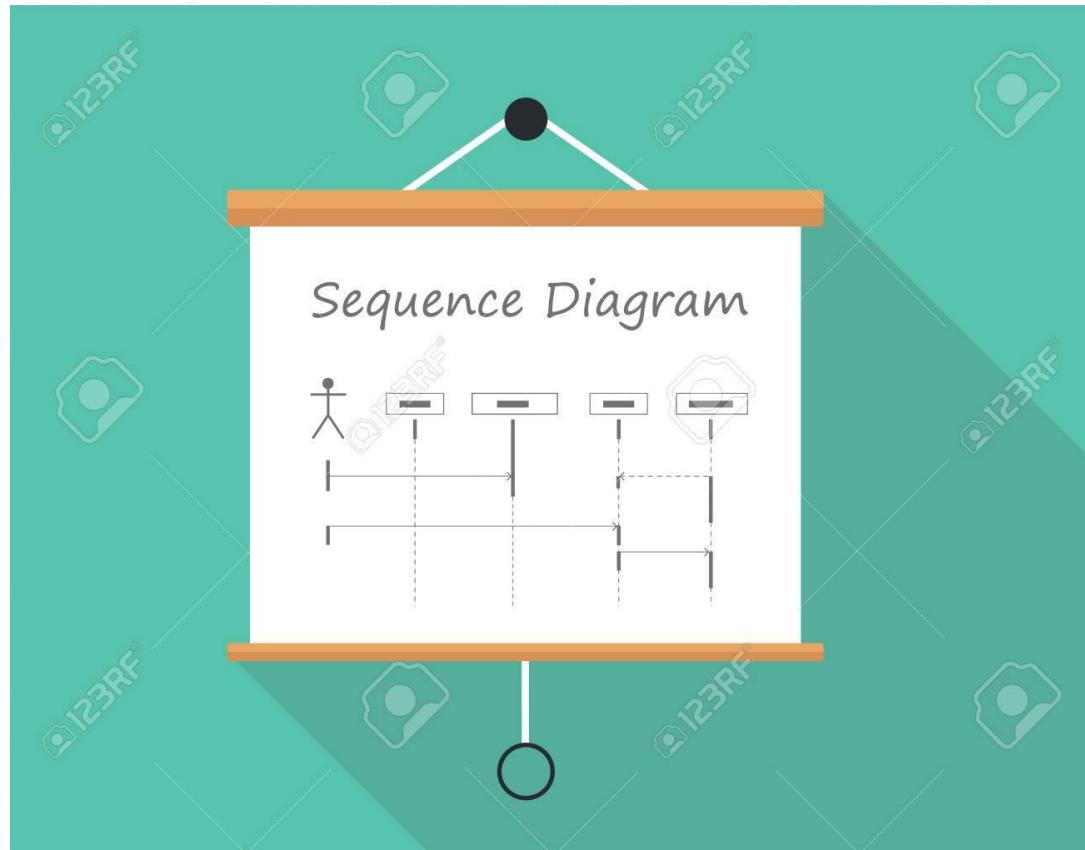
Thank you!!!

Software Engineering (IT2020)

2022

Lecture 2 - Interaction Diagrams – Part II

Interaction Diagrams



- An interaction diagram represents an interaction, which contains a set of objects and the relationships between them including the messages exchanged between the objects.

Complex Interactions with Sequence Diagrams

- Sequence diagrams may contain, essentially, sub diagrams called **interaction fragments**.
- Each interaction fragment can have an operator, such as loop, opt (“optional”), alt (“alternative”), ref (“reference”), para (“parallel”), and so on.
- These interaction fragments and operators greatly enhance the ability of sequence diagram.

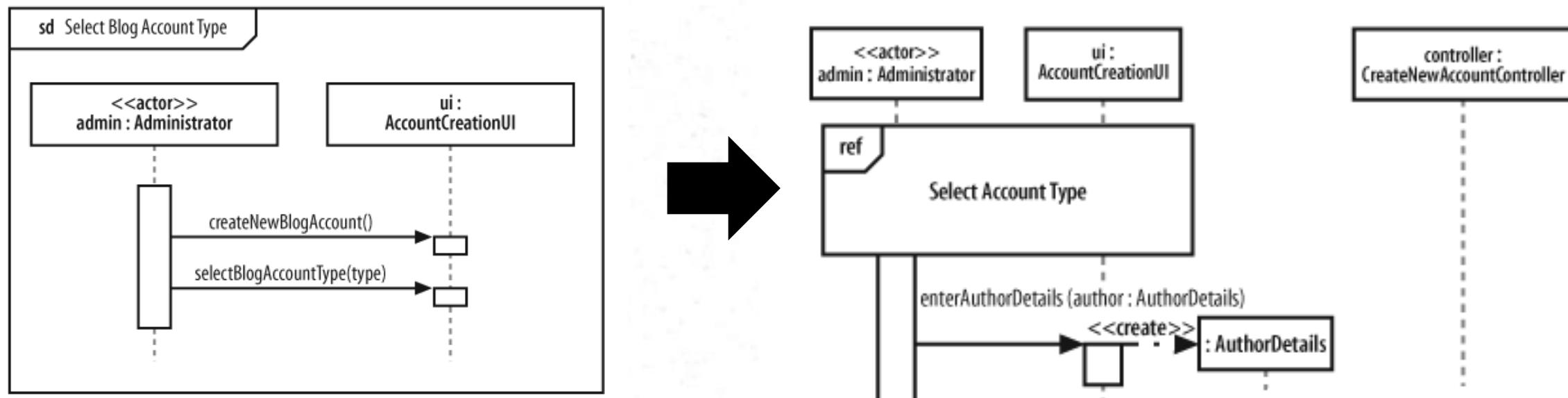
Common Interaction Fragments

- Ref - Reference
- alt - alternatives
- opt - option
- loop - iteration
- break - break
- par - parallel

Reference Fragment - Ref Tag

- If one sequence diagram is too large or refers to another diagram, ‘ref’ tag can be used.
- Helps you to manage a large diagram by splitting, and potentially reusing, a collection of interactions.
- Reference fragment that uses or calls another interaction. It is shown in the sequence diagram as a fragment with the operator ref to indicate the reference to another interaction.

Reference Fragment - Ref Tag cont...



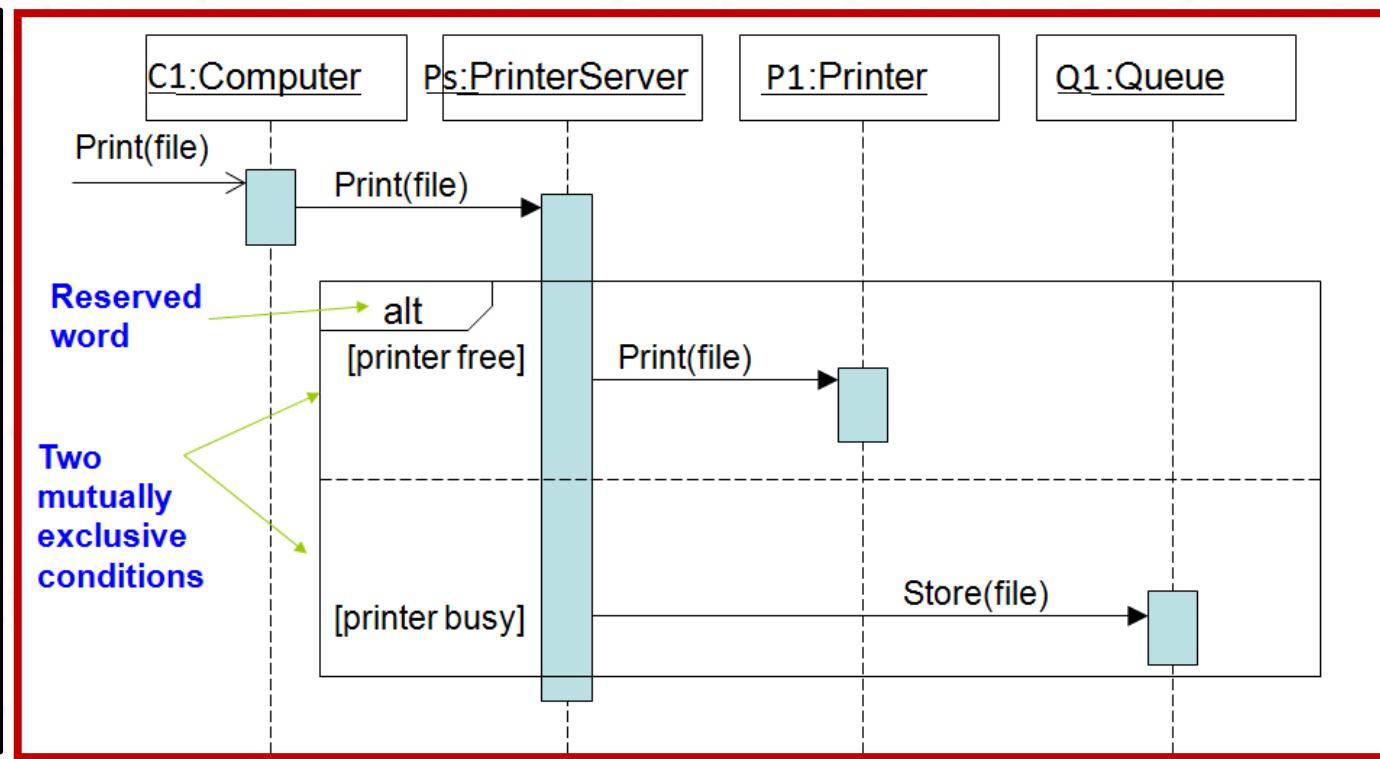
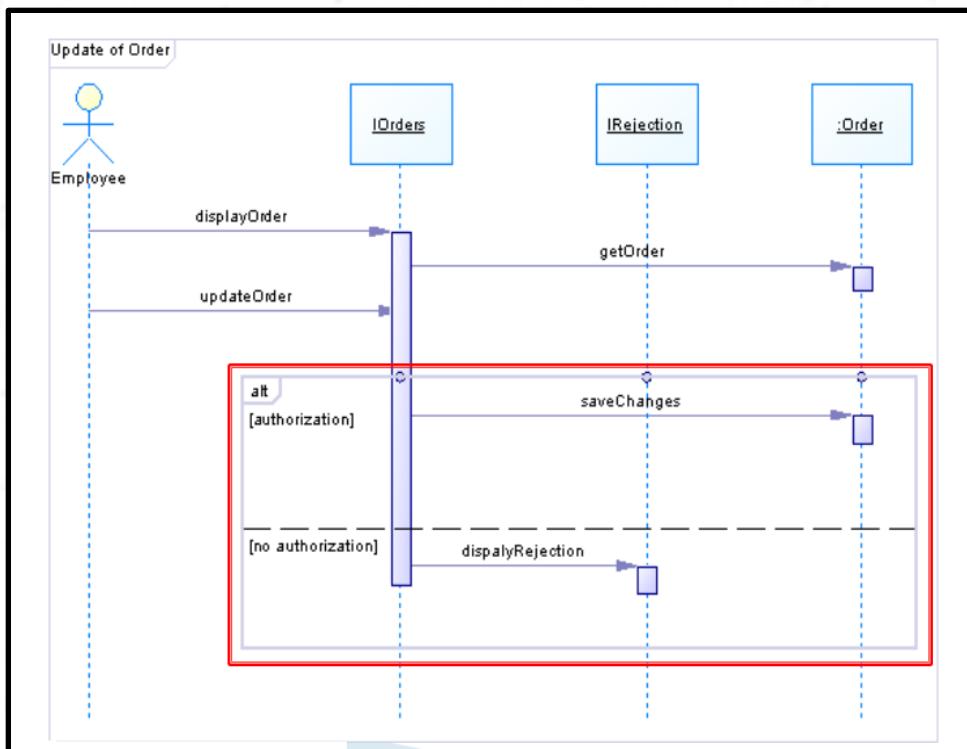
Activity 01

- Customer needs to log in to the system to purchase an e-ticket for cinema.
- After the login, customers request movie details via MovieUI and the request will pass to the MovieManager.
- Then MovieManager request the details from the MovieDB and pass to the customer.
- Then customer can request to book with MovieID and MovieManager generate the ticket and send to the customer. Then MovieManager update the seat count via the SeatInfo class.
- Assume that the logging process is already designed and you have to use it in your diagram.

Alternative Fragment – Alt tag

- **Alt** : Alt can have number of guard conditions known as alternatives.
- At one time one of the alternatives will be true and the messages related to that condition will execute.
- There can be an else clause, which will execute whenever none of the other options are selected.
- Alt tag can have **two or more** alternatives.

Alternative Fragment – Examples



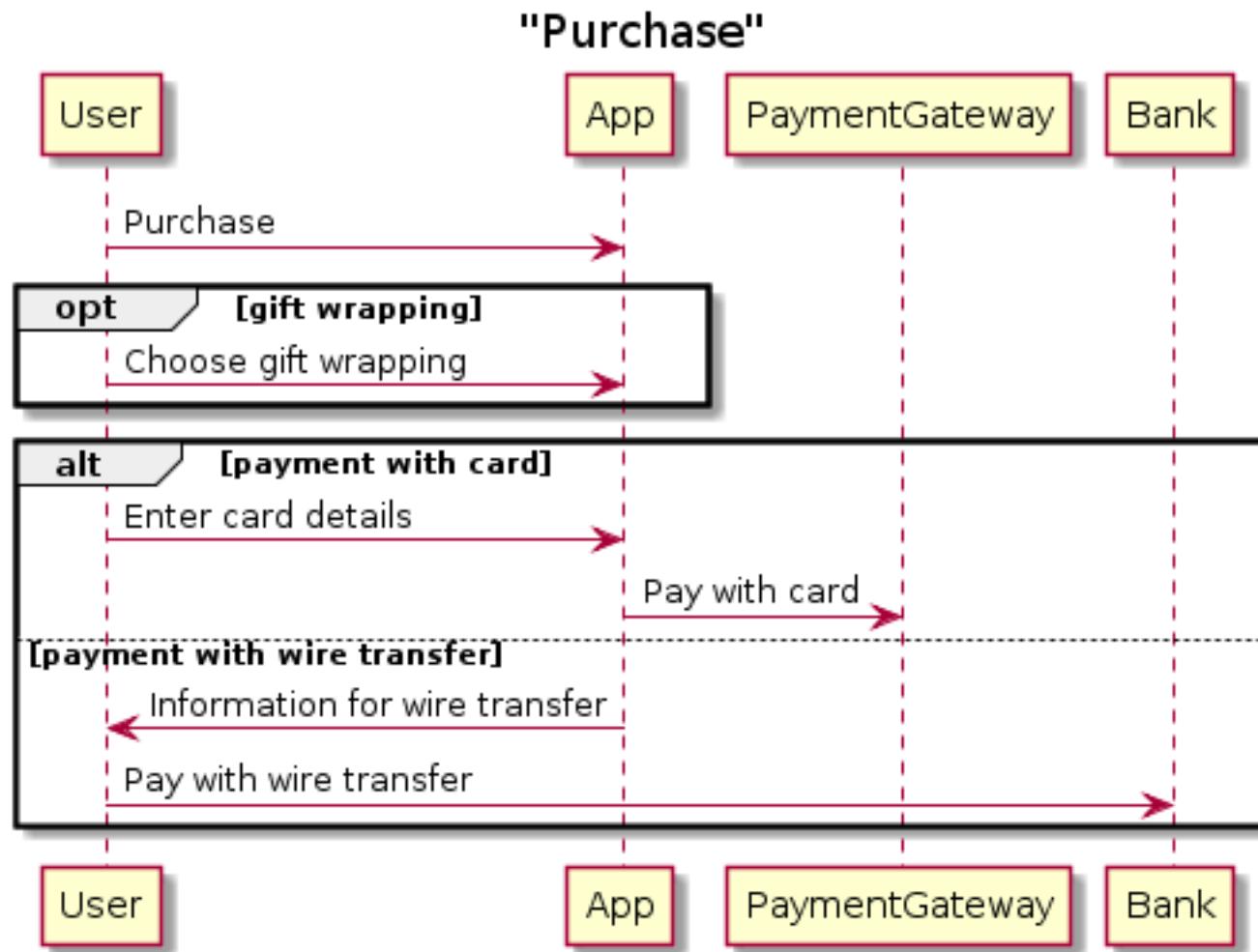
Activity 02

- Draw a sequence diagram for a login process of an online ordering system.
- Note: You may use suitable boundary, control and entity classes.
 - Customer login using his/her username and password.
 - System request user details from the user DB and validates the username and password by comparing entered data with the user details in the user DB.
 - If he/she is a valid user, the system will display a success message and else ask to relogging.

Optional Fragment – Opt Tag

- The opt (optional) interaction operator use to handle a single condition situation.
- The model for an opt combined fragment looks like an alt that offers only one interaction.
- To be used, the guard condition must be satisfied.
- If the guard condition fails, the behavior is simply skipped.

Optional Fragment – Example



Activity 03

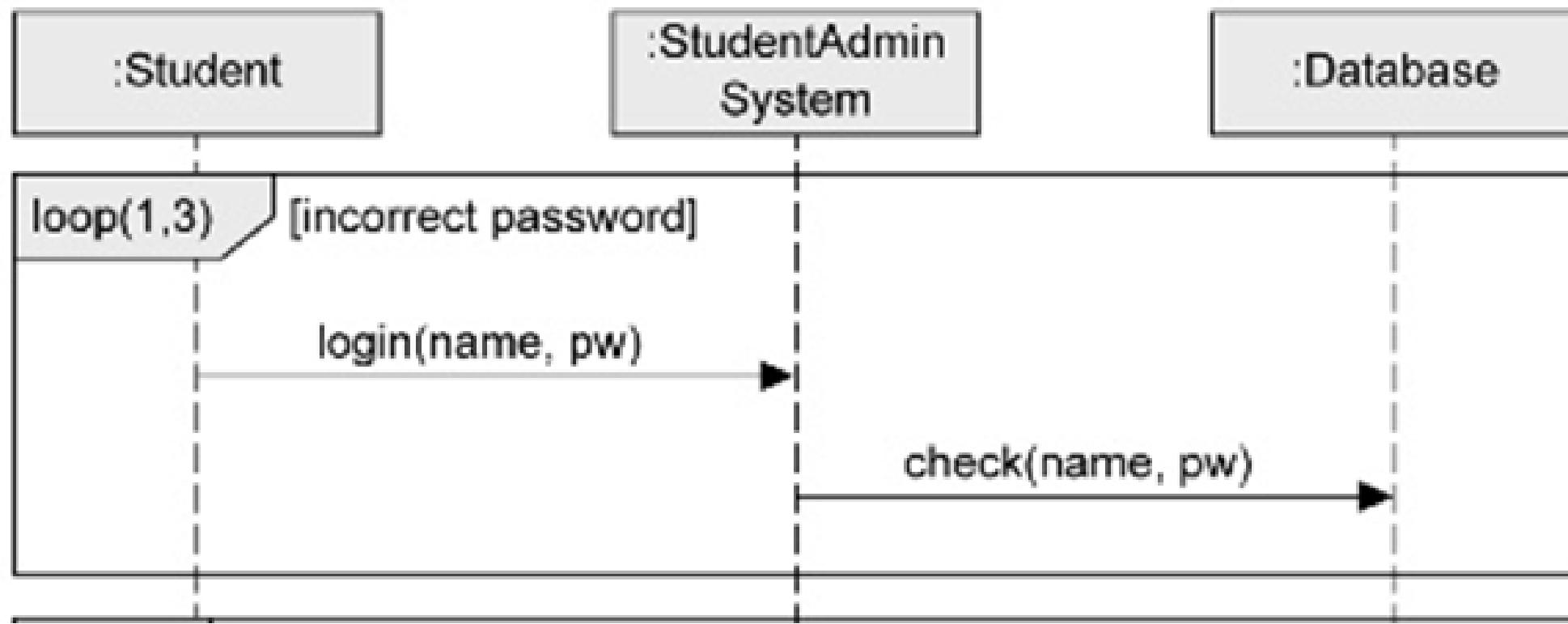
- “ABC” cinema has a eTicket booking system. Customer can request a particular movie details via MovieUI and the request will pass to the MovieManeger.
- Then MovieManager request the details from the MovieDB and pass the results to the customer.
- Only if the requested movie is available, the customer can request to book with MovieID and MovieManager generate the ticket and send to the customer.



Loop Fragment – Loop Tag

- Loop fragment is used to represent a repetitive sequence.
- Place the words ‘loop’ in the name box and the guard condition near the top left corner of the frame.
- The guard condition in a loop fragment can have two other special conditions tested against. These are minimum iterations (min int) and maximum iterations (max int).
- When the guard condition become false, the loop ends.

Loop Fragment – Example

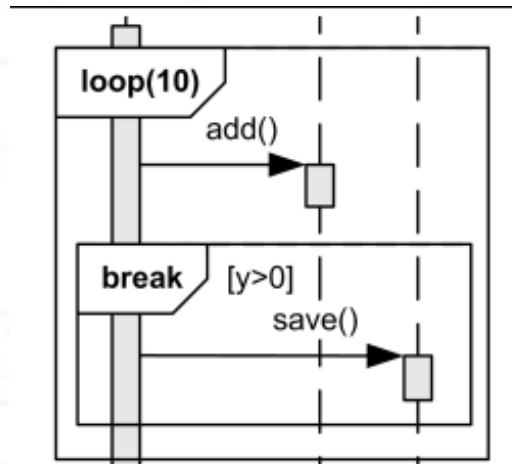


Activity 04

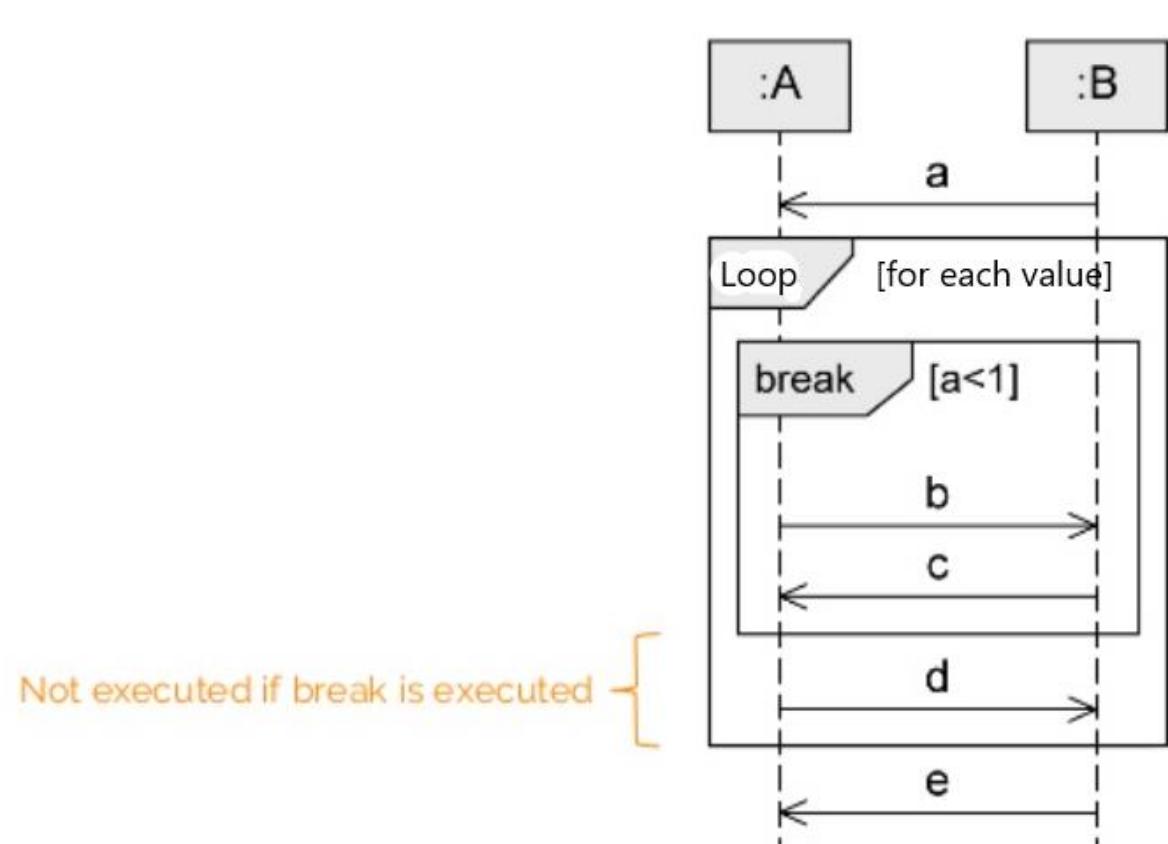
- “ABC” cinema has a eTicket booking system. Customer can request a particular movie details via MovieUI and the request will pass to the MovieManager.
- Then MovieManager request the details from the MovieDB and pass the results to the customer.
- Only if the requested movie is available, the customer can request to book with MovieID and MovieManager generate the ticket and send to the customer.
- In the same way, customer can make any number of requests for eTickets for different movies.

Break Fragment – Break Tag

- If Break fragment is executed, the rest of the sequence is abandoned.
- You can use the guard to indicate the condition in which the break will occur.
- The break interaction operator provides a mechanism similar to the break syntax in many programming languages.
- Breaks are most commonly used to model exception handling.



Break Fragment - Example

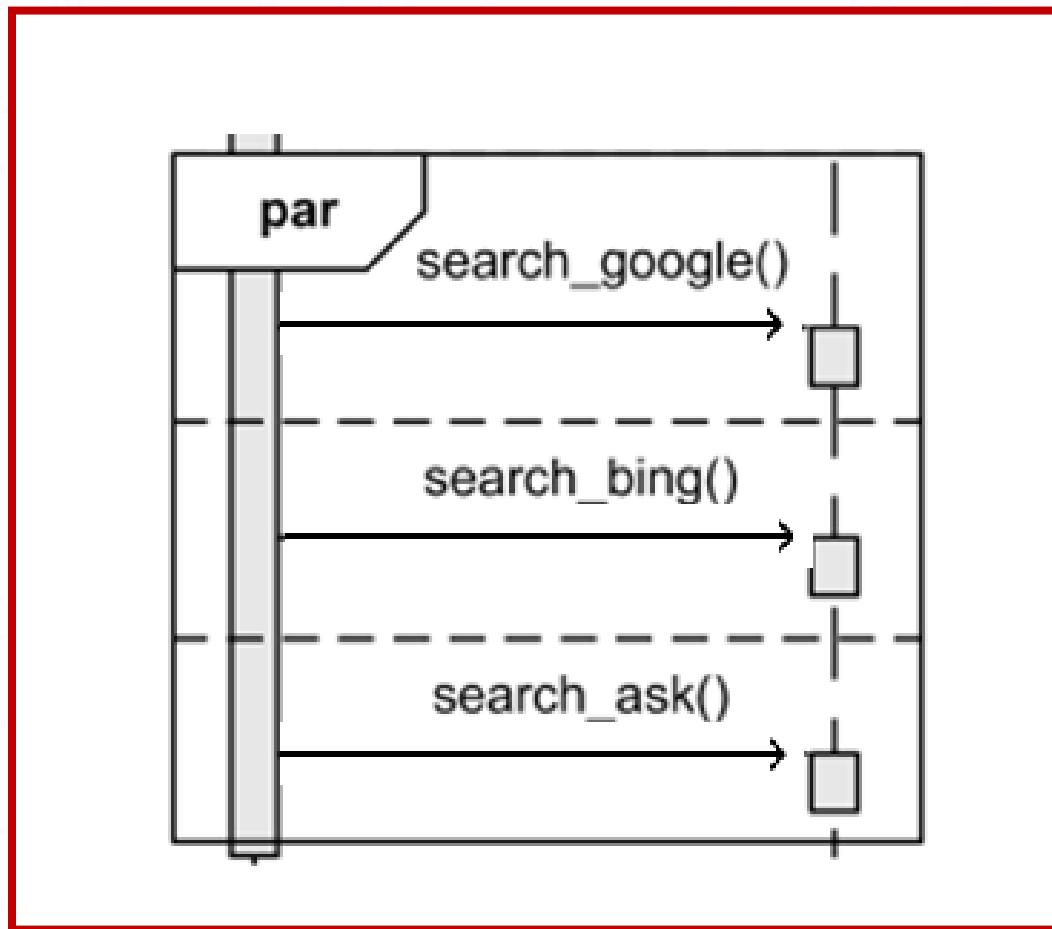


Source: <https://www.slideshare.net/iivanoo/modeling-objects-interaction-via-uml-sequence-diagrams-software-modeling-computer-science-vrije-universiteit-amsterdam-20162017>

Parallel Fragment – Par Tag

- When the processing time required to complete portions of a complex task is longer than desired, some systems handle parts of the processing in parallel.
- The interaction operator "par" defines parallel execution of behaviors.
- Different operands can be interleaved in any way as long as the ordering imposed by each operand is preserved.

Parallel Fragment - Example



Search Google, Bing and Ask in any order, possibly parallel.

References

- UML 2 Bible
Chapters 8 & 9
- Learning UML 2.0 by Kim Hamilton, Russ Miles
- Applying UML and Patterns by Craig Larman
- Chapter 15UML 2 Bible
Chapters 8 & 9
- TheElements of UML 2 Style
Chapter 7



SLIIT

Discover Your Future

Software Engineering (IT2020)

2022

Lecture 3 - Communication Diagram

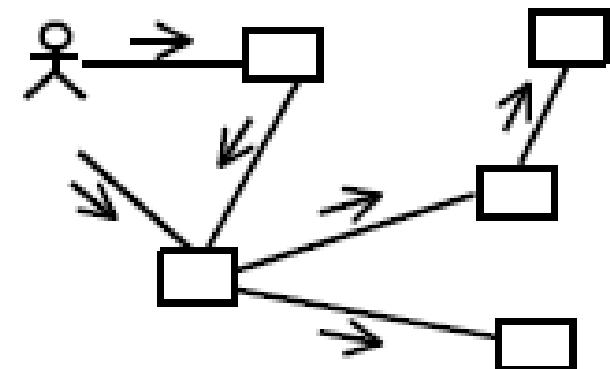


Session Outcomes

- Symbols of communication diagrams
 - Objects
 - Links
 - Messages and directions
 - Message sequence numbers
- Iteration and Looping
- Guard Expressions
- Parallel Activities

What Is a Communication Diagram?

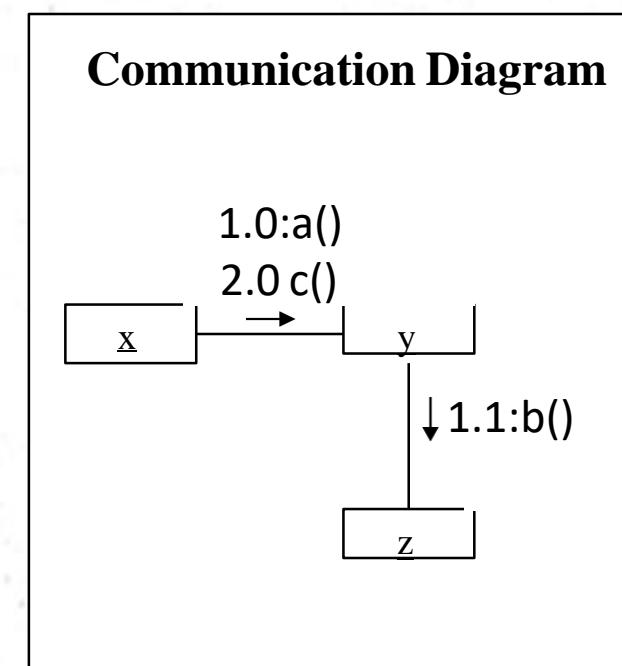
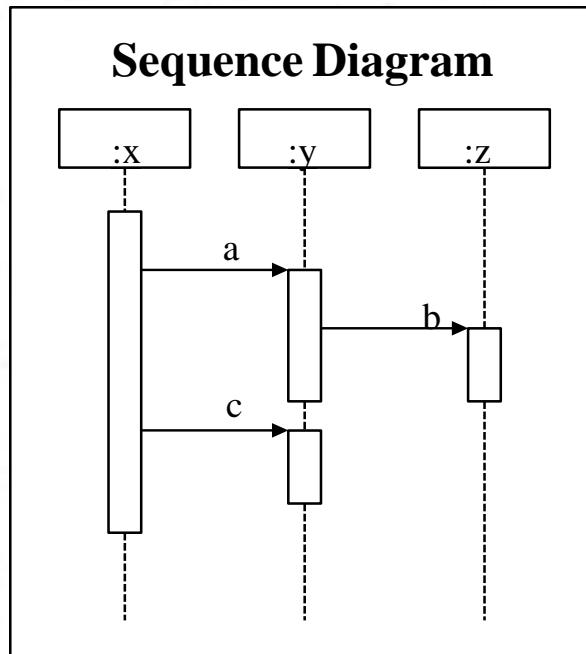
- A communication diagram emphasizes the organization of the objects that participate in an interaction.
 - The communication diagram shows:
 - The objects participating in the interaction.
 - Links between the objects.
 - Messages passed between the objects.



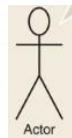
Communication Diagrams

Sequence and Communication Diagrams

- Interaction diagrams
 - Sequence diagram (temporal focus)
 - Communication diagram (structural focus)



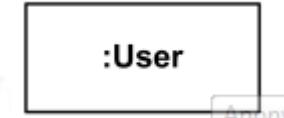
Symbols of Communication Diagram



Actors : Each Actor is named and has a role



Placed anywhere



Links between objects

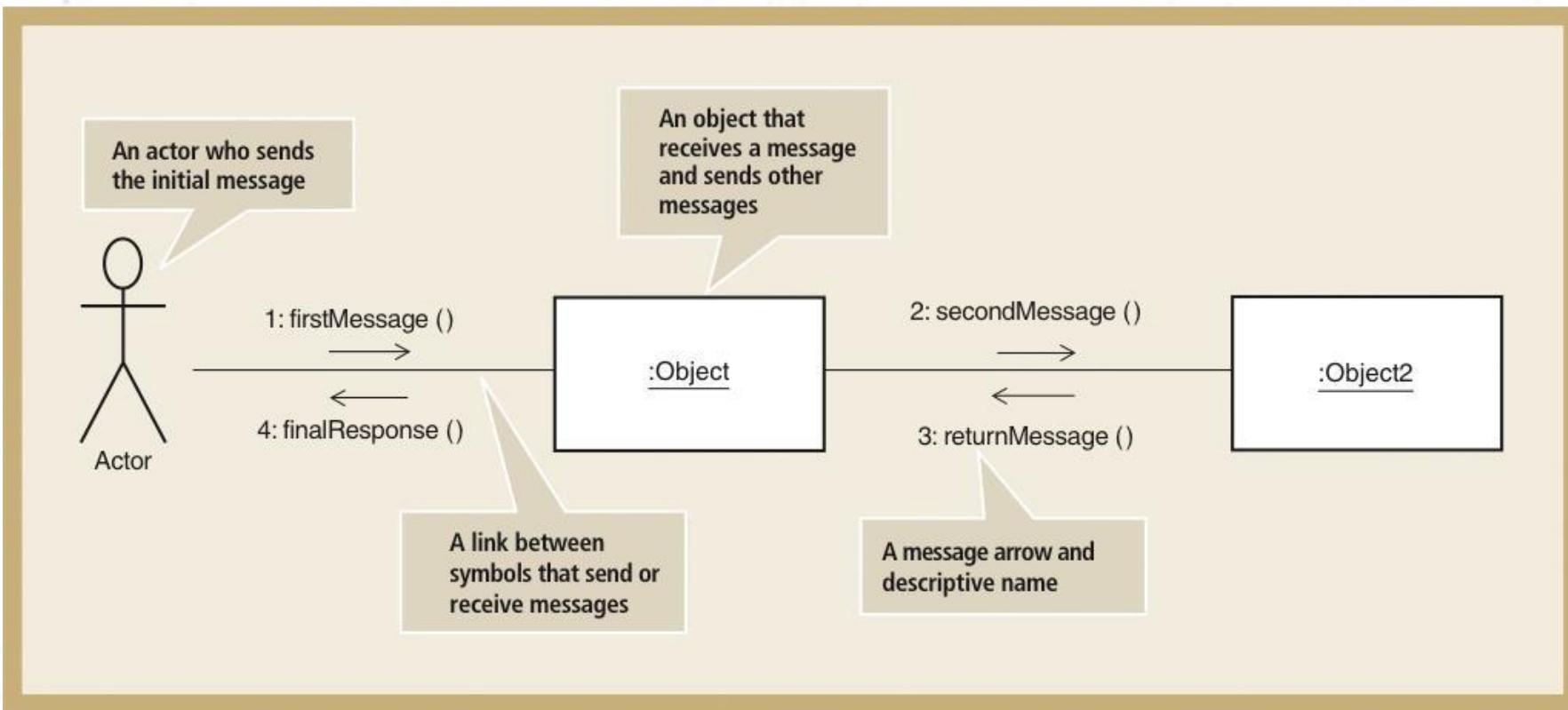


Direction of messages from one object to another object.

1, 2, 2.1, 2.2

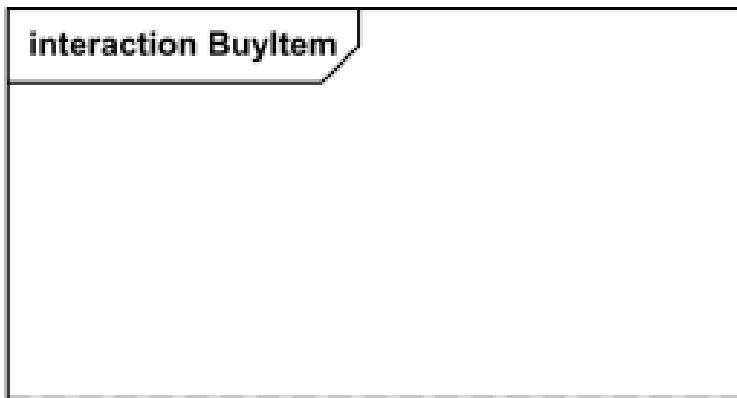
Message sequence numbers.

Communication Diagram - Example



Frame

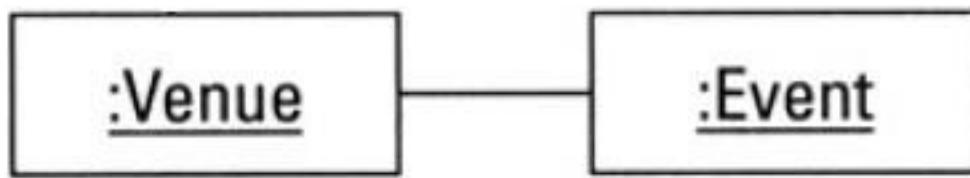
- Communication diagrams could be shown within a rectangular frame with the diagram name in the name box preceding with the “interaction” keyword.



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

Objects and Links

- Objects : Similar to Sequence Diagram.
- The connecting lines drawn between objects are links.
- They enable you to see the relationships between objects.
- This symbolizes the ability of objects to send messages to each other.
- A single link can support one or more messages sent between objects



Messages

- The message types in a Communication diagram are the same as in a Sequence diagram.

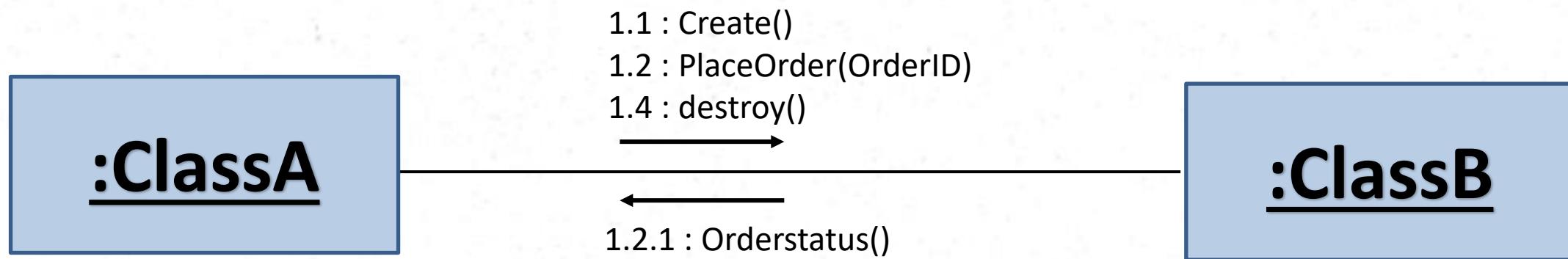
Message Syntax

Message Sequence Number : Message signature

e.g. 1.0 : Login (UserName, Pwd)
 3.1.1 : getPerformance ()

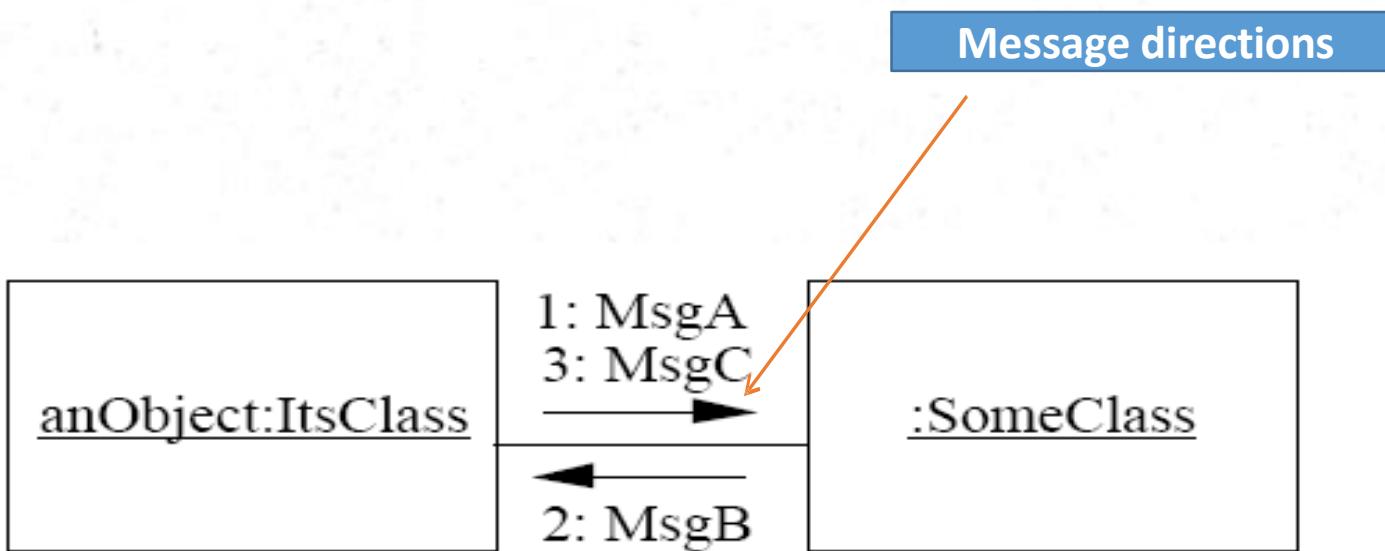
Message Types

In a communication diagram all the message types (Synchronous, Asynchronous, Create, Destroy and Reply) indicate in the same way.

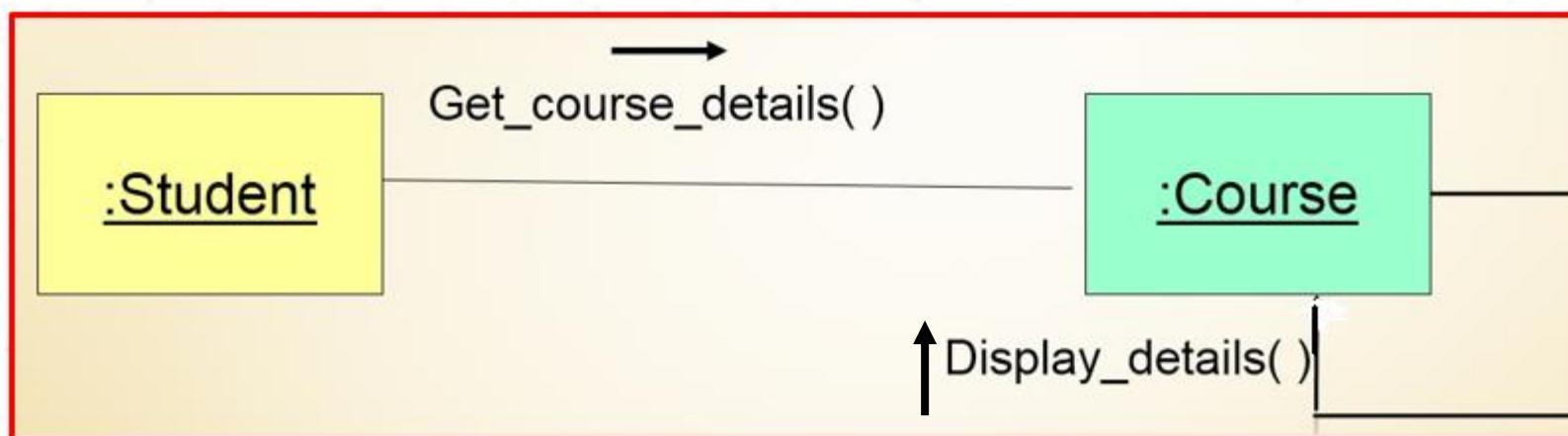


Message Directions

- A message on a communication diagram is shown using an arrow from the message sender to the message receiver.



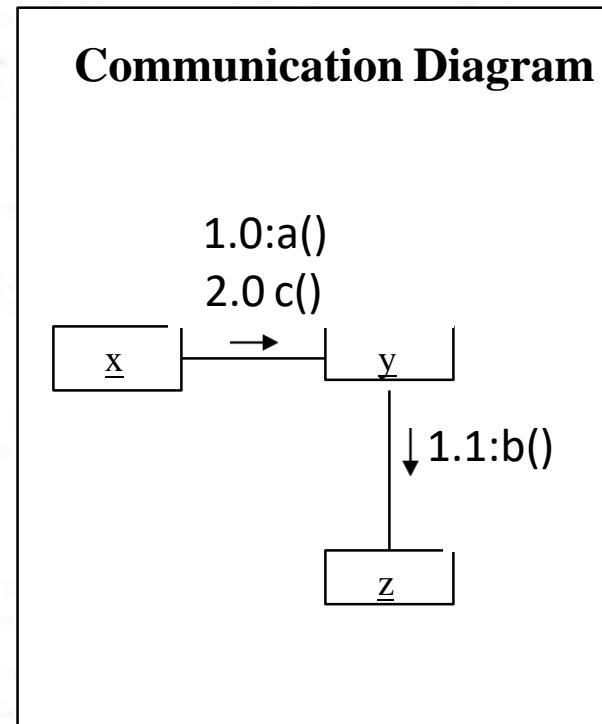
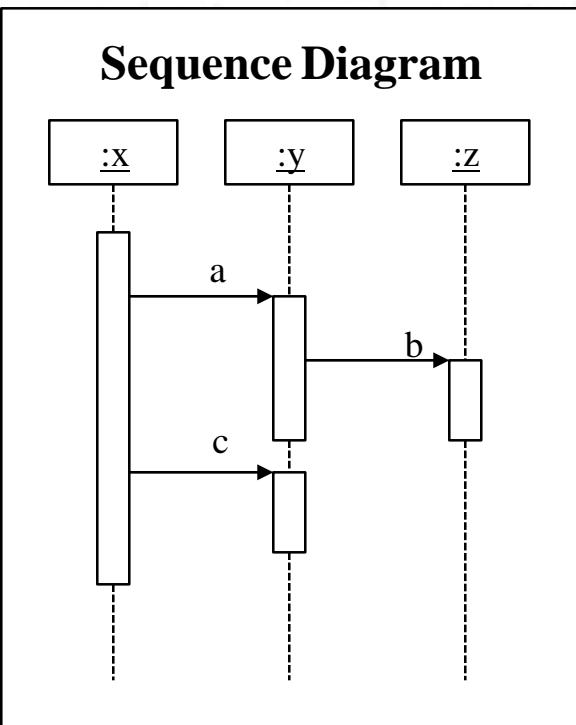
Self Calls



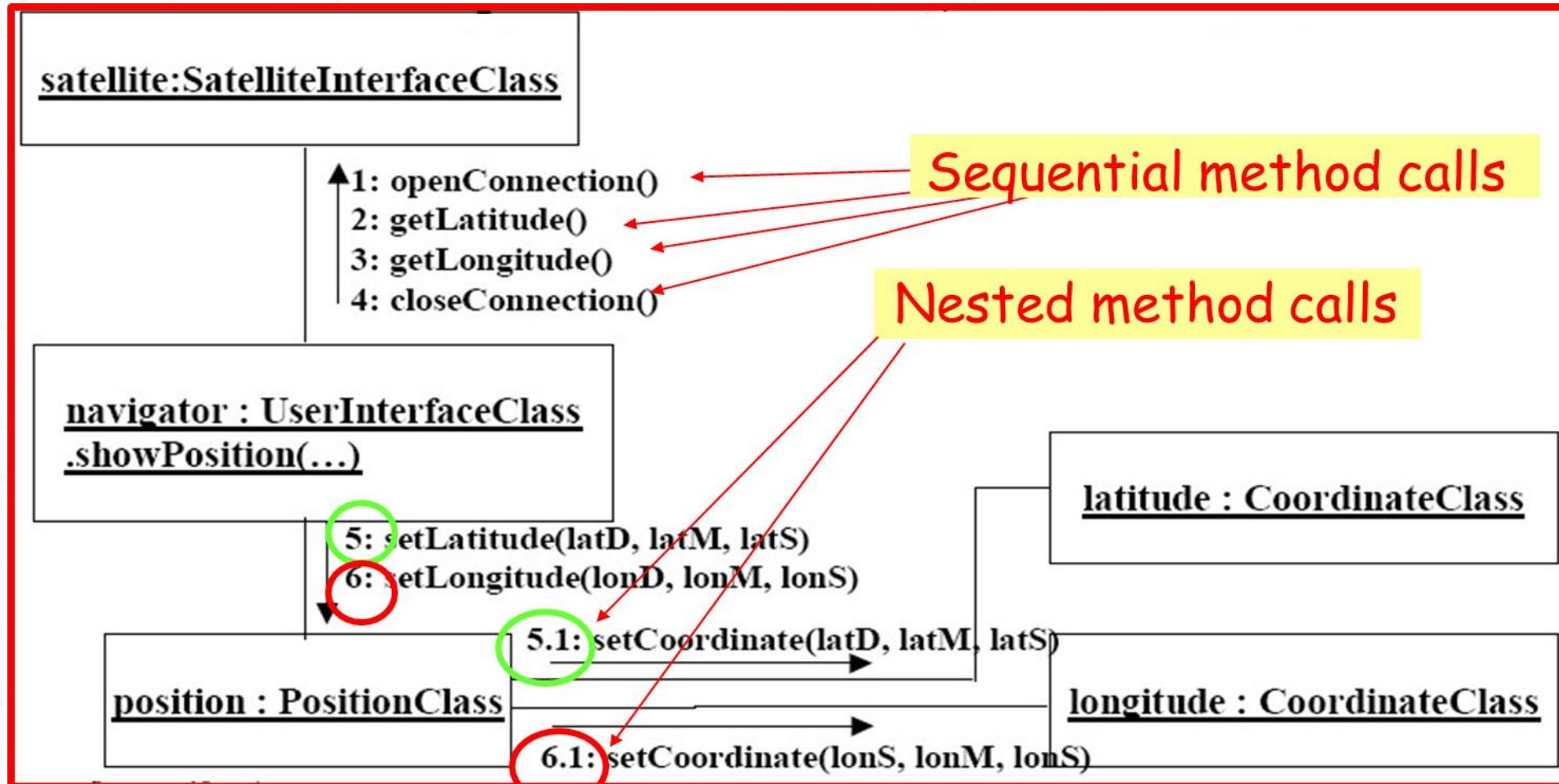
Message Sequence Numbers

- “Message Sequence number” is the integer represents the sequential order of the message.
- Each sequence term represents a level of procedural nesting.
- If message sequence numbers are at the same dot-level such as 1.1 and 1.2, those messages are considered to be sequential.
- If the model adds steps 1.1.1 and 1.1.2, then these new steps are understood to execute after step 1.1 and before step 1.2.
- In other words, they are nested beneath/within step 1.1.

Message Numbering – example 1

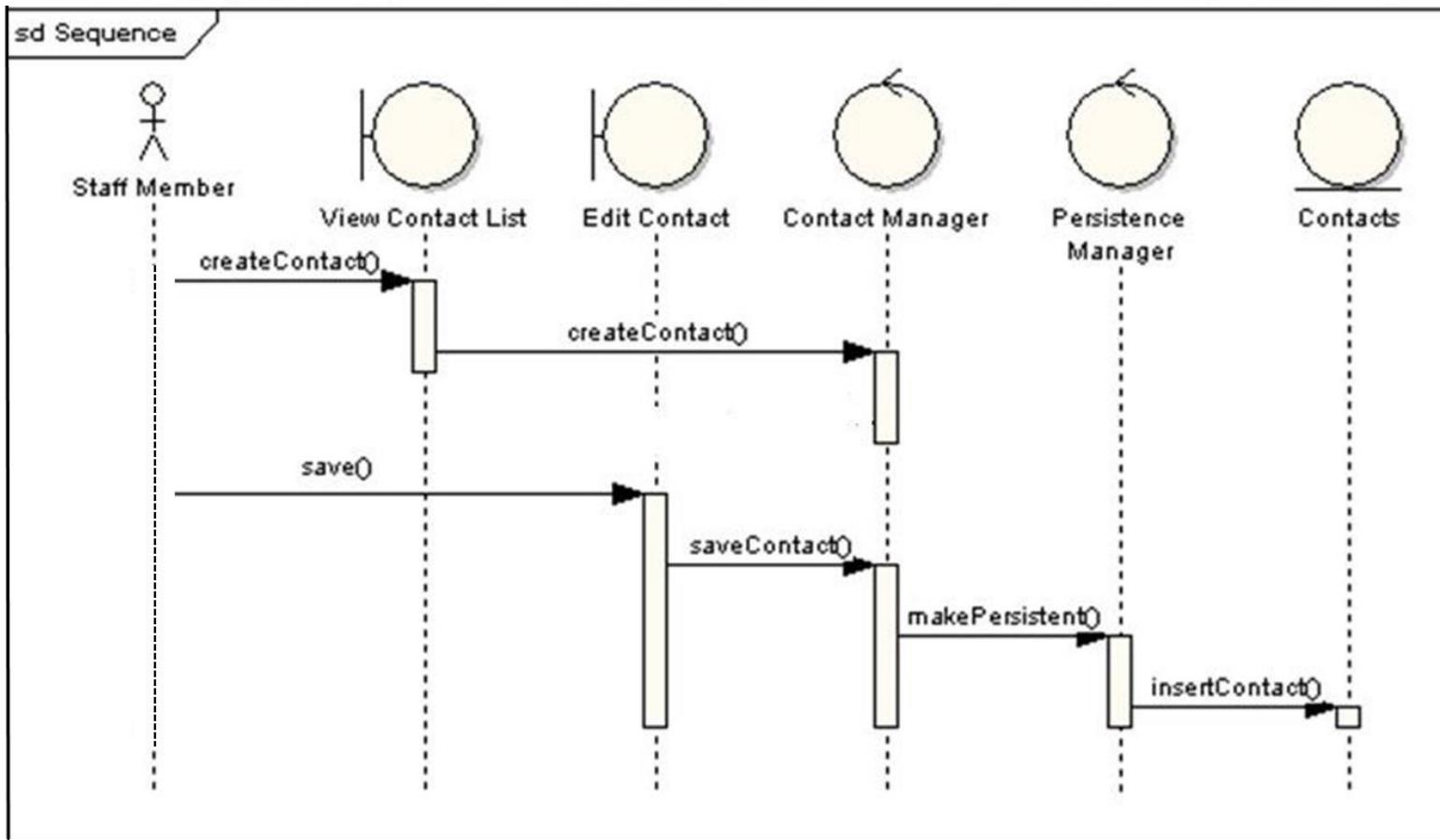


Message Numbering - example 2



Activity 1

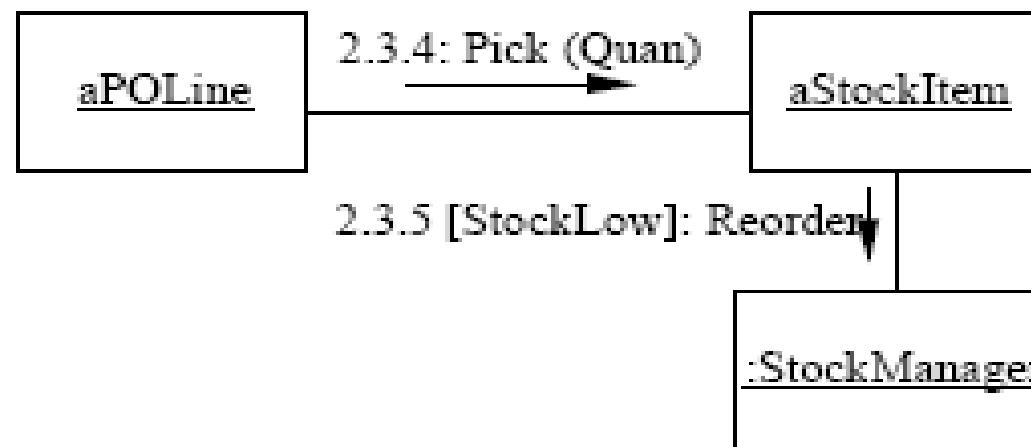
Convert following sequence diagram to a communication diagram



Guard Expressions

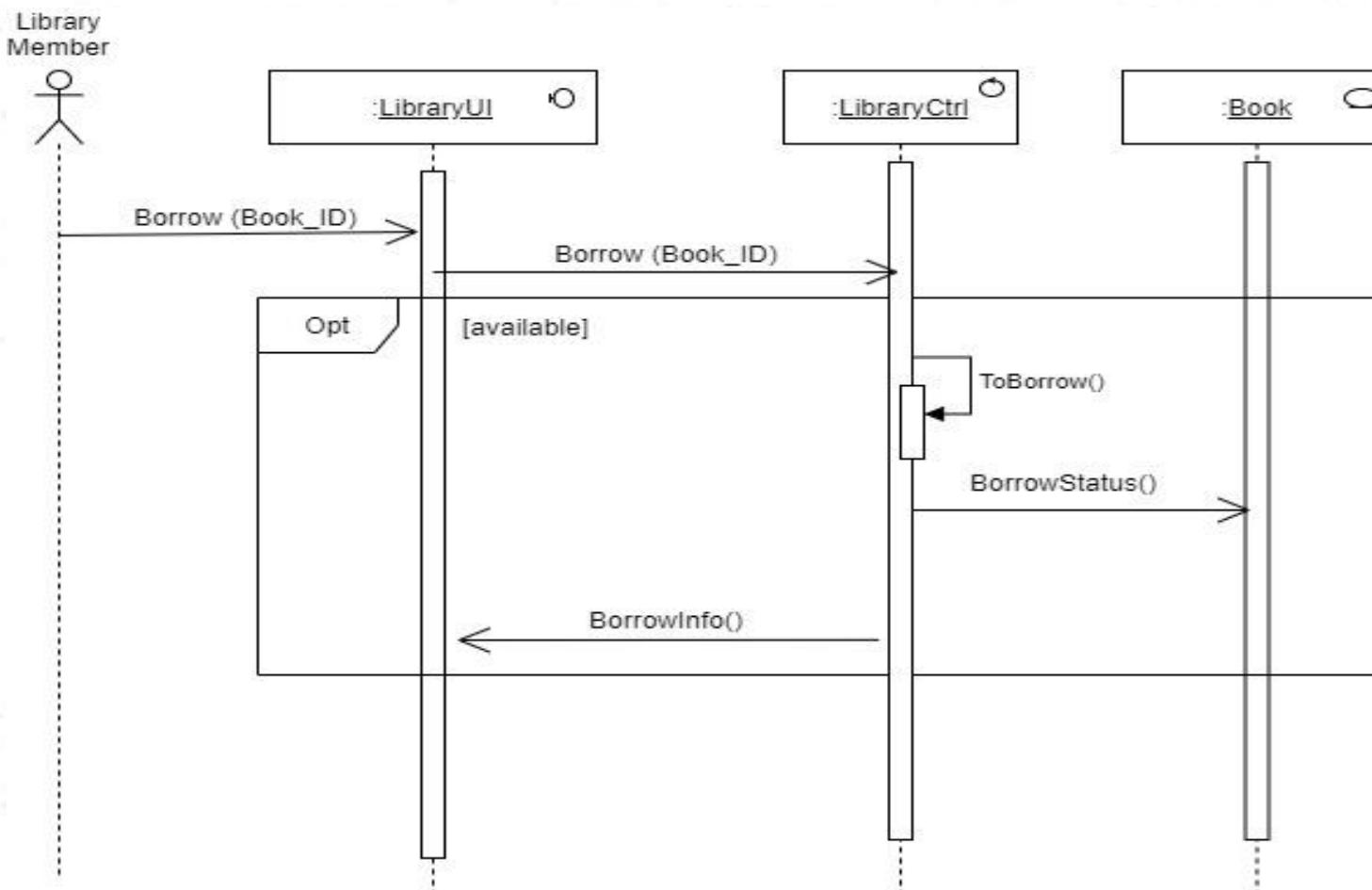
- Use to indicate messages which send under a certain condition.
- The message will be send only if the condition in the square bracket is true.

Syntax:- **message sequence number [condition] : Message**



Activity 2

Convert following sequence diagram into a communication diagram



Iteration and Looping

- A message may be executed repeatedly.
- The message repeats while the condition in the square brackets is true.

Syntax:

Message Sequence Number *[Condition] : Message signature

Message Sequence Number *[Condition][iterative clause] : Message signature

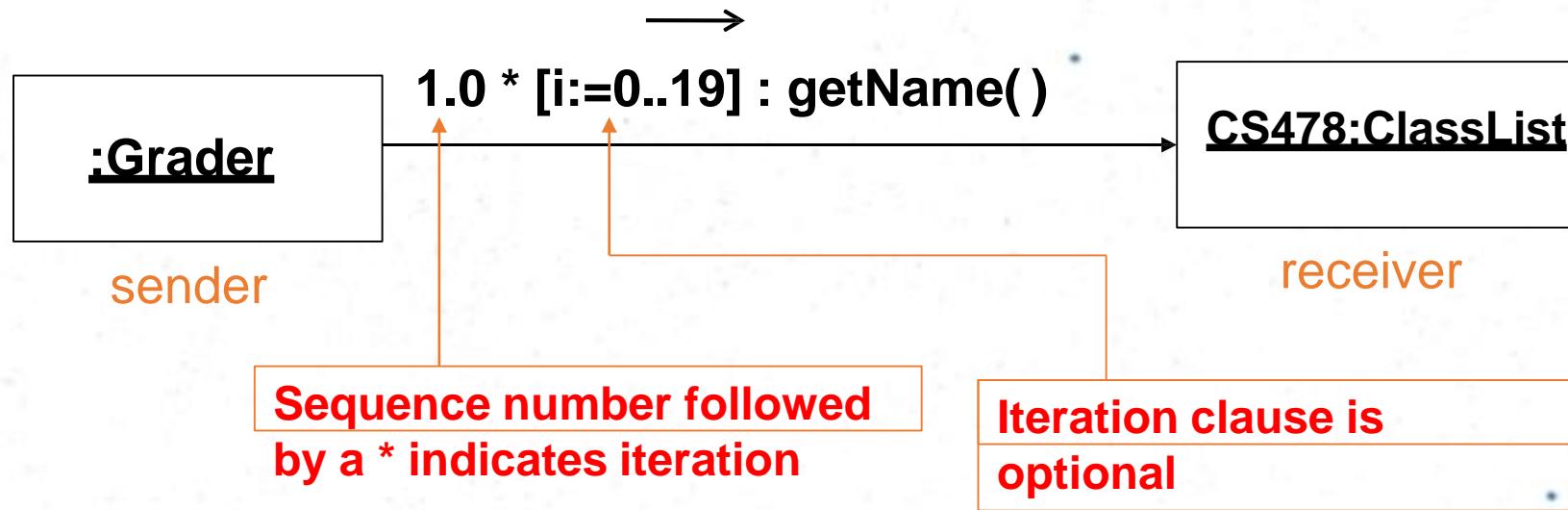
“The asterisk () indicates that the message is repeating”*

Example:

1.2 *[amount > 50,000] : Withdraw()

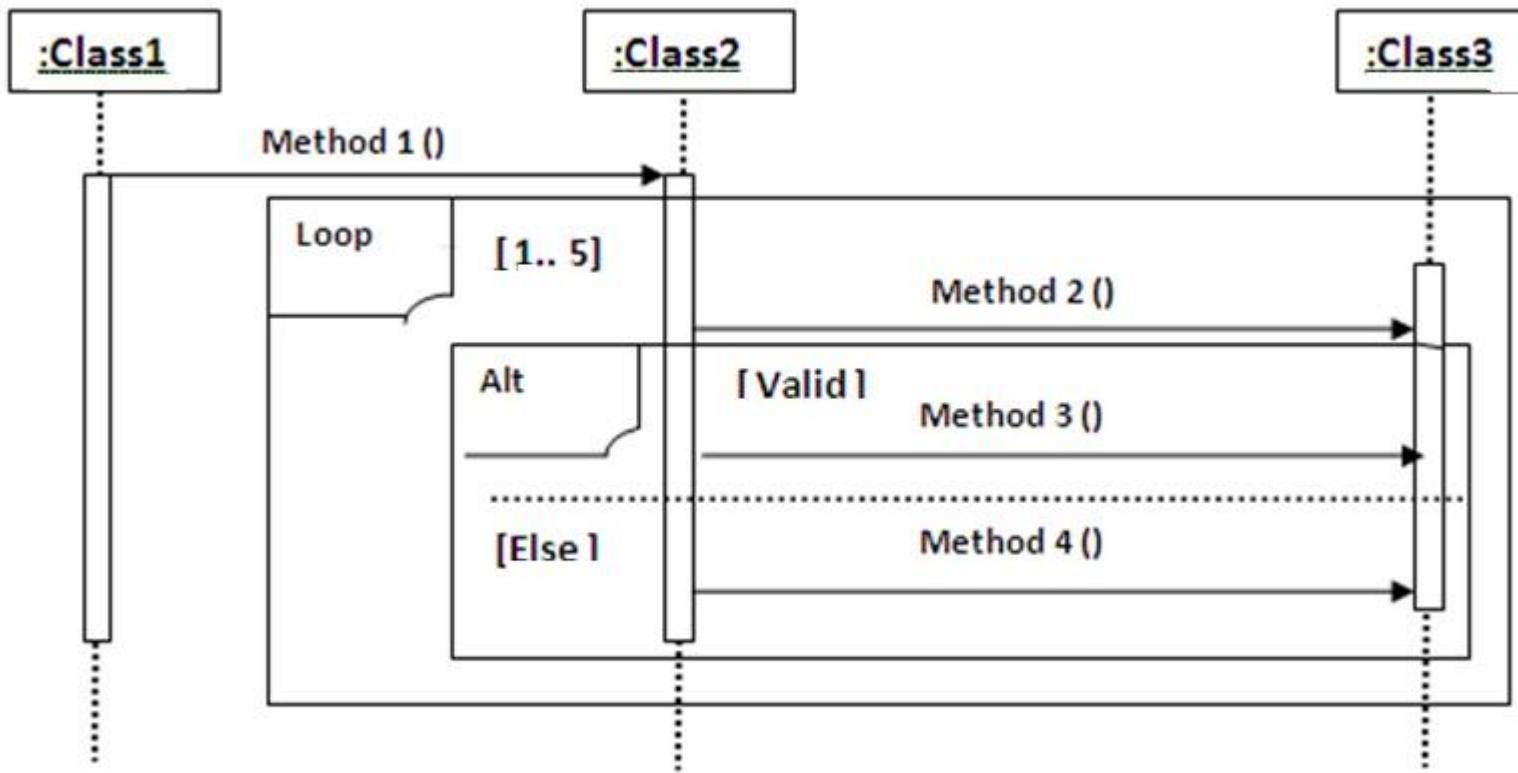
1.3 *[incorrect password] [i:=1..3] : Relogging()

Iteration and Looping - Example



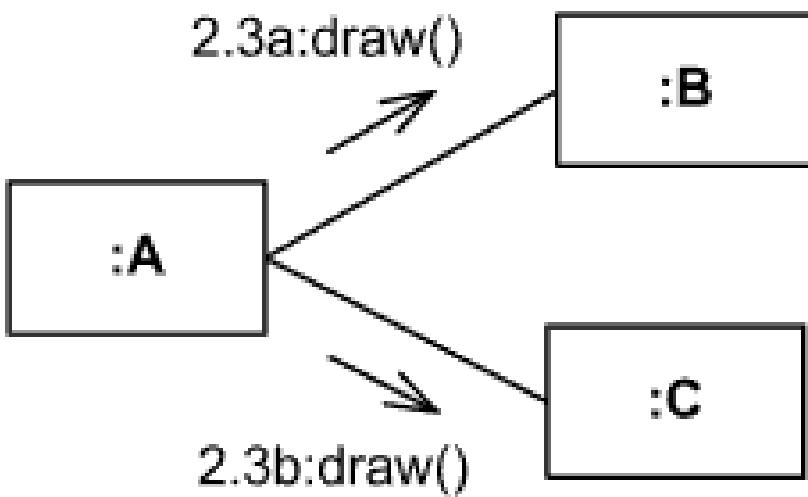
Activity 3

Convert following sequence diagram into a communication diagram



Parallel Activities

Indicate concurrent threads of execution in a UML communication diagram by having letters precede the sequence numbers on messages.

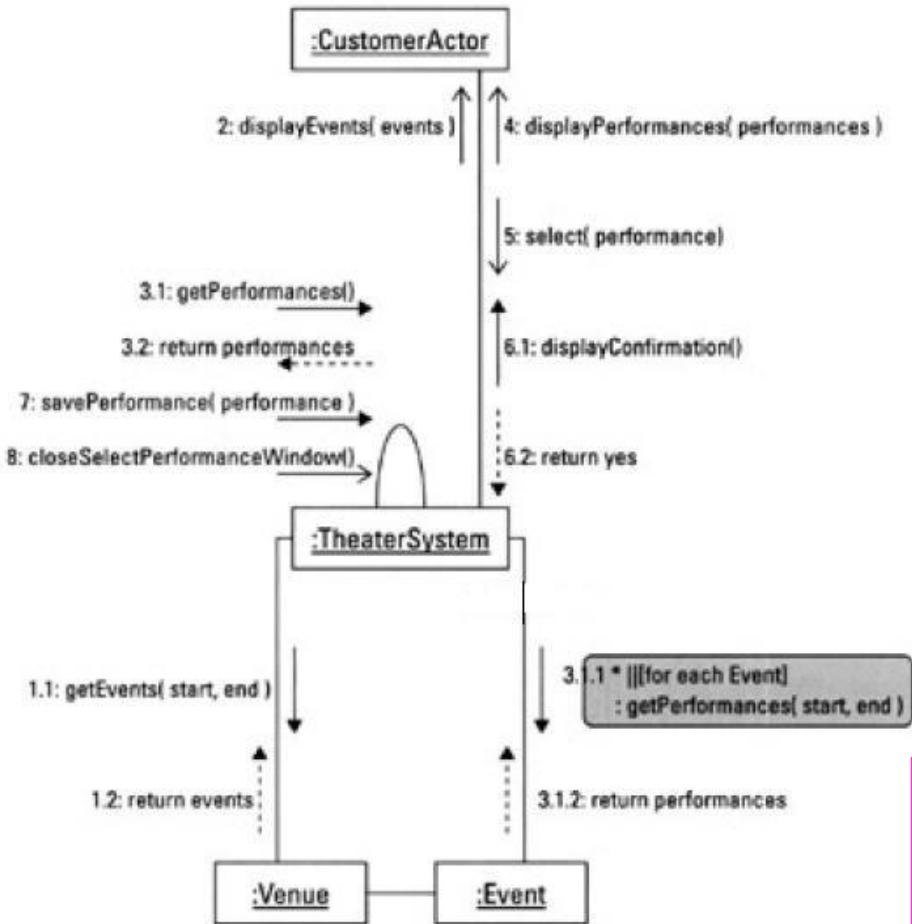


Instance of class A sends draw() messages concurrently to instance of class B and to instance of class C

Iteration and Parallel activities

- The iteration expression assumes that the messages in the iteration will be executed sequentially. But this is not always true.
- To model the fact that the messages may execute concurrently (in parallel), use a pair of vertical lines (||) after the iteration indicator (*).

Iteration and Parallel activities example



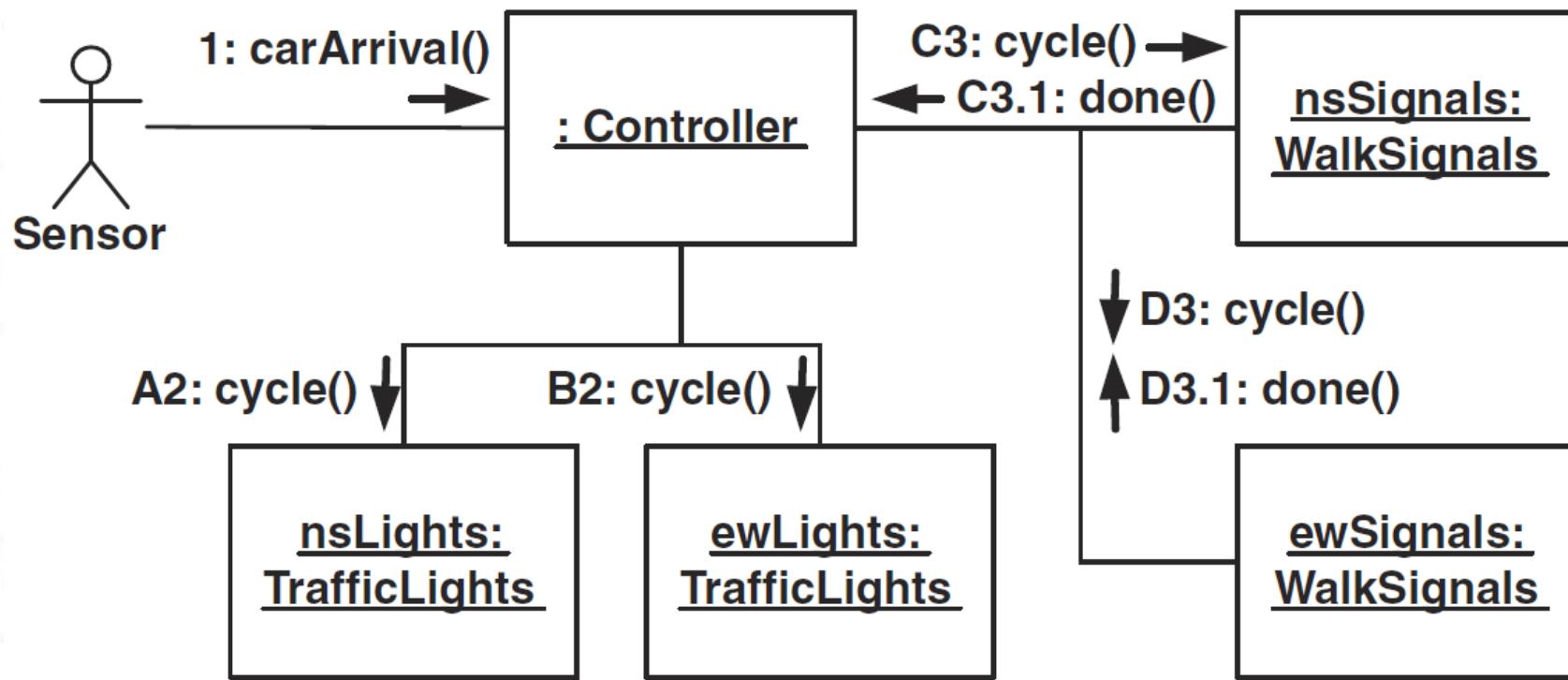
message 3.1.1
retrieves the performances for
each event, one at a time.

But we could change it to
retrieve the performances for
all events concurrently by
adding the concurrency
notation to the sequence term.

3.1.1 * | | [For each Event] :
getPerformances (start, end)

Activity 4

Find the concurrent activities in the following communication diagram.



Rules of Thumb

- **Avoid crossing links and crowded diagrams.**
- **Do not show all interactions** on an interaction diagram - only what is important for the scenario.
- **Do Not Model Obvious Return Values.**
- Model a return value **only when you need to refer to it elsewhere in a diagram.**

Sequence Diagram vs. Communication Diagram

- Sequence diagrams emphasize the **sequences of events** well.
- Communication diagrams show the **relationships between the classes** well.
- Keep both types of diagrams simple.

Strengths and Weaknesses

Type	Strengths	Weaknesses
Sequence	Show sequence or time order	Forced to extend to the right when adding new objects
Communication	Flexibility to add new objects in two dimensions. Better to illustrate complex branching, iteration and concurrent behavior	Difficult to see sequence of messages

Sequence and Communication Diagram Similarities

- Semantically equivalent.
- Can convert one diagram to the other without losing most of the information.
- Model the dynamic aspects of a system.
- Model the implementation of a use-case scenario.

References

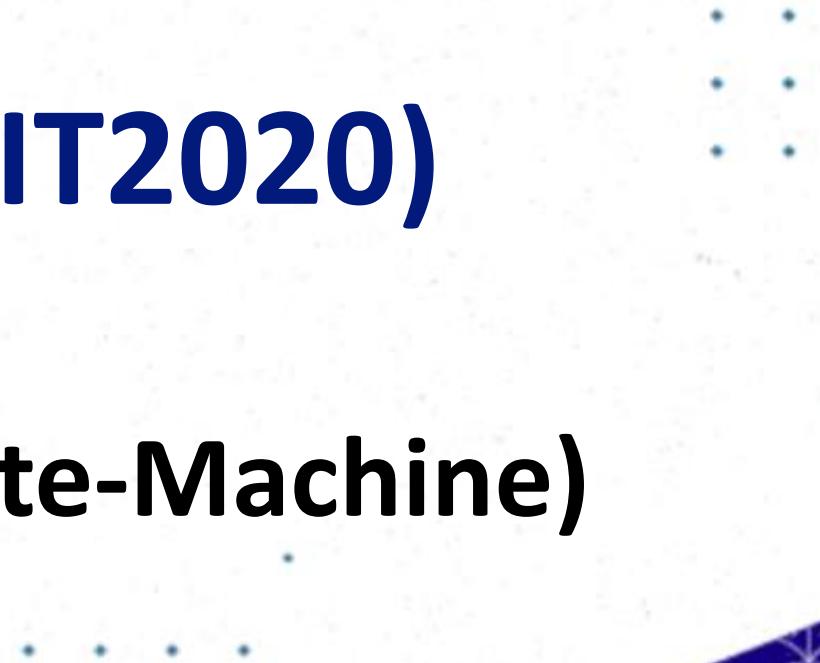
- UML 2 Bible
 - Chapters 8 & 9
- Applying UML and Patterns by Craig Larman
 - Chapter 15
- TheElements of UML 2 Style
 - Chapter 7

Thank you

Software Engineering (IT2020)

2022

Lecture 4 - State-Chart (State-Machine) Diagram



Session Outcomes

- Introduction to State Diagrams
- State Diagram symbols
 - States
 - Simple states
 - Composite states
 - Transitions
 - Call Event
 - Change Event
 - Time Event
 - Composite states in detail
 - Composite states with direct sub states
 - Composite states with regions

What is a State in General?

- **State** is a particular condition that someone or something is in at a specific time.

States of a Human Life



States of a Bulb



Determine states of these objects

- A Fan



- A Car



What are the States of an Object?

Objects has states ...

Active

Idle

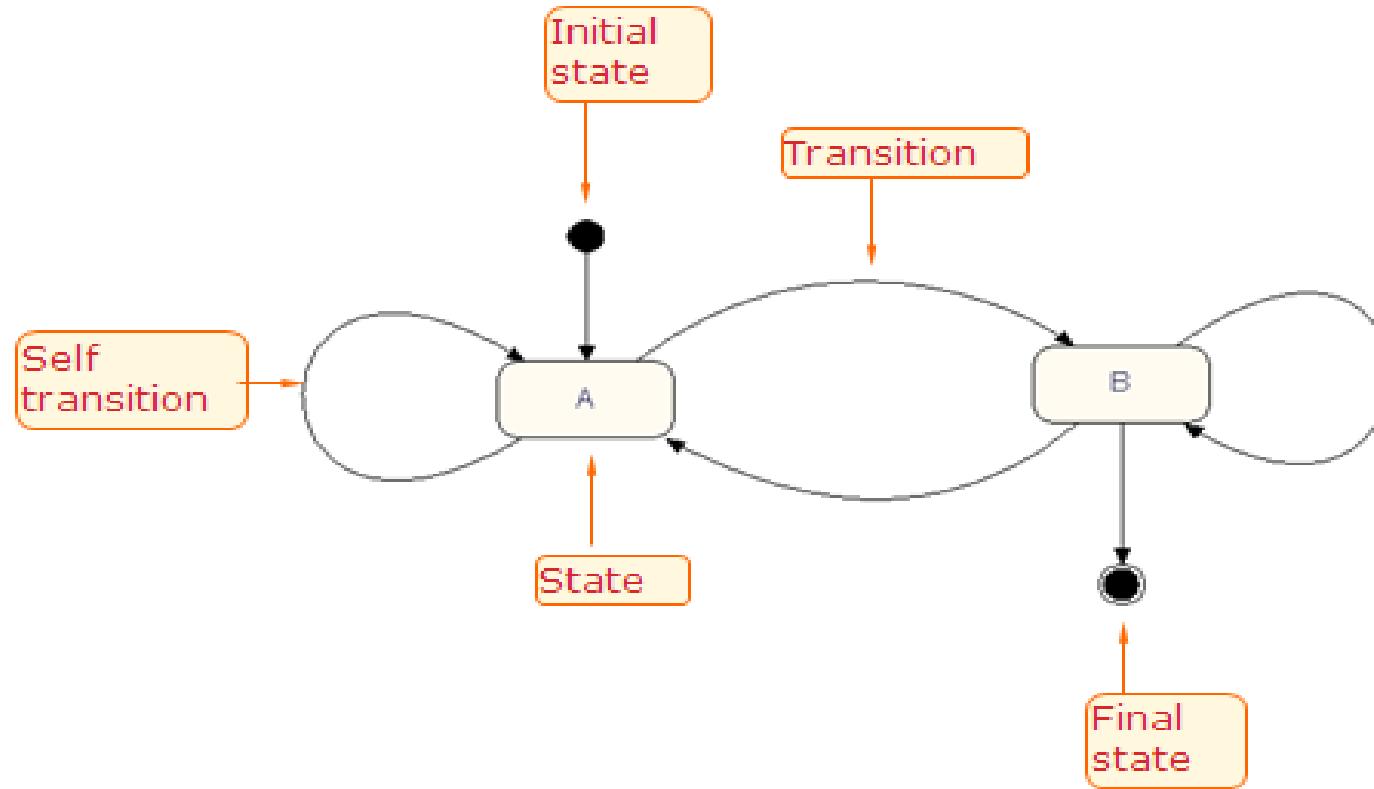
Waiting

UML State Machine Diagram

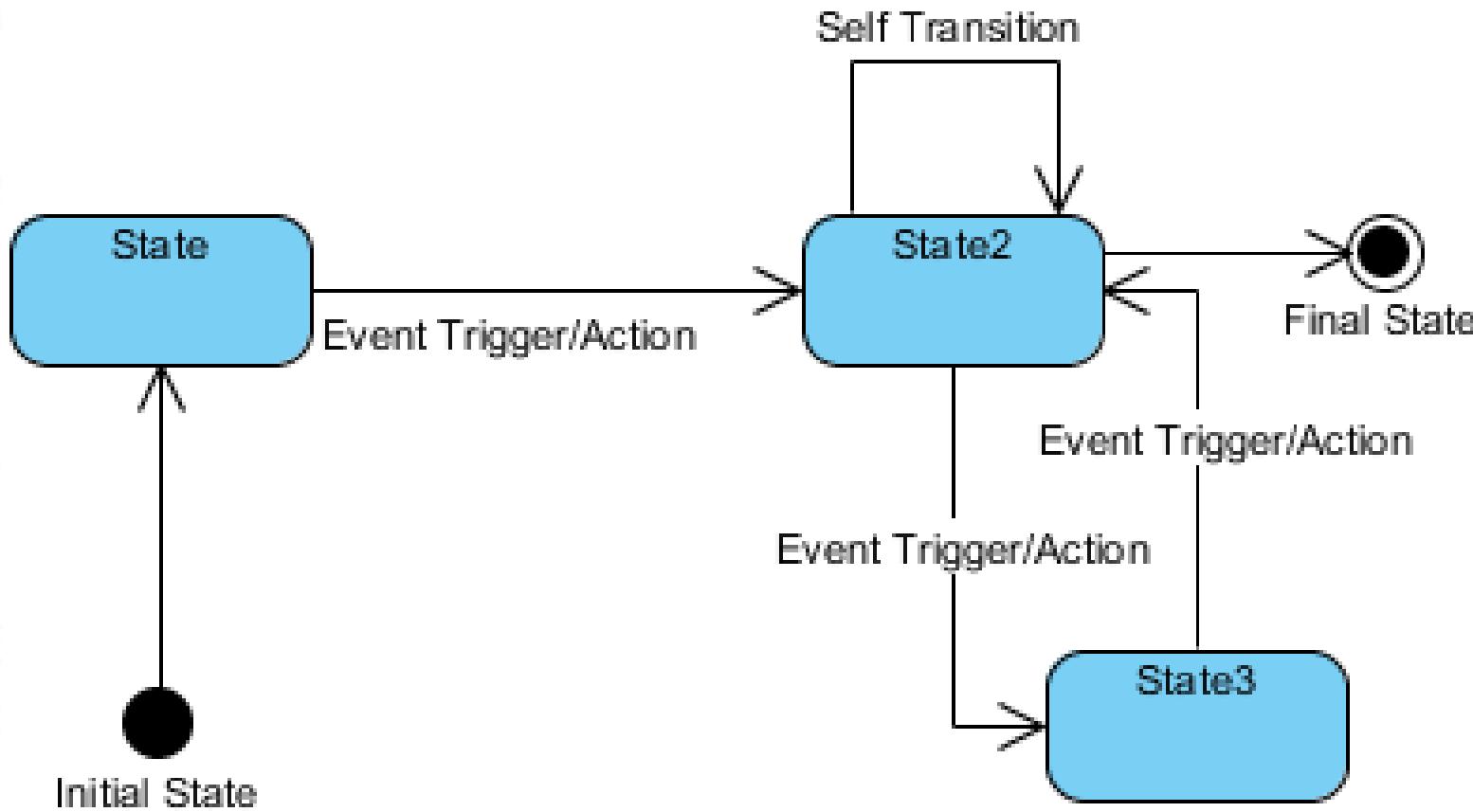
- A state machine diagram models the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime.
- There is only one state machine diagram for a class.
- A state diagram is typically drawn for only for the classes which contains significant dynamic behavior.

State-Machine Diagram Notations

- States
- Transitions



State Machine Diagram Example



Source: <https://www.visual-paradigm.com/tutorials/how-to-draw-state-machine-diagram-in-uml/>

What is a State?

- A state represents a **state that an object is in at a particular time**.
- Shown as a **rectangle with rounded corners**, with the **state name shown within the state**.
- While in the state, the object satisfies some condition, performs some action, or waits for some event.

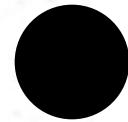
ON

OFF

Special States

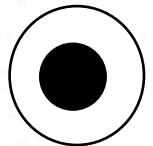
- Initial state (the object being constructed)

✓ denoted by a filled black circle



- Final state (the object being destructed)

✓ denoted by a circle with a dot inside

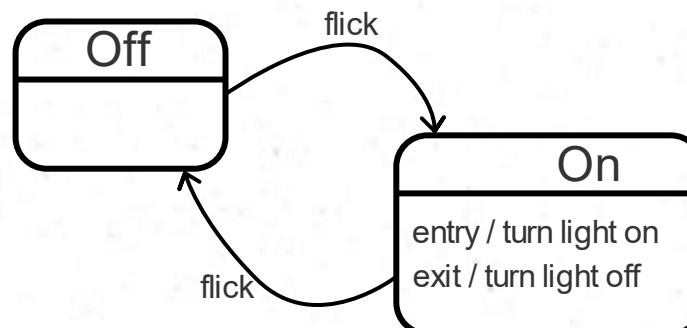


Types of States

- **Simple State** : Is a state that does not have sub states or compartments.

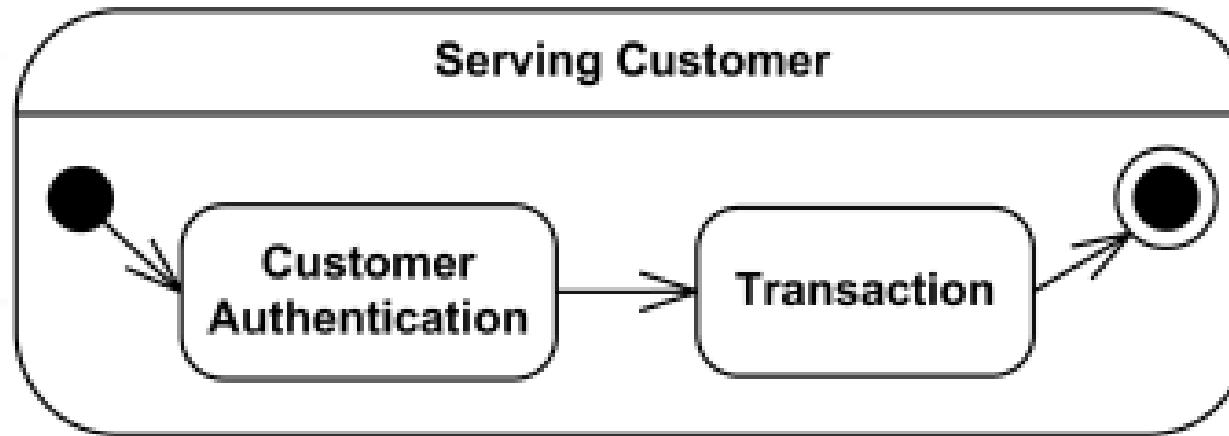


- **Simple State with compartments** :



Types of States cont...

- **Composite State** :Is defined as a state that has sub states (nested states).



Simple States

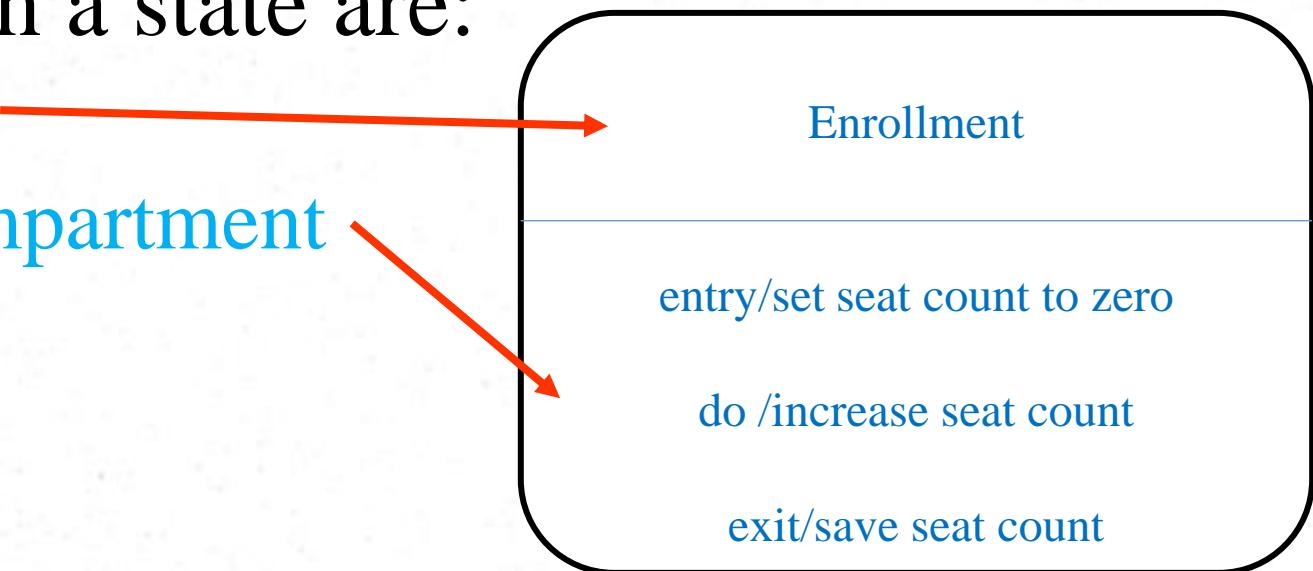
- Simple state is a state without compartments.
- Indicate using a simple rounded rectangle and state name inside.

Idle

Run

State with compartments

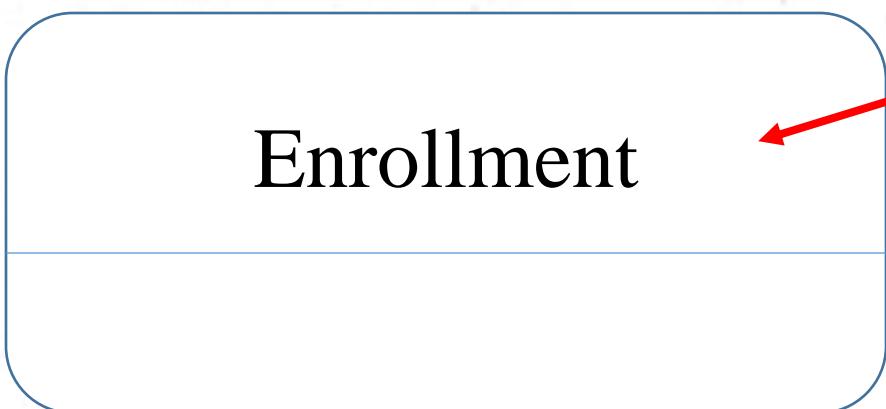
- A state may be subdivided into multiple compartments separated from each other by a horizontal line.
- Basic Compartments in a state are:
 - Name compartment
 - Internal behaviors compartment



Name Compartment

- This compartment holds the name of the state, as a string.

State with name compartment

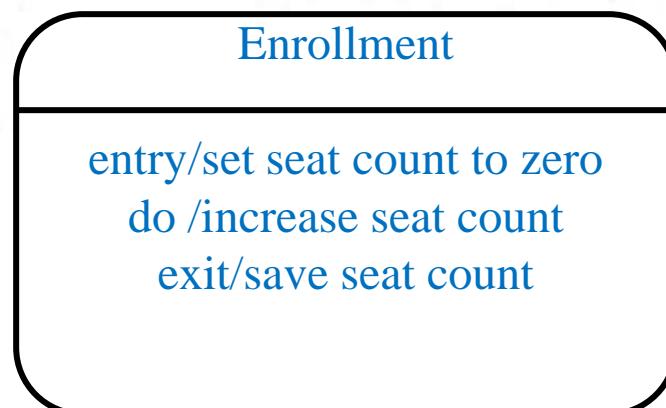


Internal Behaviors/Activities compartment

- This compartment holds a list of internal behaviors associated with a state.
- Each activity has the following format:
<<behavior-type-label>> / <<action>>
- The <behavior-type-label> identifies the situations under which will be invoked and can be one of the following:
 - **entry**
 - **exit**
 - **do**

Entry and Exit Activities/Behaviors

- **Entry** label identifies a Behavior, specified by the corresponding expression, which is performed upon entry to the State (entry Behavior).
- **Exit** label identifies a Behavior, specified by the corresponding expression, that is performed upon exit from the State (exit Behavior).



State Do behavior

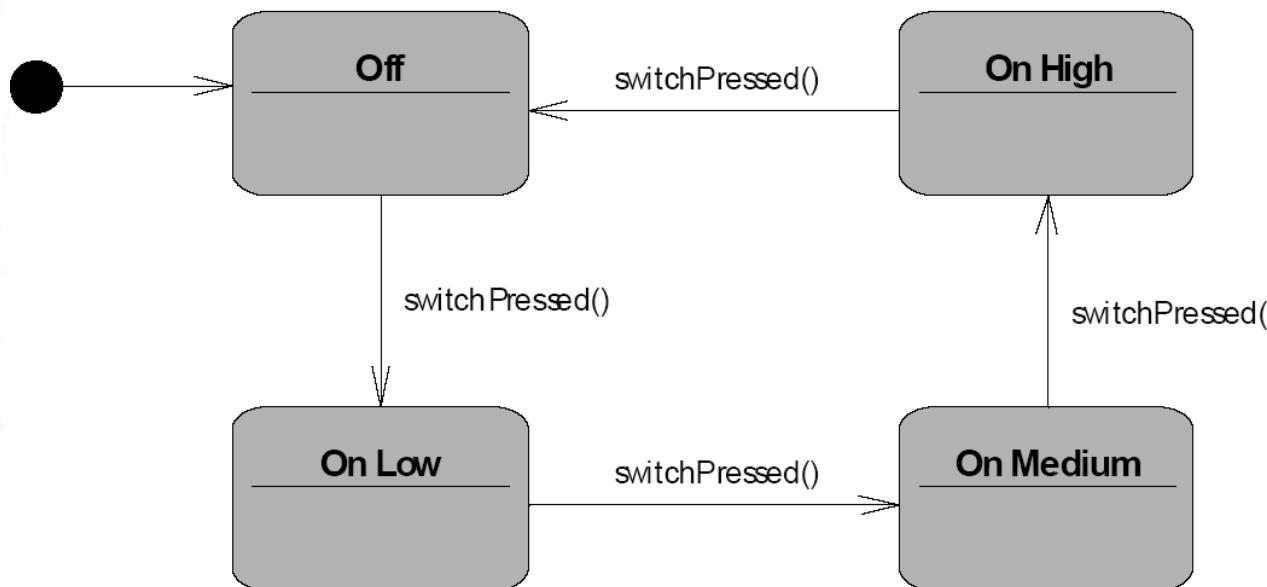
Withdraw

do / Read Account Balance

- Performed within a state. Ongoing work that an object performs while in a particular state. This is started after the entry action is finished. The work automatically terminates when the state is exited.
- May be interrupted.
- Activities (do), by definition, cannot change the state of the object (Unlike actions Entry and Exit), so interrupting them will not corrupt the state of the object.

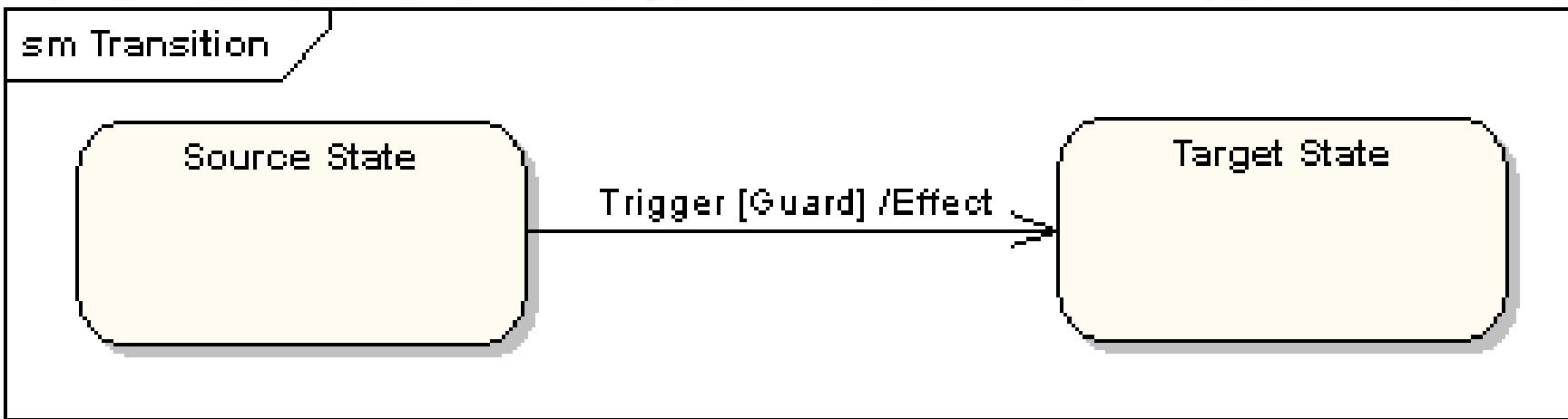
Activity 01

- A lamp has two bulbs (50w and 100w) and one switch and is controlled by a LampClass object. It has four states; Off, On Low, On Medium and On High. Following is the incomplete state diagram for LampClass object. Write internal behaviors for all four states.



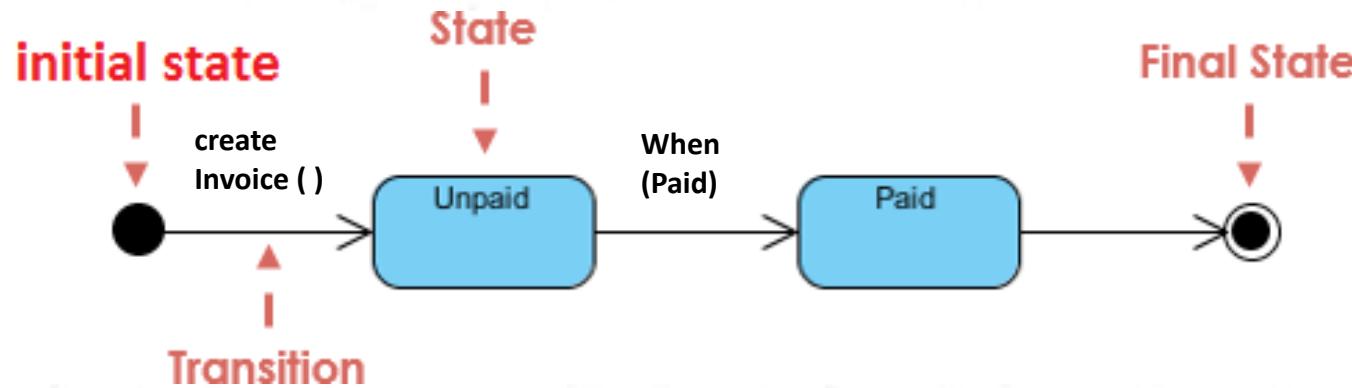
Transitions

- **Transitions** from one state to the next are denoted by **lines with arrow heads**.



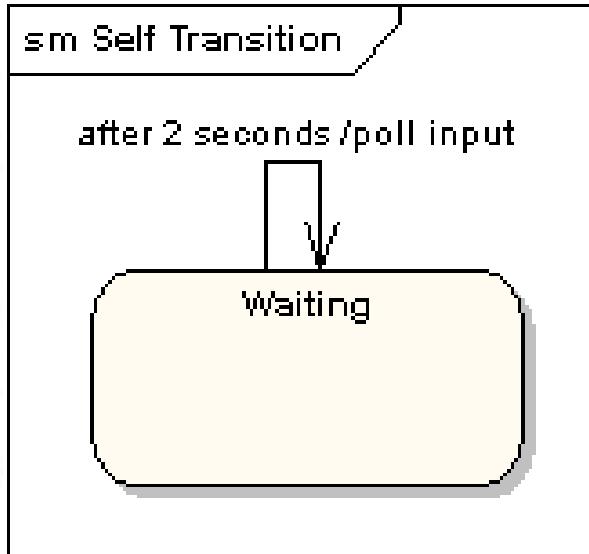
Elements of a Transition

- **Source State:** the state of the object before the transition.
- **Target State:** the state of the object after the transition.
- **Trigger:** the event that causes the transition



Self-Transitions

- A self transition is a transition that starts and ends in the same state.
- When a self transition is taken, the state in question is exited and entered again.
- Self transitions are commonly used to “restart” the current state, causing the exit actions to happen, followed by the entry actions.



Types of Triggers

- Mainly there are three types of triggers.
 - **Call Event** :Message (Parameters)
 - **Change Event** :When (Condition)
 - **Time Event** :After (Time Period)

Call Trigger / Event

- A Call-Event is denoted by the name of the triggering Operation.
- Used to represent a transition that occurs as a result of a message being received by the object.
- The arguments are optional.

Examples :

- buttonPressed().
- buttonPressed(buttonID).

Change Trigger / event – When (condition)

- Used to represent a transition that occurs when a condition / Boolean expression becomes true.
- Strictly, the condition should be a Boolean expression, but some Engineers use Plain English.
- Example : If the temperature T rises above 100° the object must change state:
 - when($T > 100^\circ$).
 - when (temperature exceeds 100 °).

Time Trigger / event – After (time duration)

- Time-Event is denoted with “after” followed by a Time Expression, such as “after 5 seconds.”
- The period can be expressed in any stated units.

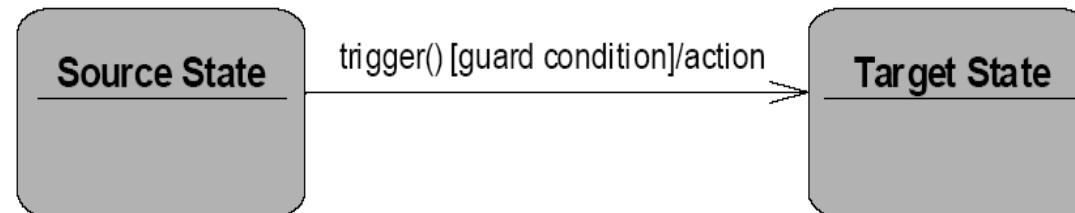
Example : After 60 seconds the object must change state:

 after(1 minute)

 after(60 seconds)

Triggers with Guard Conditions

All the triggers can have guard conditions and actions (action executes after the trigger). But these are **optional**.



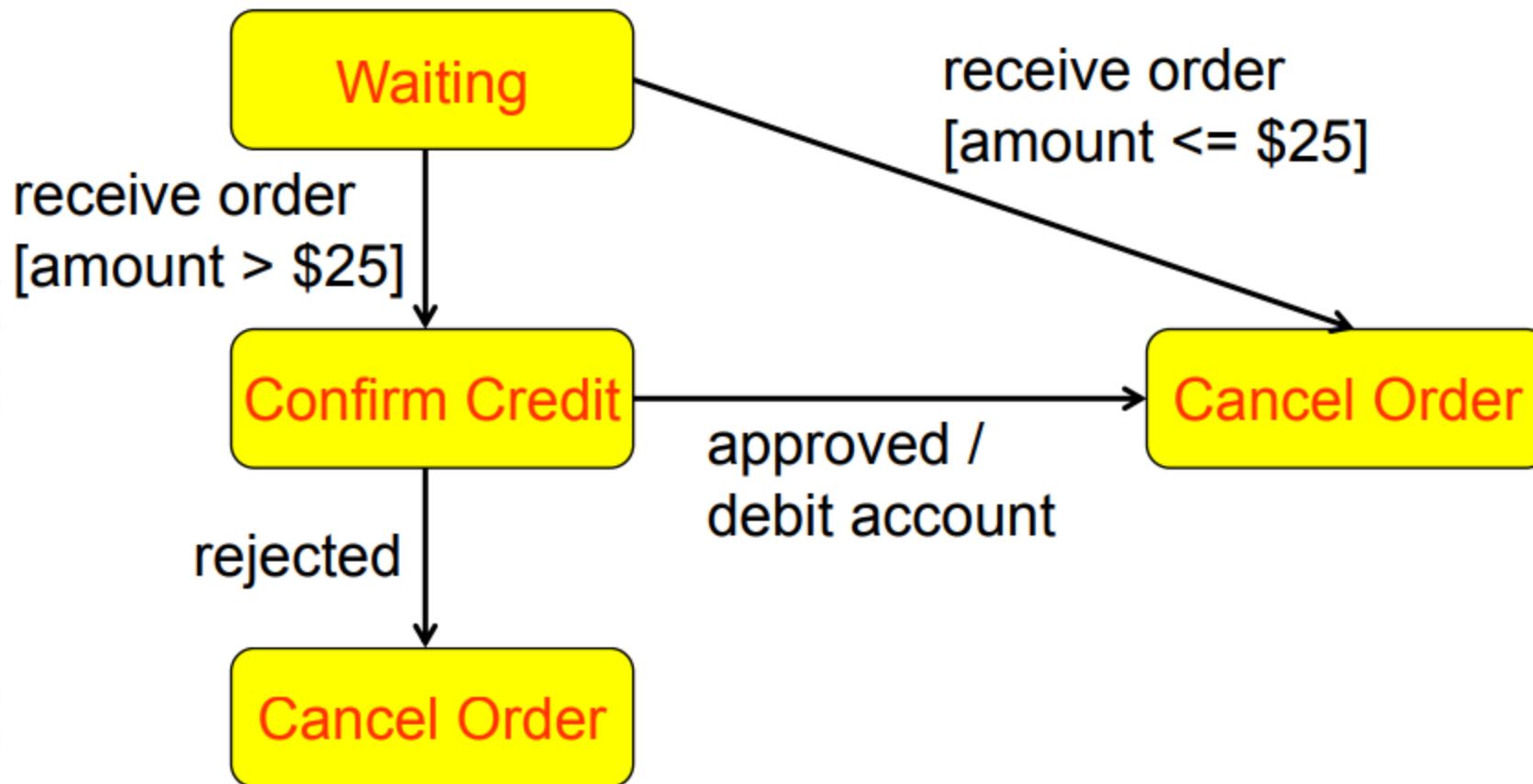
message() [Guard Condition] / Action

message(arguments) [Guard Condition] / Action

when(condition) [Guard Condition] / Action

after(timespan) [Guard Condition] / Action

Triggers with Guard Conditions - Example



Activity 02

Draw a state machine diagram for the Seminar class during registration.

- The Seminar is first proposed and then it is scheduled for an agreed date.
- Then the Seminar will be opened for enrollment for the students. When the enrolling starts, seat capacity is zero.
- As the number of seats are limited, as long as the Seminar is opened for enrollment, the remaining seats/size will be updated when each time a student is enrolled to the seminar.

Activity 03

Draw the State Diagram for the following Scenario.

- Burglar alarm is initially at the state of resting .
- Then by setting the alarm, burglar alarm state may be changed to the state set.
- When the alarm is set, it may be turned off. This will allow the alarm to be resting. While the alarm is set it can be triggered, which will make it ring. When the alarm is ringing, it can be turned off. Then the alarm will be resting again. Draw a state diagram for the Burglar Alarm class.

Note: No need to model the final state.

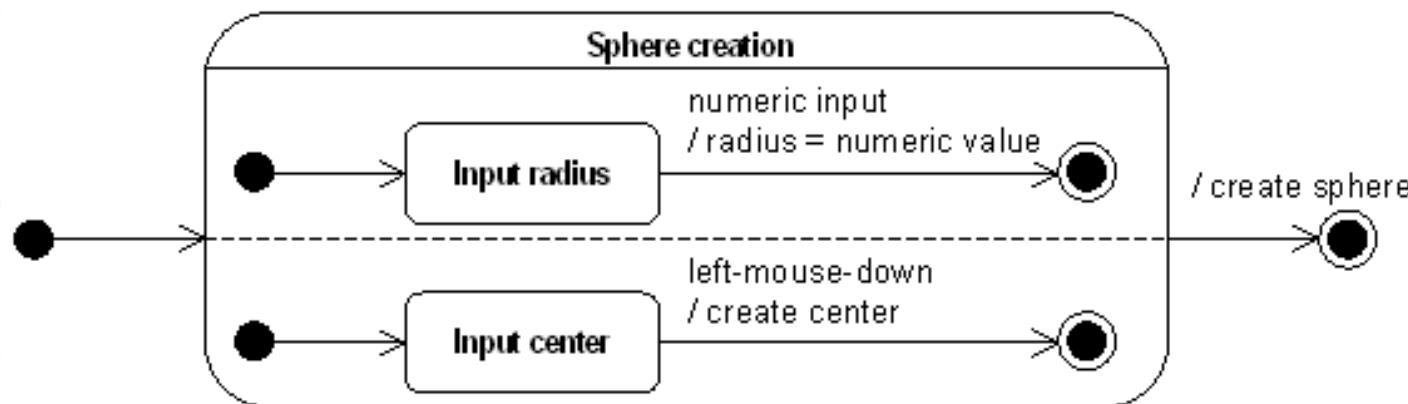
Activity 04

Draw the State Diagram for the following diagram

- You need to type a valid username and password to login to Courseweb. Once you are logged in you have access to unit contents.
- You can logoff once you have completed using the Courseweb. There is also a timeout of 5 minutes if you are inactive and you are automatically logged off. Draw a state diagram for the Courseweb user class.

Composite State

- A state that has sub states (nested states).
- Sub states could be sequential (disjoint) or concurrent.
- Sub states are in a separate compartment called “**Decomposition**” compartment.

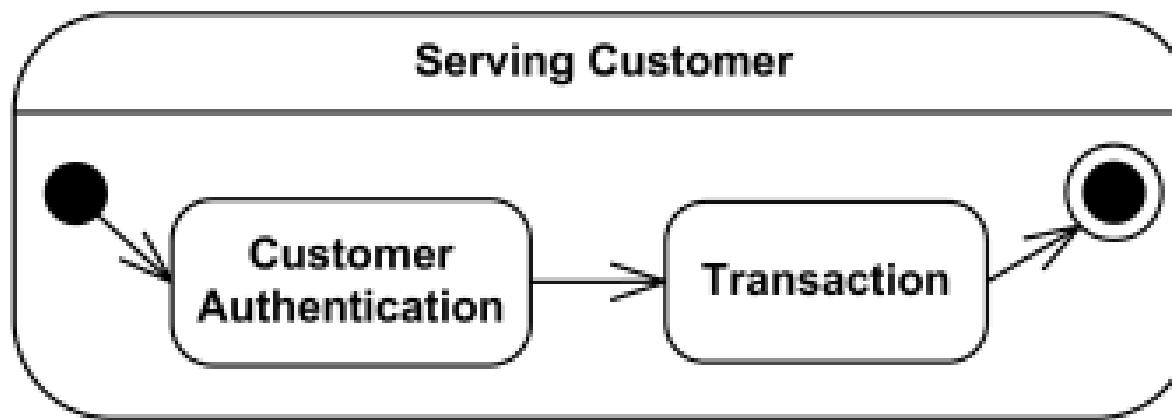


Composite State cont...

- Composite state can be simple composite state or Orthogonal composite state.

Simple Composite State

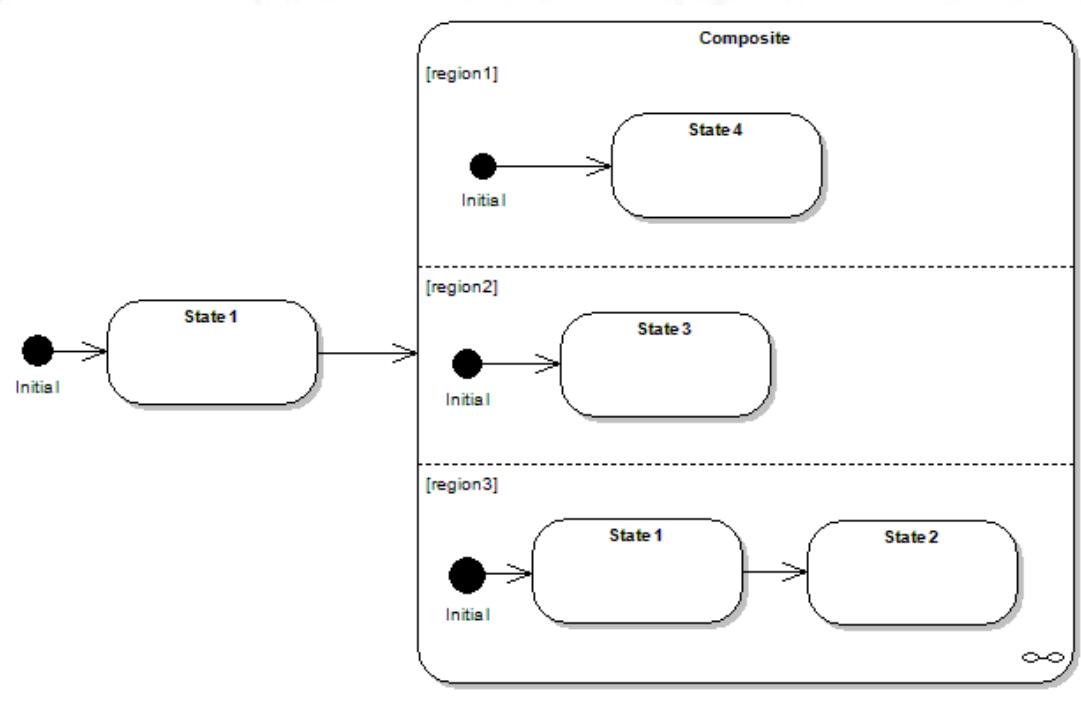
- Simple composite state contains just one region.



Composite State cont...

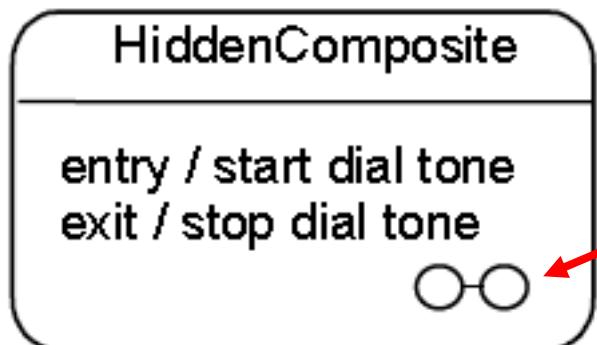
Orthogonal Composite State

- Orthogonal composite state has more than one region within the decomposition compartment.



Composite State – Hidden Decomposition Indicator

- In some cases, it is convenient to hide the decomposition of a composite state.
- For example, there may be a large number of states nested inside a composite state and they may simply not fit in the graphical space available for the diagram.
- In that case, the composite state may be represented by a simple state graphic with a special “composite” icon, usually in the lower right-hand corner.



Composite State with a
hidden decomposition indicator icon

Activity 05

Draw a State Chart for the given ATM

- ATM is initially turned off, it is in the **Off state**. After the power is turned on, ATM performs startup action and enters **Self Test** state. If the test unsuccessful, ATM goes into **Out of Service** state, otherwise goes to the **Idle** state. In this state ATM waits for customer interaction.
- The ATM state changes from **Idle** to **Serving Customer** when the customer inserts debit or credit card in the ATM's card reader. On entering the **Serving Customer** state, the entry action **readCard** is performed. The state also has exit action **ejectCard** which releases customer's card on leaving the state, no matter what caused the transition out of the state.

Activity 05 cont...

- **Serving Customer** state is a composite state with sequential sub states **Customer Authentication**, **Selecting Transaction** and **Transaction**. **Customer Authentication** and **Transaction** are composite states by themselves which is shown with hidden decomposition indicator icon. These states follow in order within the composite state.
- **Serving Customer** state has a **transition** back to the **Idle** state after transaction is completed . While serving customer if failure occurs it will move into the **Out of Service** state. ATM machine can be turned off when it is in the **Idle** state.
- While in the **Out of Service** state when a maintenance is performed it will move back to **Self Test** state
- A transition from **Serving Customer** state back to the **Idle** state could be triggered by **cancel** event as the customer could cancel transaction at any time.

State Charts VS Activity Diagrams

- A UML activity diagram can sometimes be used as an alternative to a state chart.
- **A UML state chart:**
 - shows all possible states and transitions (i.e. events) that can occur during the lifecycle of a single object;
 - all possible sequences of events can be traced via the transitions.
- **A UML activity diagram:**
 - can involve many different objects;
 - shows one possible sequence of transitions.

State Machine Diagram Guidelines

- **Keep the diagram simple**
 - If it is too complex, perhaps it should be broken down into separate diagrams.
- **Question “Black-Hole” States**
 - A black-hole state is one that has transitions into it but none out of it, something that should be true only of final states.
- **Question “Miracle” States**
 - A miracle state is one that has transitions out of it but none into it, something that should be true only of start points.

References

- UML 2 Bible
- Learning UML 2.0 by Kim Hamilton,
Russ Miles
- Chapter 15UML 2 Bible

End of the Lecture

Software Engineering (IT2020) 2022

Lecture 5 - Physical Diagram

UML Diagram Classification

Static (Structural Diagrams)

- Class diagrams
- Object Diagram

Dynamic (Behavioral Diagrams)

- State diagram
- Activity diagram
- Sequence diagram
- Collaboration diagram

Implementation (kind of a Structural diagrams)

- Physical Diagram
 - Component diagram
 - Deployment diagram

Implementation Diagram describes the elements required to implement a system

UML Physical Diagram

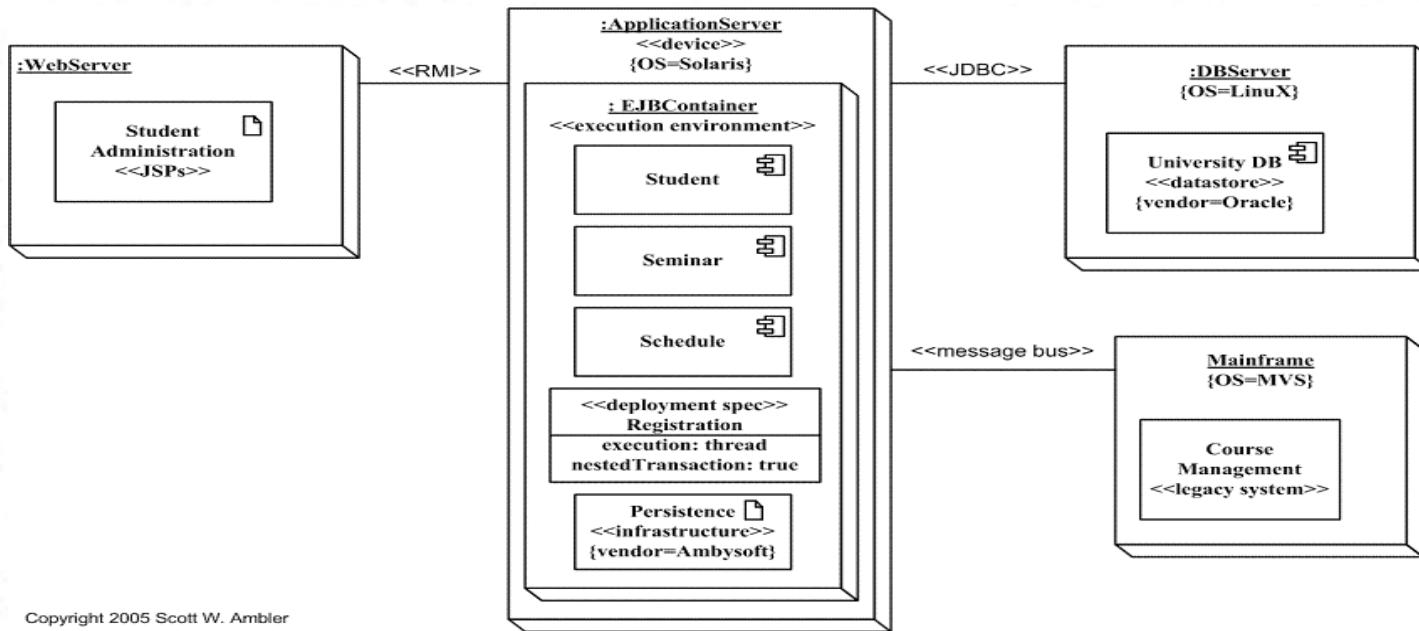
- Physical diagrams are used when the development of the system is complete.
- This use to give descriptions of the physical information about a system.
- Physical diagrams consists of two main diagrams.
They are :
 - Component diagram
 - Deployment diagram

UML Physical Diagram

- **Component diagrams** show the software components of a system and how they are related to each other.
- **Deployment diagrams** show the physical relationship between hardware and software in a system.

Physical = Component + Deployment
Diagrams Diagram Diagram

UML Physical Diagram Example



Copyright 2005 Scott W. Ambler

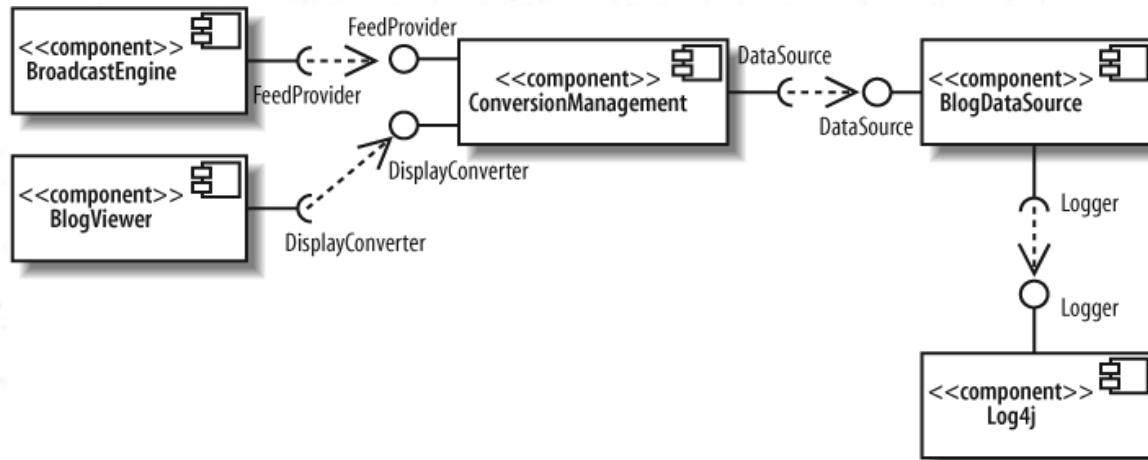
Component Diagram

Component Diagram

- Components are used to organize a system into manageable, reusable, and swappable pieces of software.
- Component Diagram Consists of:
 - Components
 - Interfaces
 - Relationships in between interfaces

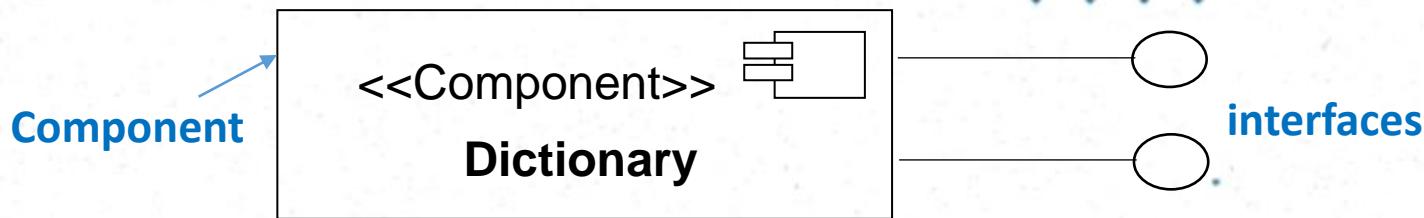
Component Diagram

- **Component diagram** shows components, provided and required interfaces and relationships between them.



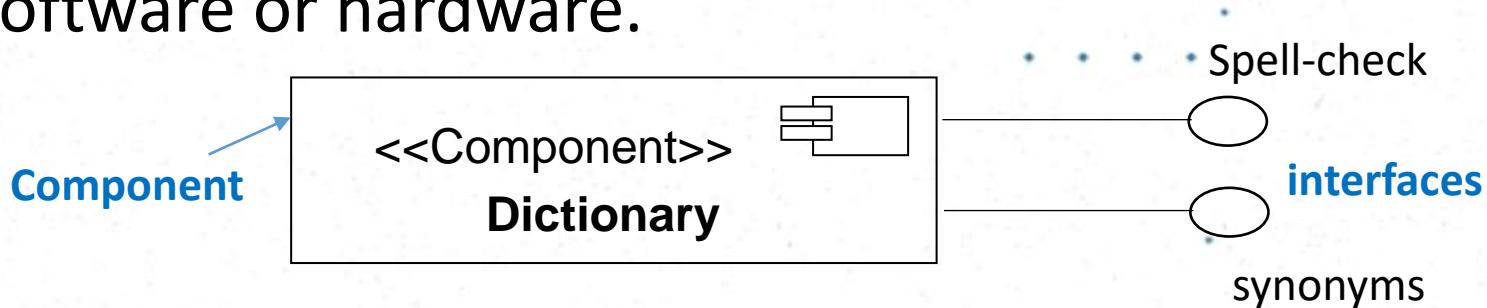
Components

- 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.
- Components can be represented in different ways.
- Most common way is as **Rectangle**.
- A component is drawn as a rectangle with <<component>> stereotype and a tabbed rectangle icon in the upper right-hand corner.



Component Interfaces

- Component should be loosely coupled, so that they can be changed without changes on the other parts of the system.
- Interfaces are used for this purpose.
- An interface is a list of operations supported by a piece of software or hardware.



Component Interfaces

- Interfaces control dependencies between components and make components swappable.
- Each component realizes (supports) some interfaces and uses other components.
- Accordingly, there are two types of interfaces
 - Provided Interfaces
 - Required Interfaces



Interfaces

Provided Interfaces

- A *provided interface* of a component is an interface that the component realizes.
- Other components and classes interact with a component through its provided interfaces .
- A component's provided interface describes the services provided by the component.



Interfaces

Required Interfaces

- A *required interface* of a component is an interface that the component needs to function.
- More precisely, the component needs another class or component that realizes that interface to function.



Interface Representations

The most common notation for interfaces is known as **Ball-and-socket notation**



Activity 1:

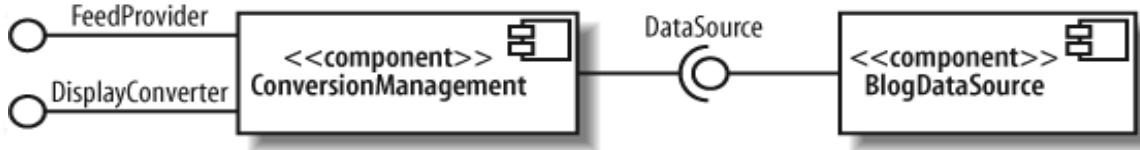
Order is a component which provides three interfaces for the other components to access. They are addLineItem, cancel and pay. It also uses applyDiscount interface to connect with other component.

Connecting Components

- There are two ways to connect components.
 - Using Dependency relationship (Dependency arrow)



- Assembly Connector notation
Snapping the ball and socket together (omitting the dependency arrow)

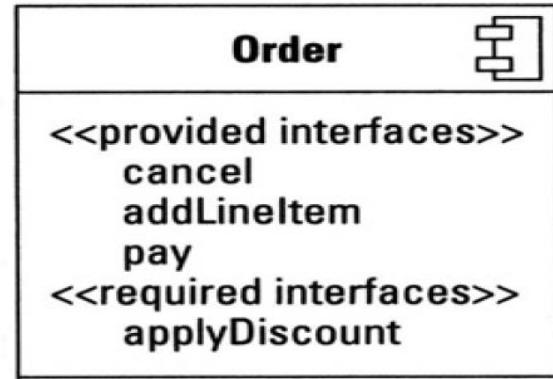
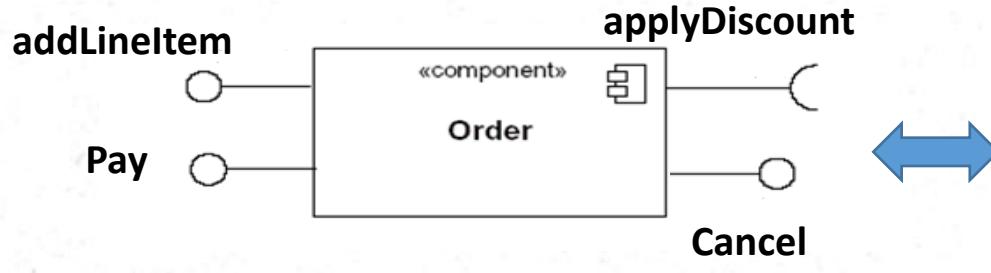


Additional Notations to represent Components and Interfaces

Components as Class Representations

White-box view

- The most compact way of showing required and provided interfaces is to list them inside the component.
- Provided and required interfaces are listed separately.



Activity 2

- Draw a component using white box notation for the given scenario.
- ConversionManagement component provides two interfaces; FeedProvider and Displayconverter. It also uses DataSource interface to access other components.

Additional Interface Representations

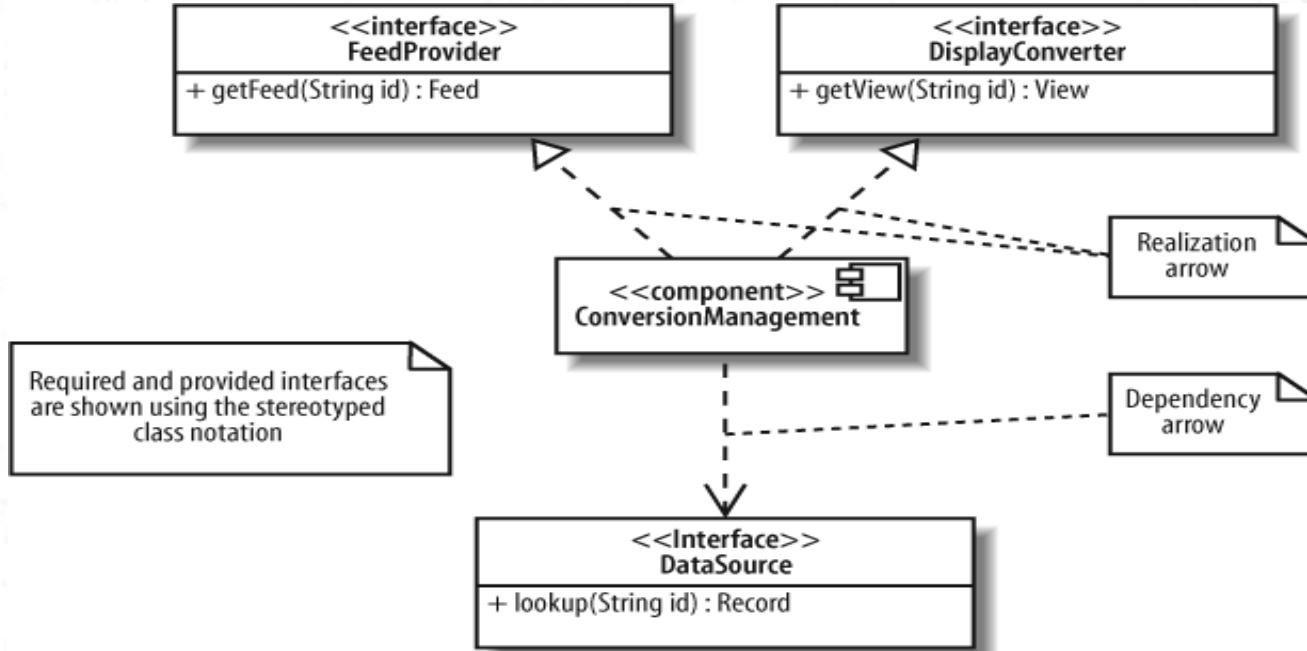
Stereotype Notation

Component's required and provided interfaces are shown by drawing the interfaces with the stereotyped class notation.

- If a component realizes an interface, draw a realization arrow from the component to the interface.
- If a component requires an interface, draw a dependency arrow from the component to the interface

Additional Interface Representations

Stereotype Notation (cont..)



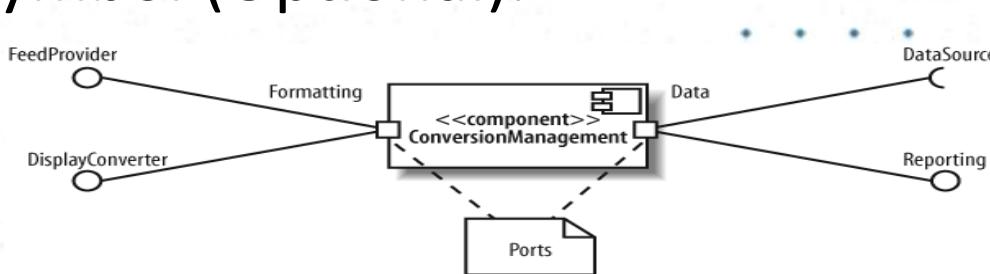
Activity 3

Draw a component diagram for the following.

- Order is a component which provides three interfaces for the other components to access.
- They are addLineItem, cancel and pay.
- Order component also needs to access itemCode, customerDetails, accountDetails and applyDiscount interfaces realized by Product, Customer, Account and Discount components respectively.

Ports

- Specifies a distinct interaction point
 - Between the component and its environment
 - Between the component and its internal parts
- Is shown as a small square symbol.
- Ports can be named, and the name is placed near the square symbol (Optional).

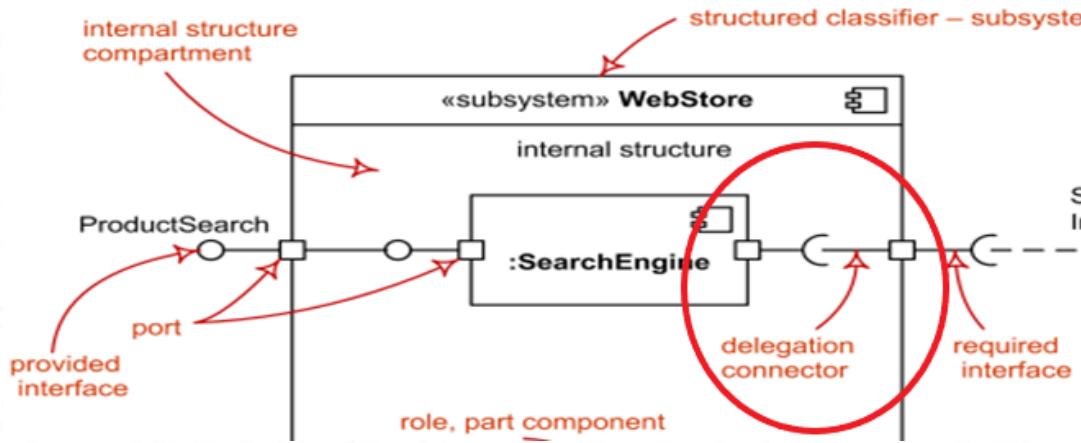


Delegation & Assembly Connectors

- Components have their own unique constructs when showing ports and internal structure called delegation connectors and assembly connectors.
- These are used to show how a component's interfaces match up with its internal parts and how the internal parts work together.

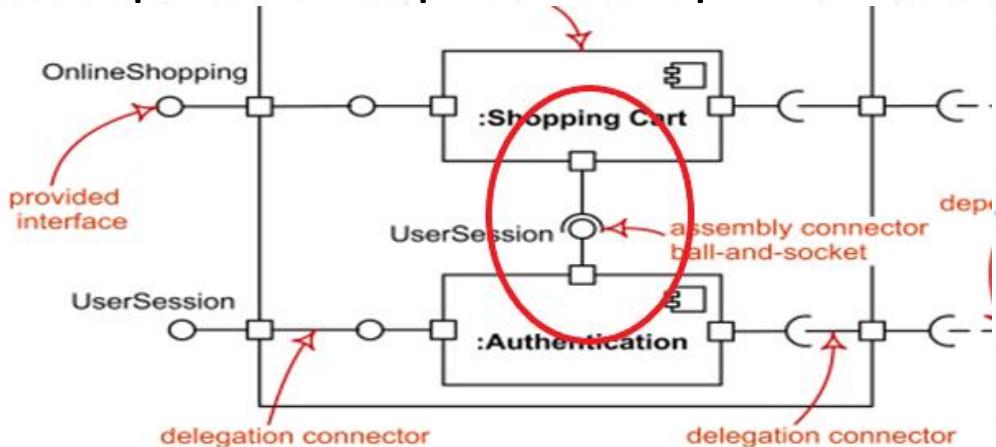
Delegation Connectors

- *Delegation connectors* are used to show that internal parts of the component realize or use the component's interfaces.

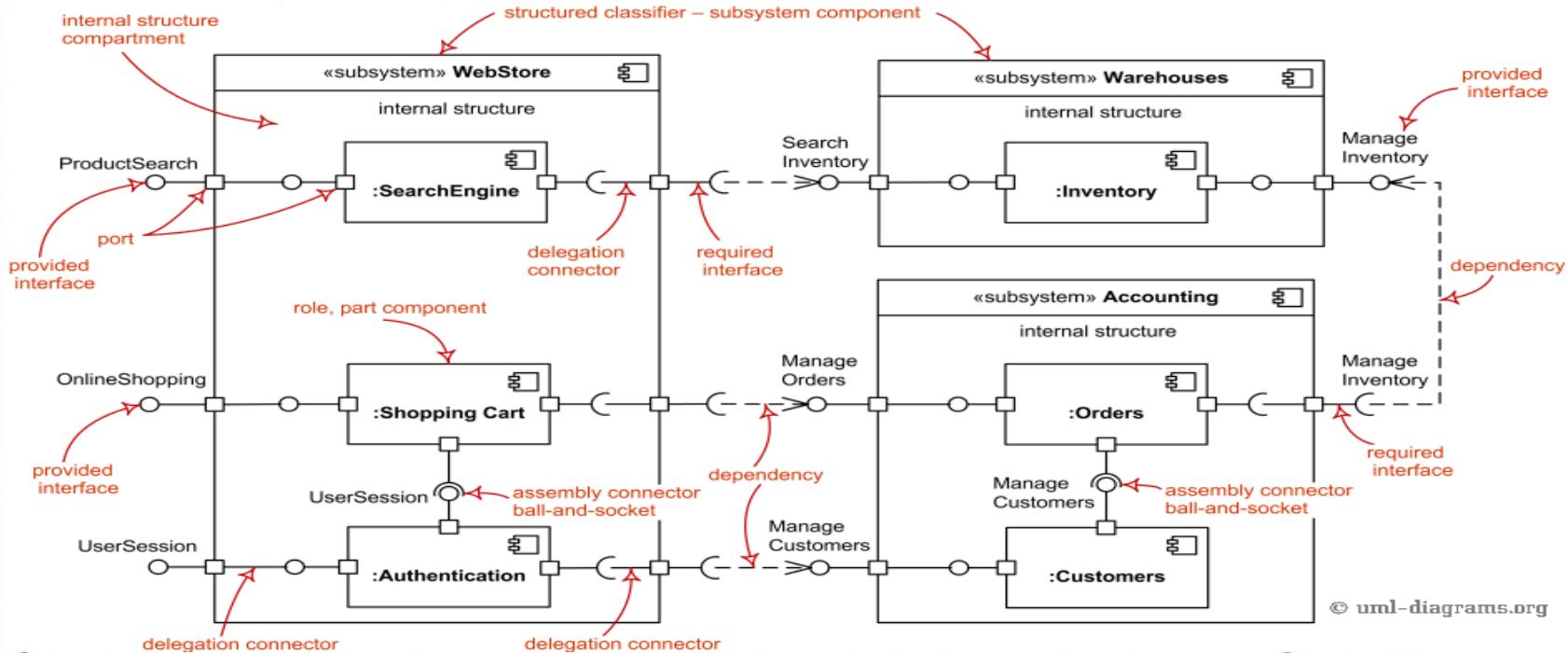


Assembly Connectors

- Assembly connectors are special kinds of connectors that are defined for use when showing composite structure of components.
- Assembly connectors snap together the ball and socket symbols that represent required and provided interfaces.



Example



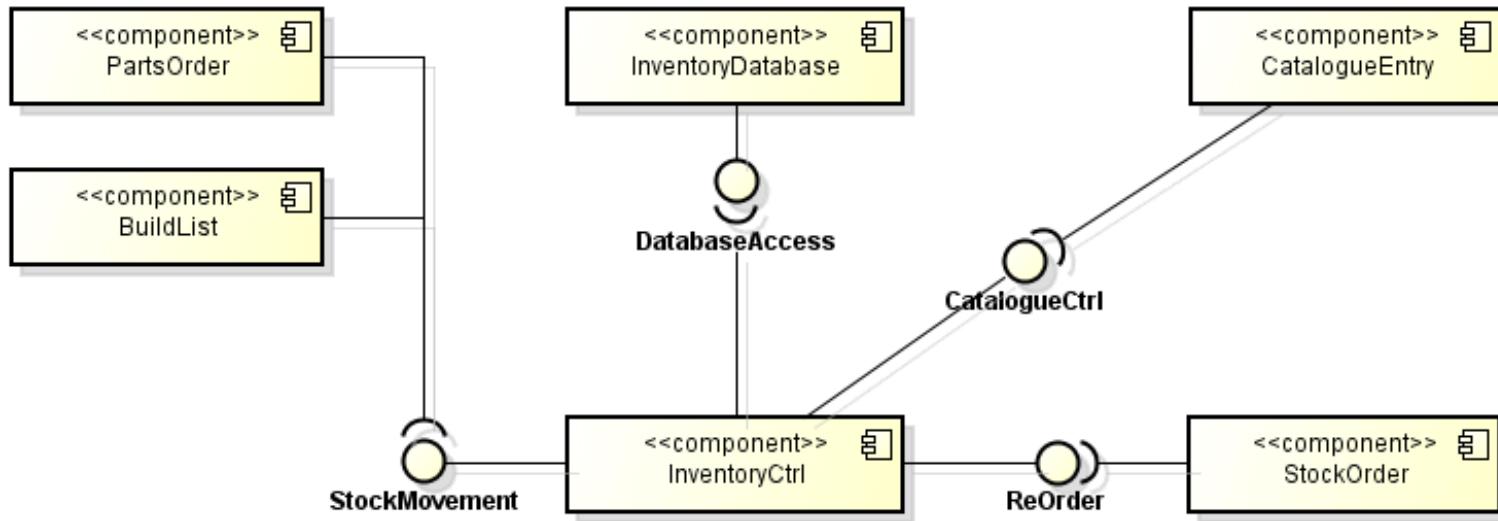
Activity 4

- The **Conversion-Management component** implements the **FeedProvider** and **DisplayConverter** interfaces.
- The **Blog-Data-Source component** implements the **DataSource** interface.
- The **Log4j component** implements the **Logger** interface.
- The BlogDataSoruce component requires the Logger interface of the Log4j
- The Conversion Management component requires the DataSource interface of Blog-Data-Source.
- The **BroadCastEngine component** requires the Feed-provider interface of the Conversion-Manamgenet component
- The **BlogViewer** component requires the DisplayConverter interface of the Conversion-Management component.

Activity 5

- PartsOrder component and BuildList component use InventoryCtrl component through StockMovement interface.
- InventoryDatabase component realizes DatabaseAccess interface which uses by InventoryCtrl.
- Also InventoryCtrl component realizes ReOrder and CatalogueCtrl interfaces that use by StockOrder and CatalogueEntry components respectively.

Activity 5: Answer



Deployment Diagram

Deployment Diagram

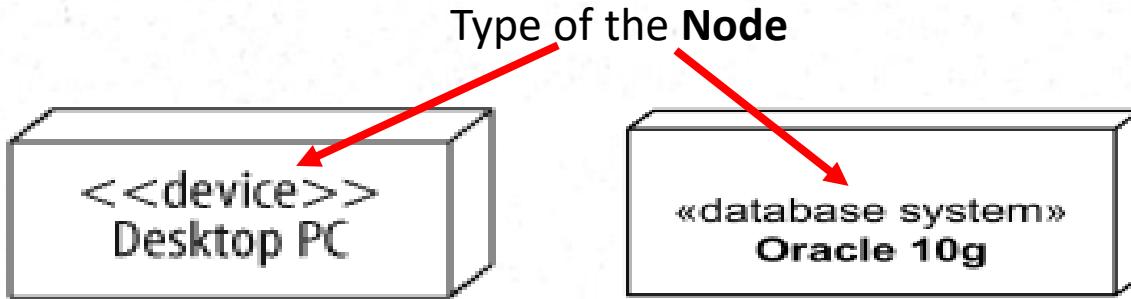
- UML *deployment diagrams* show the physical view of your system, bringing your software into the real world by showing how software gets assigned to hardware and how the pieces communicate

Components of a Deployment Diagram

- Nodes (Hardware / Software units)
- Links between nodes (connections)

Nodes

- A *node* is a hardware or software resource that can host software or related files.
- A node is drawn as a cube with its type written inside.



- There are two types of nodes;
 - Hardware nodes
 - Software nodes (Execution environments)

Hardware Nodes

- Hardware nodes use to show computer hardware.



- It's labeled with the stereotype <>device<> to specify that this is a hardware node.
- The following items are common examples of hardware nodes:
 - Hardware Server
 - Desktop PC
 - Disk drives

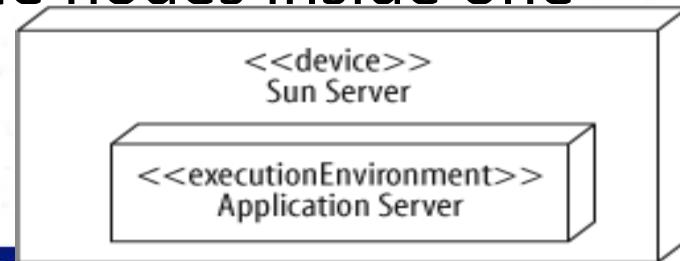
Software Nodes

- Also known as software environments.
- A software node is an application context; not parts of the software going to developed, but a third-party environment that provides services to software going to develop.
- The following items are examples of execution environment nodes:
 - Operating system
 - Web server
 - Application server
 - workflow engine
 - database system
 - web browser
 - J2EE container
- It's labeled with the stereotype <<executionEnvironment>> to specify that this is a software node.

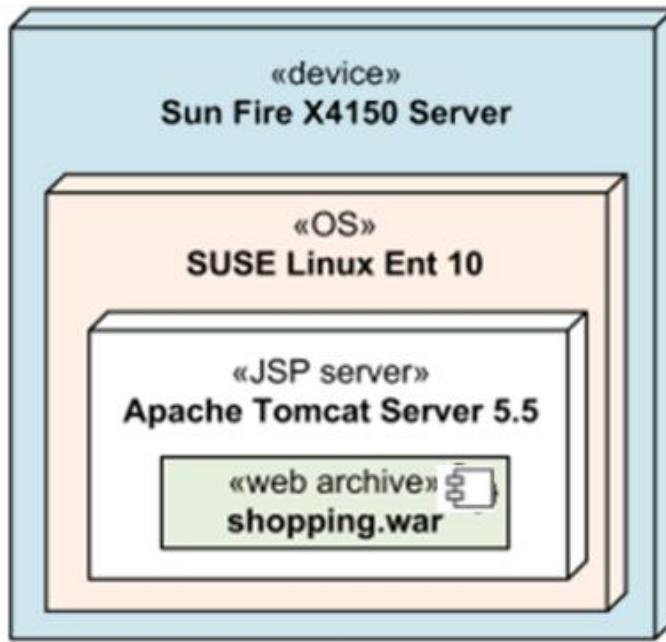


Nested Nodes

- Execution environments do not exist on their own, they run on hardware.
- For example, an operating system needs computer hardware to run on.
- Execution environment resides on a particular device indicated by placing the nodes inside one another, nesting them.



Nested Node: Example



Activity 6

Draw a nested node according to the following description.

A Hardware server contains a processor. A J2EEServer execution node resides inside the processor. A PerformanceEJB component is installed in the J2EEServer execution node .

Artifacts

- *Artifacts* are physical files that execute or are used by the software.
- Common artifacts you'll encounter include:
 - Executable files, such as *.exe* or *.jar* files
 - Library files, such as *.dlls* (or support *.jar* files)
 - Source files, such as *.java* or *.cpp* files
 - Configuration files that are used by software at runtime, in formats such as *.xml*, *.properties*, or *.txt*
- An artifact is shown as a rectangle with the stereotype <<artifact>>, or the document icon in the upper right hand corner, or both.

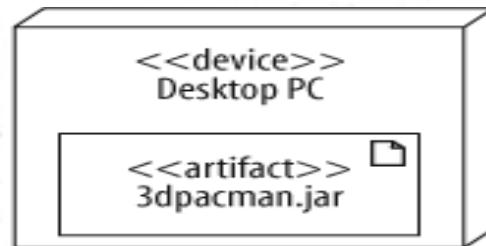
<<artifact>>
3dpacman.jar

3dpacman.jar

<<artifact>>
3dpacman.jar

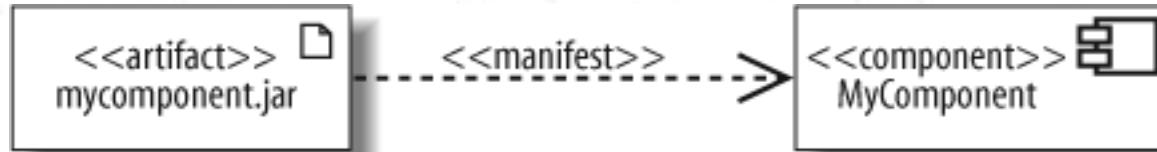
Artifacts

- Deploying an Artifact to a Node.



- Binding an Artifact to a Component.

If an artifact is the physical actualization of a component, then the artifact *manifests* that component.



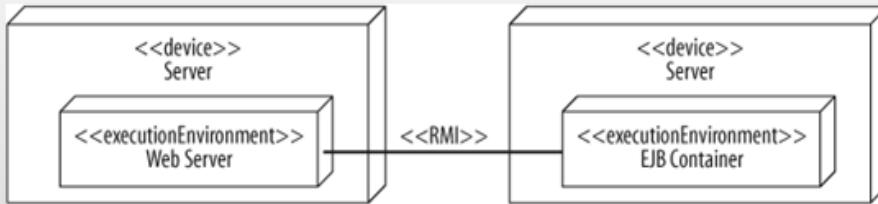
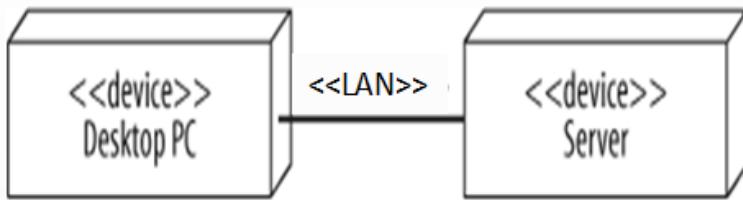
Communication paths / Links

- To get its job done, a node may need to communicate with other nodes.
- For example, a client application running on a desktop PC may retrieve data from a server using TCP/IP.
- *Communication paths / links* are used to show that nodes communicate with each other at runtime.
- Communication paths can be a physical connection (ex: LAN / WAN) or a protocol such as TCP/IP.

Communication paths / Links

- A communication path is drawn as a solid line connecting two nodes. The type of communication is shown by adding a stereotype to the path.
- communication paths also can be shown in between execution environment nodes.
- Example: A web server communicating with an EJB container through RMI
- When deployment targets are **execution environments**, communication path will typically represent some **protocol**.

Communication paths / Links



Activity 7

- Given below is a partial description of a web-based application developed for an online banking system “Smart-E-Banking”. Model a physical diagram according the given description.
- UI Component contains all the user interfaces of this system. It is installed in the main web server called UI_SEB WebServer. This web server runs in a Lenovo ThinkServer hardware server. It is installed windows OS. UI component is responsible to create i_SEB interface, which used by the SmartBanking_Webclient web application of the desktop.

Activity 7 cont...

- The desktop which is having windows OS, access the system through “SmartBanking_Webclient” web application connected to the UI component.
- There is a connection in between browser and Lenovo ThinkServer OS via HTTP.

References

- The Unified Modeling Language Reference Manual, Second Edition.
- Learning UML 2.0 by Kim Hamilton, Russ Miles.

Software Engineering (IT2020)

2022

Lecture 6 - Software Testing



Session Outcomes

- White Box Testing
- White Box Testing Techniques
 - Statement Coverage
 - Branch Coverage
- Unit Testing
 - Junit
- Non-functional Testing

Story So Far ...

- So far we have discussed about
 - Types of testing
 - Black Box testing
 - White box testing
 - Black box testing strategies
 - Equivalence Class Testing
 - Boundary Value Testing
 - Software testing levels
- Now lets look into white box testing in more details...



What is Software Testing?

**Is a systematic process to identify bugs/errors in a software program
and**

**Verify whether the system has developed according to the
specification or**

It provides the functionalities required by the client

Why Testing is necessary?

- Executing a program with the intent of finding an *error*.
- To check if the system meets the requirements and the system does what it is expected to do.
- To check if it can be executed successfully in the Intended environment.
- To check if the system is “ Fit for purpose”.



Verification and Validation

- Verification:

"Are we building the product right"

- The software should conform to its specification – functional and non-functional requirements.
- It is a static method of checking the documents, plans, code, requirements, designs and specifications.
- Typically involves reviews, walkthroughs, and meeting to evaluate



- Validation:

"Are we building the right product"

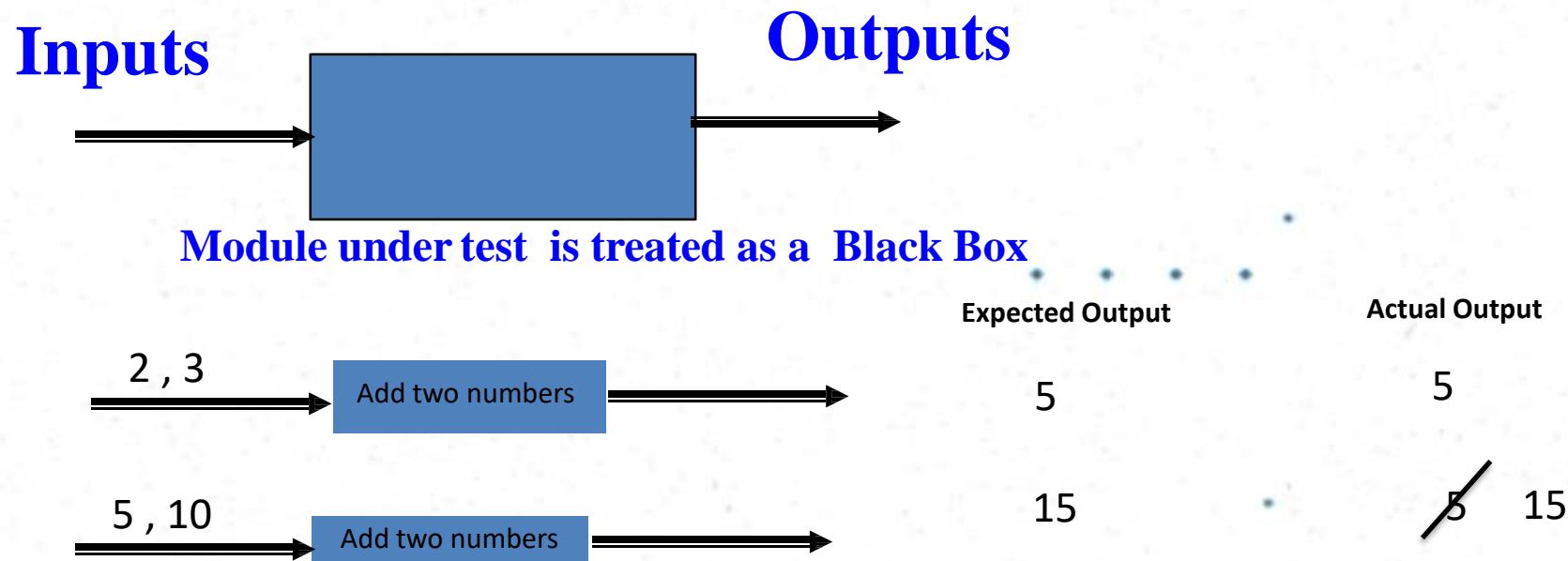
- The software should do what the user really requires which **might** be different from specification.
- Typically involves actual testing and takes place after verifications are completed.



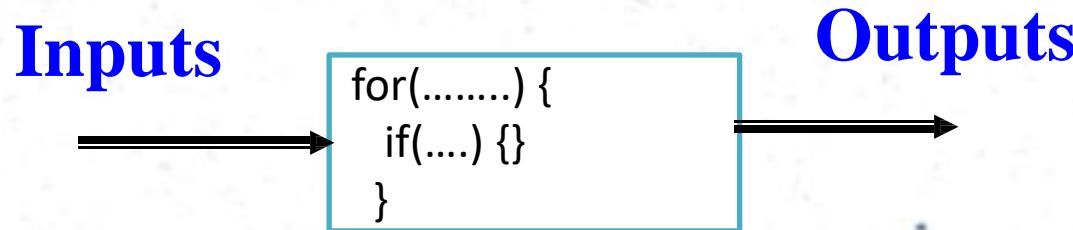
Verification	Validation
1. It is the process to ensure whether we are developing the product right or not. (Whether we are developing it according to the plans and specifications)	1. It is the process to test/check whether the product we developed is correct. Whether it satisfies the client requirements.
2. Is a static method of checking the documents, designs, program code, db schemas and specifications.	2. Is a dynamic method of testing the real product.
3. Inspections, reviews and walkthroughs	3. Testing frameworks , testing strategies are used.
4. Low – level activity	4. High – level activity
5. It does not involve code execution	5. It involves code execution
6. Low cost compared to validation tests	6. Costly compared to verification tests

Black Box Testing

- Testing focus on the software functional requirements, and input/output.



White Box Testing



- Testing is based on the structure of the program
 - In white box testing internal structure of the program is taken into account.
 - The test data is derived from the structure of the software.
 - Should have the programming knowledge.



Quality Assurance Engineers, Software Engineers, Programmers perform these tests.

- Most of the defects found in Unit and Integration is done using the white box testing.
- Tests are based on coverage of code statements, branches, paths, conditions.

White Box Testing - Techniques

Statement Coverage

- Execute all statements at least once

Branch (Decision/Edge) Coverage

- Execute each decision direction at least once

Condition (Predicate) Coverage

- Execute each decision with all possible outcomes at least once



Decision/Condition Coverage

- Execute all possible combinations of condition outcomes in each decision

Multiple Condition Coverage

- Invoke each point of entry at least once
- Execute all statements at least once

Statement Coverage

Statement coverage involves execution of all the executable statements in the source code at least once.

Minimum number of test cases we need to execute **all the statements** in the program at least once.

Methodology

- Design test cases so that every statement in a program is executed at least once.

Principal Idea

- Unless a statement is executed, we have no way of knowing if an error exists in that statement.



Statement coverage is used to derive scenario based upon the structure of the code under test.

$$\text{Statement Coverage} = \frac{\text{No of executed statements}}{\text{Total no of statements}} * 100\%$$

Example

- Calculate the no of minimum test cases needed for full statement coverage for the given scenario.

```
Printsum (int a, int b) {  
    int result = a+ b;  
    If (result> 0)  
        Print ("Positive", result);  
    Else  
        Print ("Negative", result);  
}
```

Step 1: What is the total number of statements in the code?

```
Printsum (int a, int b) { → 1
    int result = a+ b; → 2
    If (result> 0) → 3
        Print ("Positive", result); → 4
    Else → 5
        Print ("Negative", result); → 6
} → 7
```

Total no of statements = 7

Step 2: Find out the executed no of statements for Test case 1

Test Case 1 – if a=3, b=9

No of executed statements = 5

```
Printsum (int a, int b) {  
    int result = a+ b;  
    If (result> 0)  
        Print ("Positive", result);  
    Else  
        Print ("Negative", result);  
}
```

- | | |
|---|---|
| 1 | ✓ |
| 2 | ✓ |
| 3 | ✓ |
| 4 | ✓ |
| 5 | ✗ |
| 6 | ✗ |
| 7 | ✓ |

Step 3: Find the statement coverage.

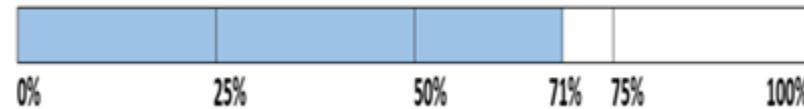
Test Case 1 – if a=3, b=9

```
Printsum (int a, int b) {  
    int result = a+ b;  
    If (result> 0)  
        Print ("Positive", result);  
    Else  
        Print ("Negative", result);  
}
```

No of executed statements = 5

Total no of statements = 7

Statement coverage = $5/7 * 100 = 71\%$



Step 4: Again check the statement coverage for Test case 2

Test Case 2 – if a= -2, b= -3

Printsum (int a, int b) {

 int result = a+ b;

 If (result> 0)

 Print ("Positive", result);

 Else

 Print ("Negative", result);

}

1 

2 

3 

4 

5 

6 

7 

Step 4: Again check the statement coverage for Test case 2

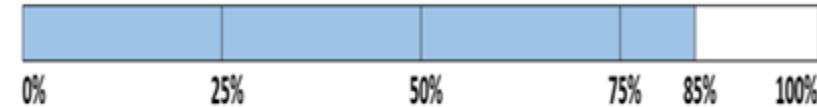
Test Case 2 – if a=-2, b=-3

```
Printsum (int a, int b) {  
    int result = a+ b;  
    If (result> 0)  
        Print ("Positive", result);  
    Else  
        Print ("Negative", result);  
}
```

No of executed statements = 6

Total no of statements = 7

Statement coverage = $6/7 * 100 = 85\%$

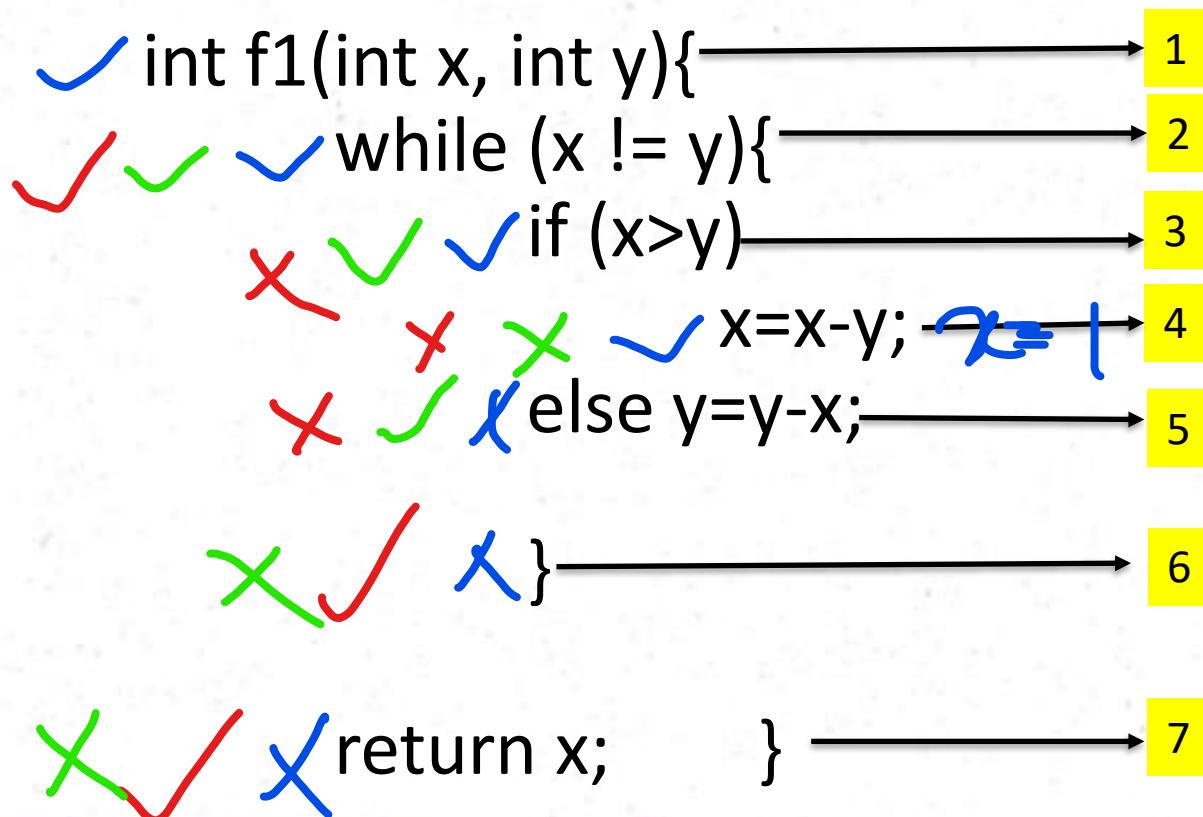


Overall we can say all the statements are fully covered by using the two test cases. So the overall statement coverage of 100% can be achieved by the above two test cases.

Activity *at least one*

- Calculate the no of minimum test cases needed for full statement coverage for the given scenario.

① ②



Total no of statements = 7

$$T \in \lceil \frac{3}{2} \rceil$$

$$\frac{7}{7} \times 100\% = 100\%$$

Branch Coverage

Branch coverage covers both the true and false conditions unlikely the statement coverage.

Minimum number of test cases we need to execute **all the branches** in the program at **least once**.

Methodology

- Test cases are designed such that different branch conditions given true and false values in turn.

Principal Idea

- This technique checks every possible path (decisions).
- A decision is an IF statement, a loop control statement (e.g. DO-WHILE or REPEAT-UNTIL), or a CASE statement, where there are two or more outcomes from the statement.

- A branch is the outcome of a decision, so branch coverage simply measures which decision outcome have been tested.
- This takes more in depth view of the source code compared to statement coverage.

$$\text{branch Coverage} = \frac{\text{No of executed branches}}{\text{Total no of branches}} * 100\%$$

Control Flow Graph

Node



Edge

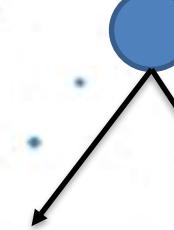


Nodes

Simple



Decision



Simple Statements

Decision Statements

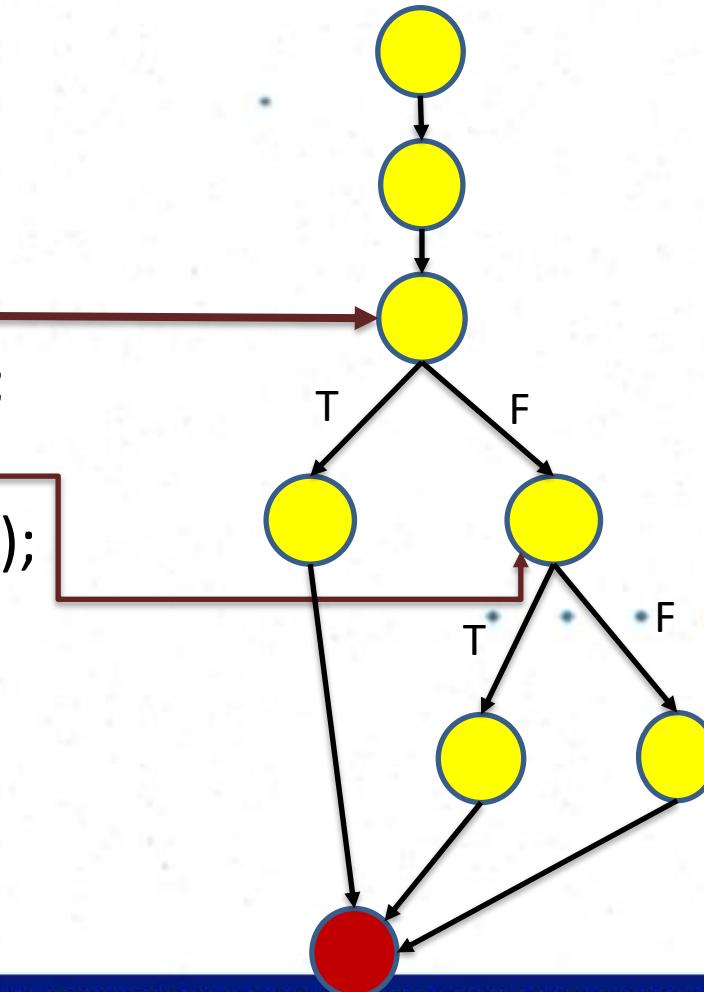
Example

Calculate the no of minimum test cases needed for full branch coverage for the given scenario.

```
Printsum (Int a, Int b) {  
    Int result = a+ b;  
    If (result> 0)  
        Print ("Positive", result);  
    else if (result<0)  
        Print ("Negative", result);  
    else  
        do nothing;  
}
```

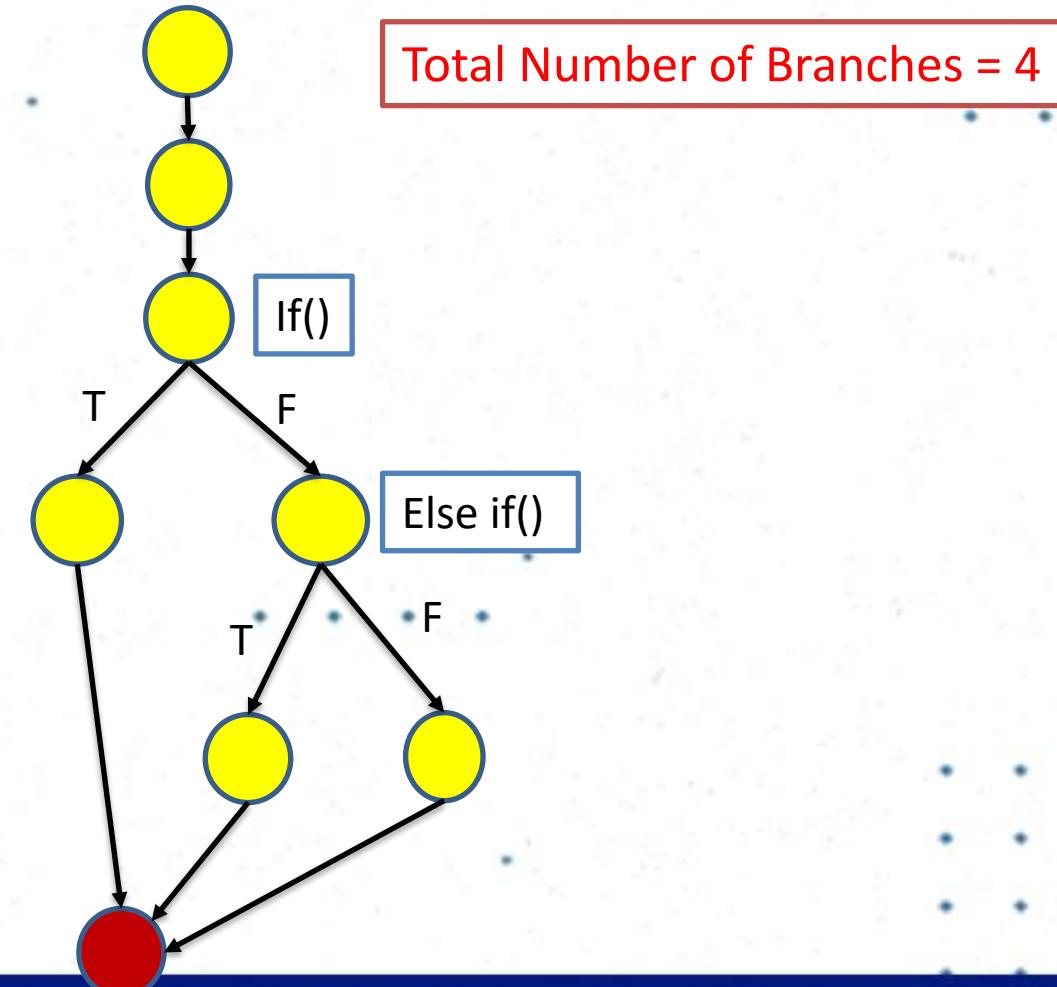
Step 1: Come up with a simple control flow graph for the given code.

```
Printsum (Int a, Int b){  
    Int result = a+b;  
  
    if(result>0)  
        Print ("Positive", result);  
    else if (result<0)  
        Print ("Negative", result);  
    else  
        do nothing;  
}
```



Step 1: Total number of branches in the program.

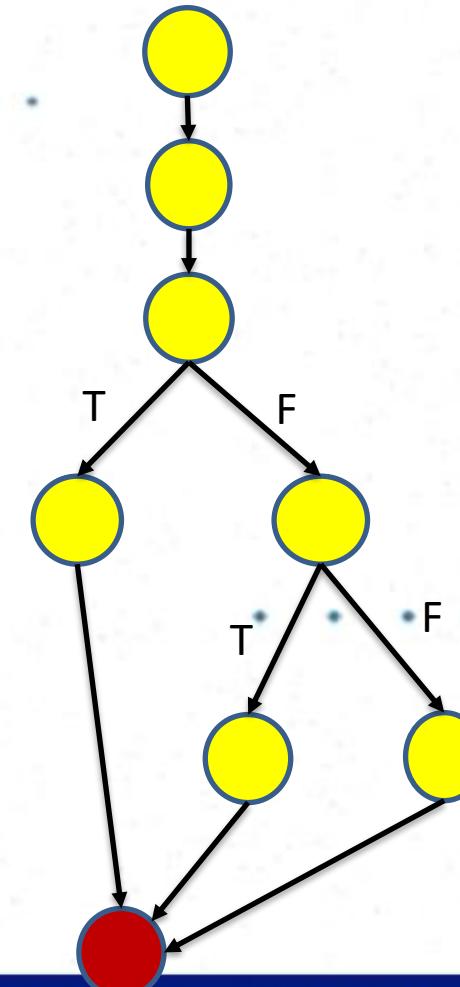
```
Printsum (Int a, Int b){  
    Int result = a+b;  
  
    if(result>0)  
        Print ("Positive", result);  
    else if (result<0)  
        Print ("Negative", result);  
    else  
        do nothing;  
}
```



Step 3: Traverse through the graph when a=3 and b=9.

Test case 1 :- a=3,b=5

```
Printsum (Int a, Int b){  
    Int result = a+b;  
  
    if(result>0)  
        Print ("Positive", result);  
    else if (result<0)  
        Print ("Negative", result);  
    else  
        do nothing;  
}
```

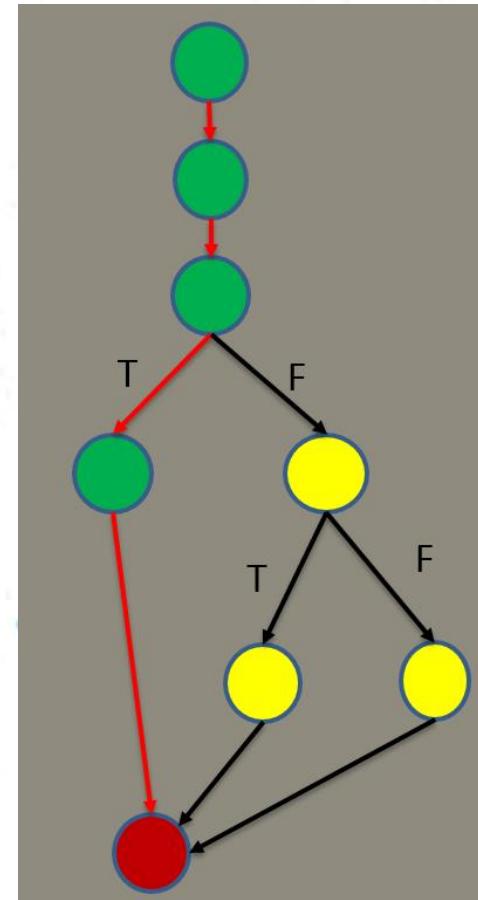
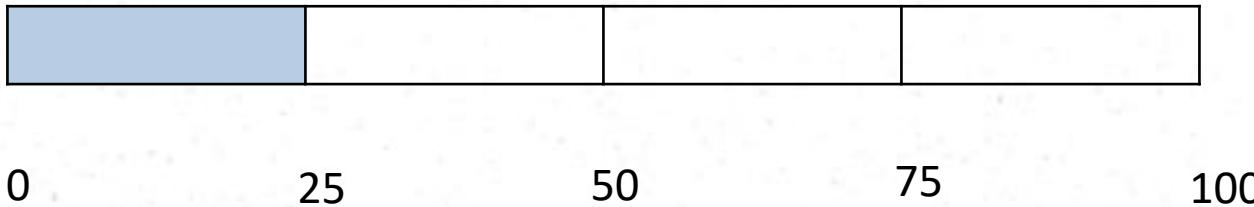


Step 4: Calculate the branch coverage when a=3 and b=5.

Test case 1 :- a=3, b=5

This test case covers the path/branch highlighted.

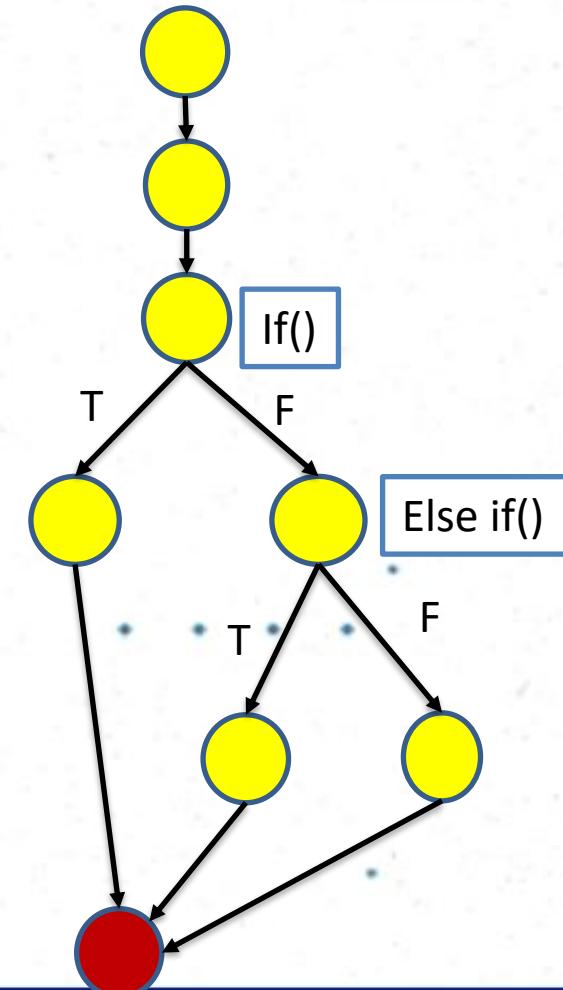
$$\text{branch coverage} = 1/4 * 100 = 25\%$$



Step 5: Calculate the branch coverage when a=-5 and b=-8.

Test case 2 – a=-5,b=-8

```
Printsum (Int a, Int b){  
    Int result = a+b;  
  
    if(result>0)  
        Print ("Positive", result);  
    else if (result<0)  
        Print ("Negative", result);  
    else  
        do nothing;  
}
```

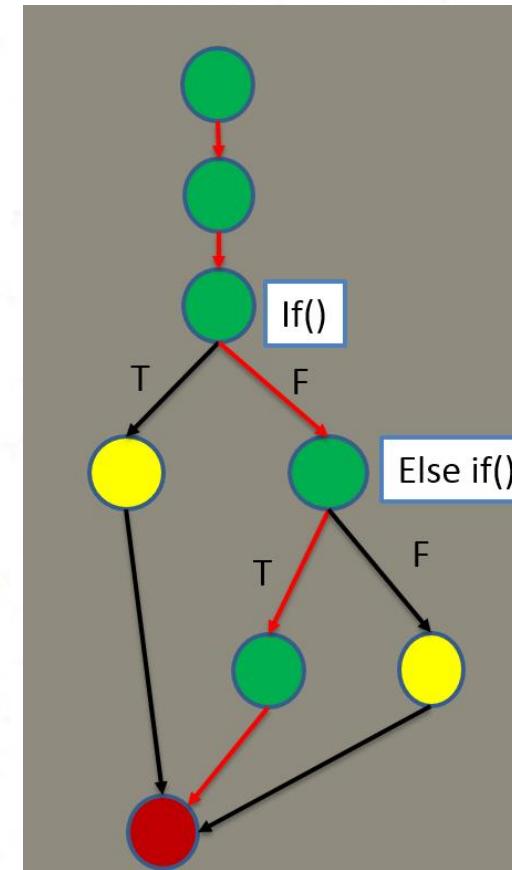
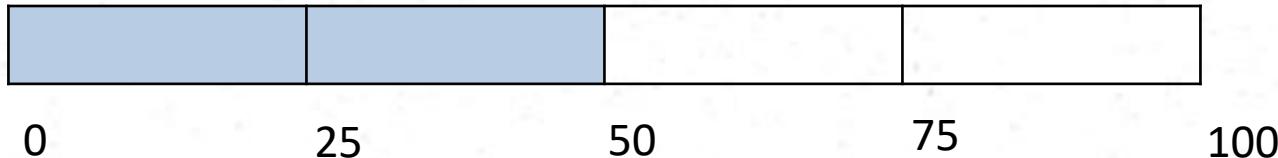


Step 5: Calculate the branch coverage when a=-5 and b=-8.

Test case 2 – a=-5,b=-8

This test case covers the path highlighted. Two branches are covered.

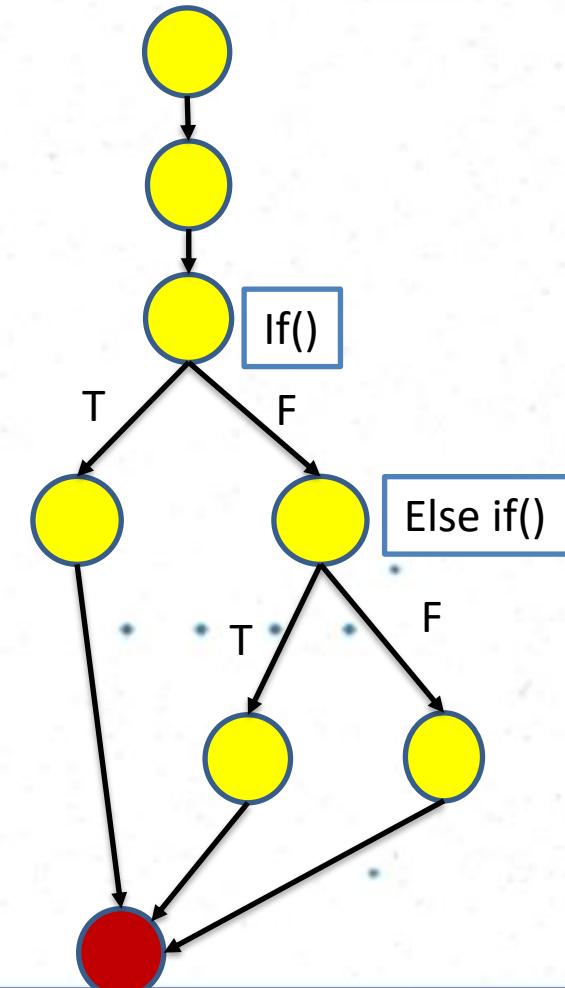
Branch coverage=2/4*100=50%



Step 5: Calculate the branch coverage when a and b are 0.

Test case 3 :- a=0, b=0

```
Printsum (Int a, Int b){  
    Int result = a+b;  
  
    if(result>0)  
        Print ("Positive", result);  
    else if (result<0)  
        Print ("Negative", result);  
    else  
        do nothing;  
}
```



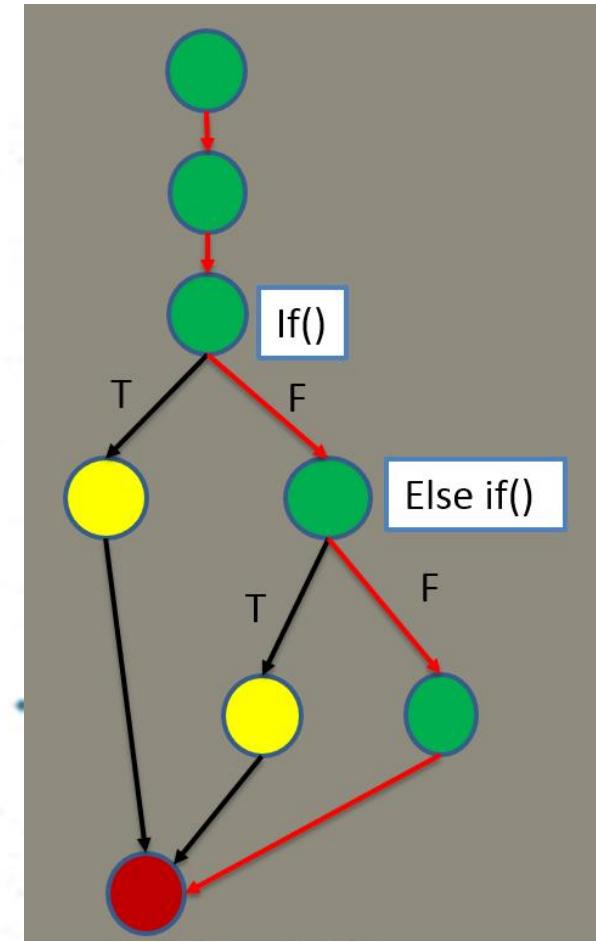
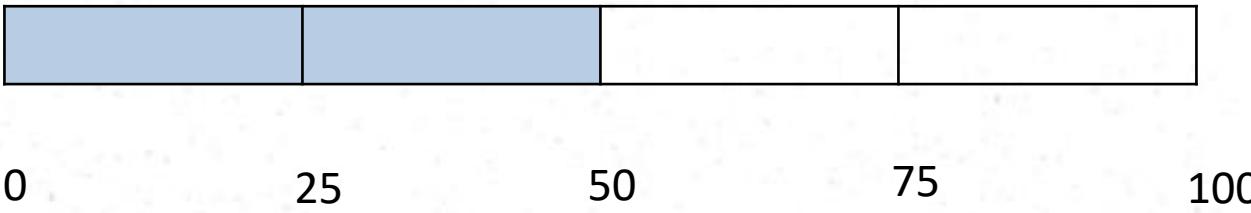
Step 5: Calculate the branch coverage when a and b are 0.

Test case 3 – a=0,b=0

This test case covers the path highlighted.

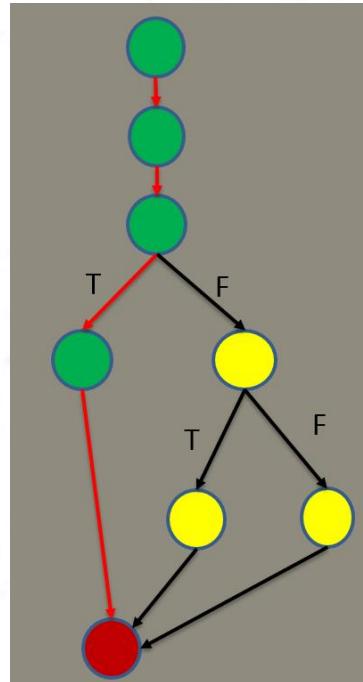
Two branches are covered.

$$\text{branch coverage} = 2/4 * 100 = 50\%$$



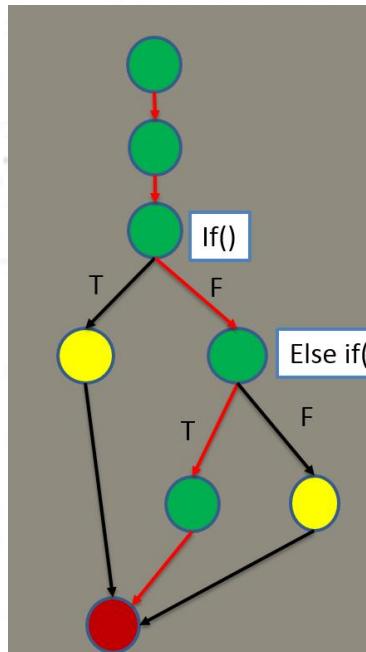
By using minimum three test cases we can test all the branches.

Test case 1 :- a=3, b=5



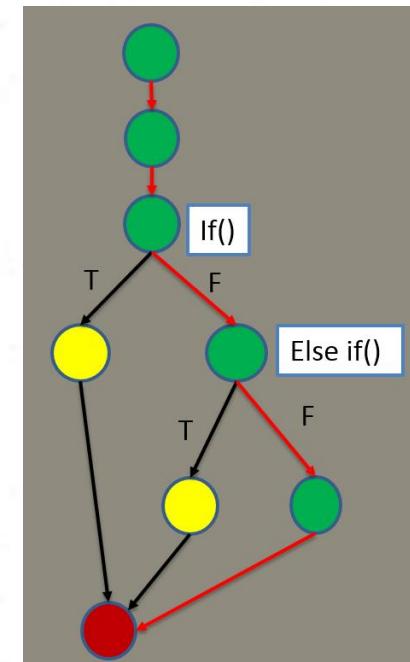
$$\text{branch coverage} = 1/4 * 100 = 25\%$$

Test case 2 :- a=-5,b=-8



$$\text{branch coverage} = 2/4 * 100 = 50\%$$

Test case 3 :- a=0,b=0



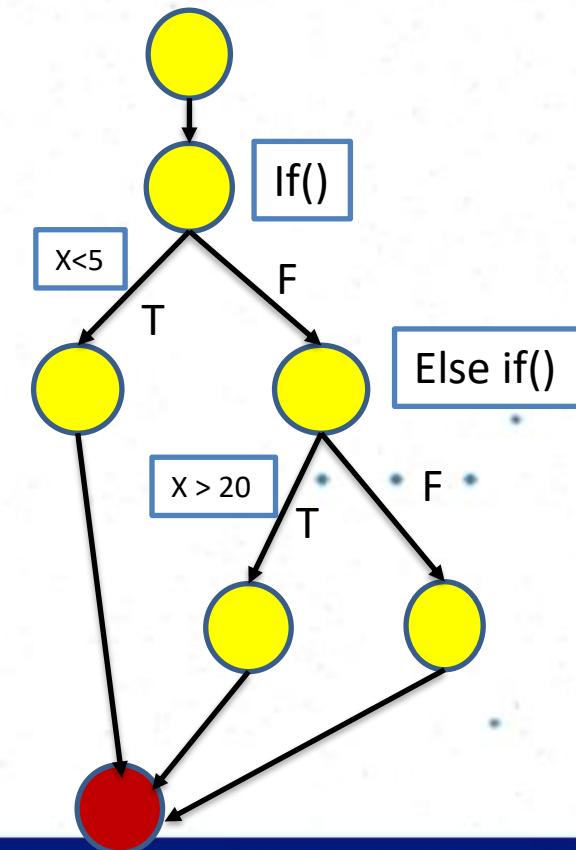
$$\text{branch coverage} = 2/4 * 100 = 50\%$$

By using minimum three test cases we can test all the branches.

Activity

What is the minimum number of test cases required to achieve full branch coverage for the program segment given below?

```
Void print(int x) {  
    if( x < 5){  
        ..... }  
    else if (x > 20){  
        ..... }  
    else {  
        ..... }  
}
```



Total Number of Branches = 4



What do we test?

- Functional Test
 - Unit Testing
 - Integration Testing
 - System Testing
 - Acceptance Testing
- Non – Functional Test
 - Performance
 - Stress / Load
 - Usability
 - Scalability etc.

Functional Testing

- Unit testing : Individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
- Integration testing : Several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
- System testing: Some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.
- Acceptance Testing: Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.



Functional Test - Unit Testing

- The most ‘micro’ scale of testing.
- Tests done on particular functions or code modules. it is the testing of single entity (class or method).
- Requires knowledge of the internal program design and code.
- Done by Programmers (not by testers).
- Unit testing can be done in two different ways.
 - Manual Testing
 - Automated Testing



Manual Testing	Automated Testing
Executing a test cases manually without any tool support is known as manual testing.	Taking tool support and executing the test cases by using an automation tool is known as automation testing.
Time-consuming and tedious – Since test cases are executed by human resources, it is very slow and tedious.	Fast – Automation runs test cases significantly faster than human resources.
Huge investment in human resources – As test cases need to be executed manually, more testers are required in manual testing.	Less investment in human resources – Test cases are executed using automation tools, so less number of testers are required in automation testing.
Less reliable – Manual testing is less reliable, as it has to account for human errors.	More reliable – Automation tests are precise and reliable.
Non-programmable – No programming can be done to write sophisticated tests to fetch hidden information.	Programmable – Testers can program sophisticated tests to bring out hidden information.



Why Unit Testing?

- Faster Debugging
- Faster Development
- Better Design
- Excellent Regression Tool
- Reduce Future Cost

Unit Testing Frameworks

- What are Unit Testing Frameworks?
 - It's a set of guidelines which will help to run the unit testing.
- Examples of UTFs and Where to get them?
 - www.junit.org
 - www.nunit.org
 - www.xprogramming.com



Characteristics of UTFS

- Most UTFS target OO and web languages
- UTFS encourage separation of business and presentation logic
- Tests written in same language as the code
- Tests are written against the business logic
- GUI and command line test runners
- Rapid feedback



JUnit (www.junit.org)

- Java-based unit testing framework
- Elegantly simple
- Easy to write unit tests
- Easy to manage unit tests
- Open source = Free!
- Use to incrementally build a test suite
 - write the tests as you write the code...
 - JUnit promotes the idea of "first testing then coding", which emphasizes on setting up the test data for a piece of code that can be tested first and then implemented.



What is a Unit Test Case?

- A Unit Test Case is a part of code, which ensures that another part of code (method) works as expected.
- A formal written unit test case is characterized by a **known input** and an **expected output**, which is worked out before the test is executed.
- There must be at least two unit test cases for each requirement – one positive test and one negative test.

Eg: Try to delete an existing employee in the system -> Positive Test Case

Try to delete a non-existing employer in the system -> Negative Test Case



Unit Testing on JUnit

1. A unit test consists of a “**test class**” normally corresponding to a specific class in your project – for a class named MyClass the test might look like this,

```
import org.junit.After;  
import org.junit.Before;  
import org.junit.Test;  
import static org.junit.Assert.*;  
  
public class MyClassTest {  
    ...  
}
```

2. This test class can have a few methods for which you provide so called “decorators”. The decorator tells JUnit that this is a special **test method**.

```
Public class MyClassTest {  
    @Test  
    public void testMyMethod()  
    {  
    }  
}
```

Decorators

- One JUnit class will normally have multiple @Test methods for the different public methods
- It can also have a few other methods in your class under test.

Method Name	Decorator	Description
<code>setUpBeforeClass</code>	<code>@BeforeClass</code>	Runs before all the @Test test methods in your Test class.
<code>tearDownBeforeClass</code>	<code>@AfterClass</code>	Runs after all the @Test test methods in your Test class.
<code>setUp</code>	<code>@Before</code>	Runs before each @Test
<code>tearDown</code>	<code>@After</code>	Runs after each @Test
<code>testMethod</code>	<code>@Test</code>	Used for each individual test.

3. Inside each test method you will be running some code usually calling one method in the class you are testing.

Assertions:

```
String expectedValue = "my expected value";  
String actualValue = myClass.method();  
assertEquals(expectedValue, actualValue);
```

There are multiple types of assertions you can make – but the most important ones are assertEquals, assertNotEquals, assertTrue and assertFalse.



Non-Functional Testing

- Non functional testing us used to test the non-functional requirements of the system.

What are Non-functional requirements?

Basically non functional requirements describe how the system works.

eg: Usability, reliability, Performance etc.

Non functional Test - Performance Testing

- Performance testing, a non-functional testing technique performed to determine the system parameters in terms of **responsiveness and stability under various workload.**
- Performance testing measures the quality attributes of the system, such as scalability, reliability and resource usage.

Load testing

It is the simplest form of testing conducted to understand the behavior of the system under a specific load.

Load testing will result in measuring important business critical transactions and load on the database, application server, etc., are also monitored.

Stress testing

It is performed to find the upper limit capacity of the system and also to determine how the system performs if the current load goes well above the expected maximum.

Spike testing

Spike testing is performed by increasing the number of users suddenly by a very large amount and measuring the performance of the system.

The main aim is to determine whether the system will be able to sustain the workload.

Load Testing

This testing usually identifies,

- The maximum operating capacity of an application.
- Determine whether current infrastructure is sufficient to run the application.
- Sustainability of application with respect to peak user load.
- Number of concurrent users that an application can support, and scalability to allow more users to access it.

- Load testing is commonly used for the Client/Server, Web based applications.

Why?

- Some extremely popular sites have suffered serious downtimes when they get massive traffic volumes.

Examples:

- Popular toy store Toysrus.com, could not handle the increased traffic generated by their advertising campaign resulting in loss of both marketing dollars, and potential toy sales.
- An Airline website was not able to handle 10000+ users during a festival offer.
- Encyclopedia Britannica declared free access to their online database as a promotional offer. They were not able to keep up with the onslaught of traffic for weeks.



SLIIT

Discover Your Future

- There are lot different tools available for performance testing.



References

- Software Engineering, I.Sommerville, 10th ed. , Pearson Education.
- Junit 5 User Guide:
<https://junit.org/junit5/docs/current/user-guide/>

Software Engineering (IT2020) - 2022

Lecture 7 - Design Patterns

.....

1

Session Outcomes

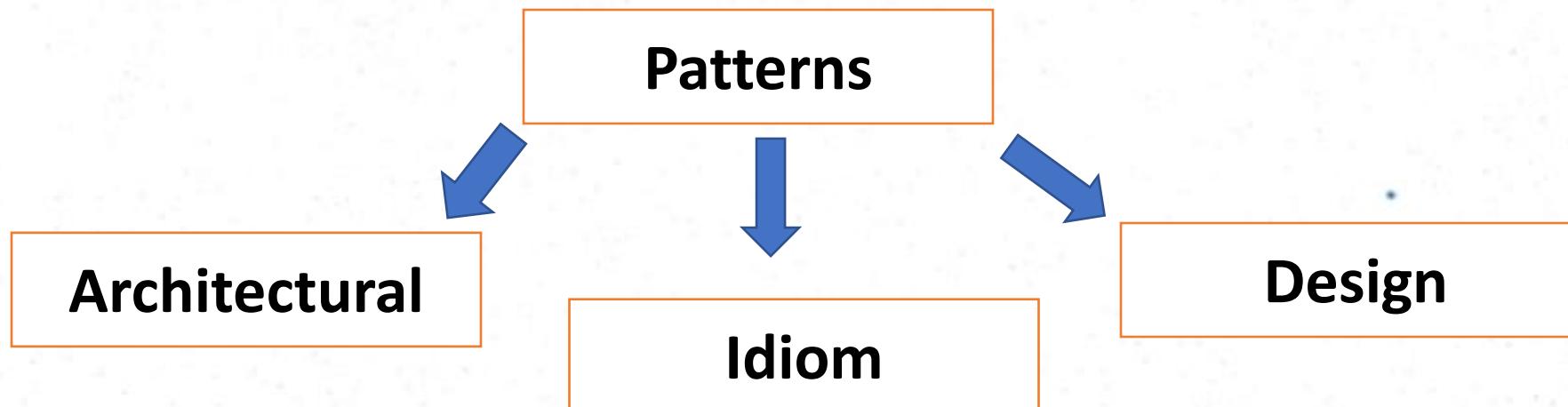
- ❖ Introduction to Design Patterns
- ❖ Creational Patterns
 - ❖ Singleton (Covered in OOP)
- ❖ Structural Patterns
 - ❖ Façade
 - ❖ Decorator
 - ❖ Fly weight
- ❖ Behavioural Patterns
 - ❖ Observer

Introduction to Patterns

- Patterns are an emerging topic in software engineering.
- All well structured systems are full of patterns.
- The principle behind patterns is to identify, document, and hence re-use general solutions to common problems.
- Patterns are discovered, they are not invented.
- Pattern is a template solution to a recurring design problem

Categories of Patterns

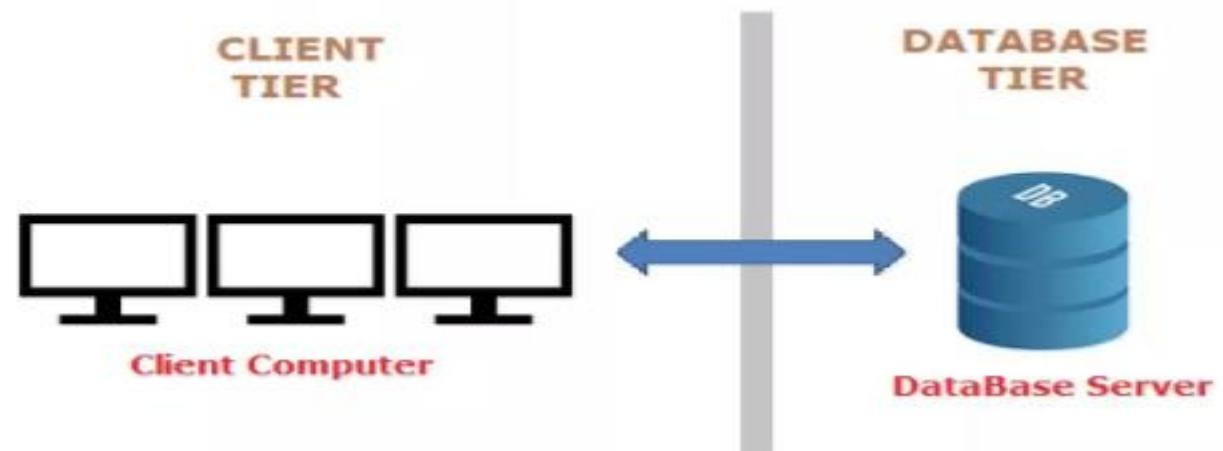
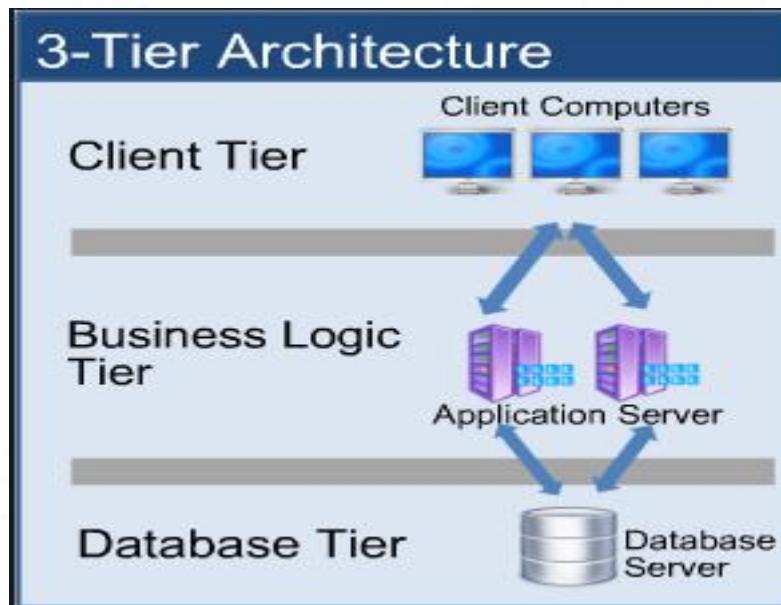
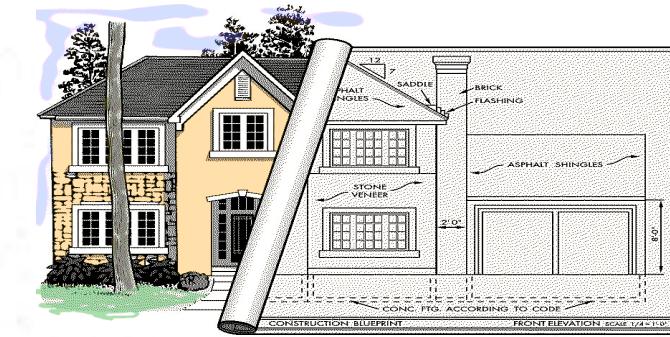
- Software Engineering Patterns can be classified in various ways .
- Classification of patterns used in Software Engineering is shown below.



Architectural Patterns

- These patterns provide an overall architecture (a large scale design) for a software system.
- *Examples :*

3-Tier Architecture, 2-Tier Architecture

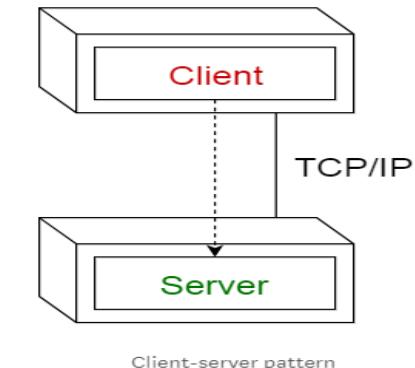


Architectural patterns Examples

- Client/server architecture:

Online applications such as email, document sharing and banking.

- Pipeline architecture -Compilers



Idioms

- These patterns describe a solution to a problem that is **language specific**:
 - The solution is described completely in terms of how it is implemented in a specific programming language.
 - Idioms represent low-level patterns.
- Eg: **Swapping values between variables**

C/C++/Java	Perl	Python
temp = a; a = b; b = temp;	(\$a, \$b) = (\$b, \$a);	a, b = b, a

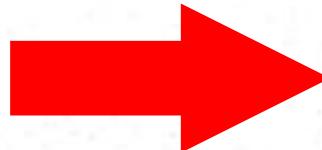
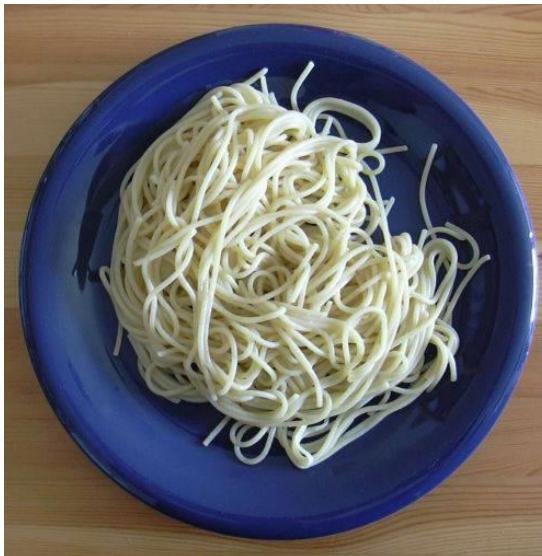
Design Patterns

- Design patterns are solutions to general problems that software developers faced during software development.
- Quote from Christopher Alexander
 - “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice” (GoF,1995).
- Patterns reflect the experience, knowledge and insights of developers who have successfully used these patterns in their own work. (They have been proven)
- They are reusable. Patterns provide a ready-made solution that can be adapted to different problems as necessary.

Why use Design Patterns?

Code Reuse is good – Software developers generally recognize the value of reusing code

- reduces maintenance
- reduces defect rate (if reusing good code)
- reduces development time



Elements of a Design Pattern

- A pattern has four essential elements (GoF)
 - Name
 - Describes the pattern
 - Adds to common terminology for facilitating communication (i.e. not just sentence enhancers)
 - Problem
 - Describes when to apply the pattern
 - Answers - What is the pattern trying to solve?

Elements of a Design Pattern (cont.)

- Solution
 - Describes elements, relationships, responsibilities, and collaborations which make up the design
- Consequences
 - Results of applying the pattern
 - Benefits and Costs
 - Subjective depending on concrete scenarios

How to Describe Design Patterns fully

This is critical because the information has to be conveyed to peer developers in order for them to be able to evaluate, select and utilize patterns.

- A format for design patterns
 - Pattern Name and Classification
 - Intent
 - Also Known As
 - Motivation
 - Applicability
 - Structure
 - Participants
 - Collaborations
 - Consequences
 - Implementation
 - Sample Code
 - Known Uses
 - Related Patterns

Format of a Design Pattern

Name	Name and classification -(taxonomy).
Intent	what the pattern is meant to achieve.
Motivation	why you'd want to use it
Applicability	where you'd want to use it
Structure	what the pattern looks like (diagrams).
Participants	classes used to realize the pattern.

Format of a Design Pattern

Collaborations	how the classes collaborate with each other, and how the system should collaborate with the pattern.
Consequences	good and bad.
Sample Code	
Known Uses	
Related Patterns	

Design Patterns Classification

A Pattern can be classified as

- Creational – Object creation (These patterns will be covered in OOP)
- Structural – Relationship between entities
- Behavioral – Communication between objects

Patterns

Design Patterns		
Creational	Structural	Behavioral
Singleton (Covered in OOP) Abstract Factory Builder Prototype	Decorator Façade Flyweight Adapter Bridge Composite Proxy	Observer Chain of Responsibility Command Iterator Mediator Memento State Strategy Visitor

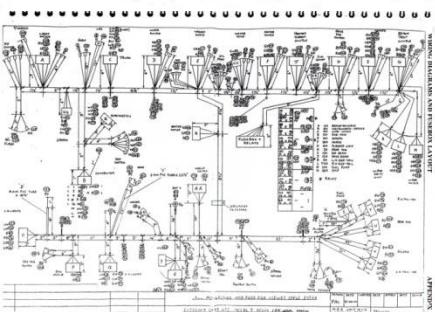
Creational Patterns

Concerns the process of object creation



Behavioural Patterns

Characterize the ways in which classes/objects interact and distribute responsibility



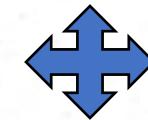
Design Pattern Classification

Final Product



Structural Patterns

Deals with the composition of classes /objects



Behavioral Patterns

Observer Pattern

Observer Definition

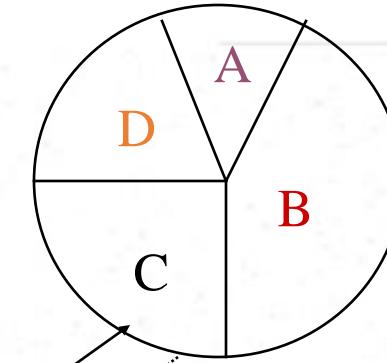
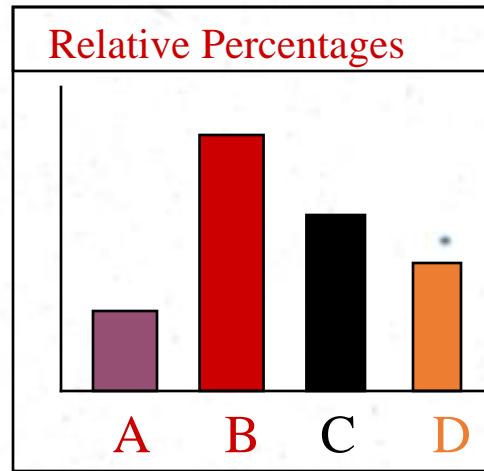
Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Observer

- Problem :We have an object that changes its state quite often. We want to provide multiple views of the current states
E.g :Histogram view, pie chart view , Bar graph ...
- Solution :Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- The system should maintain consistency across the (redundant) views, whenever the state of the observed object changes.
- The system design should be highly extensible
- It should be possible to add new views without having to recompile the observed object or the existing views.

Observer Example

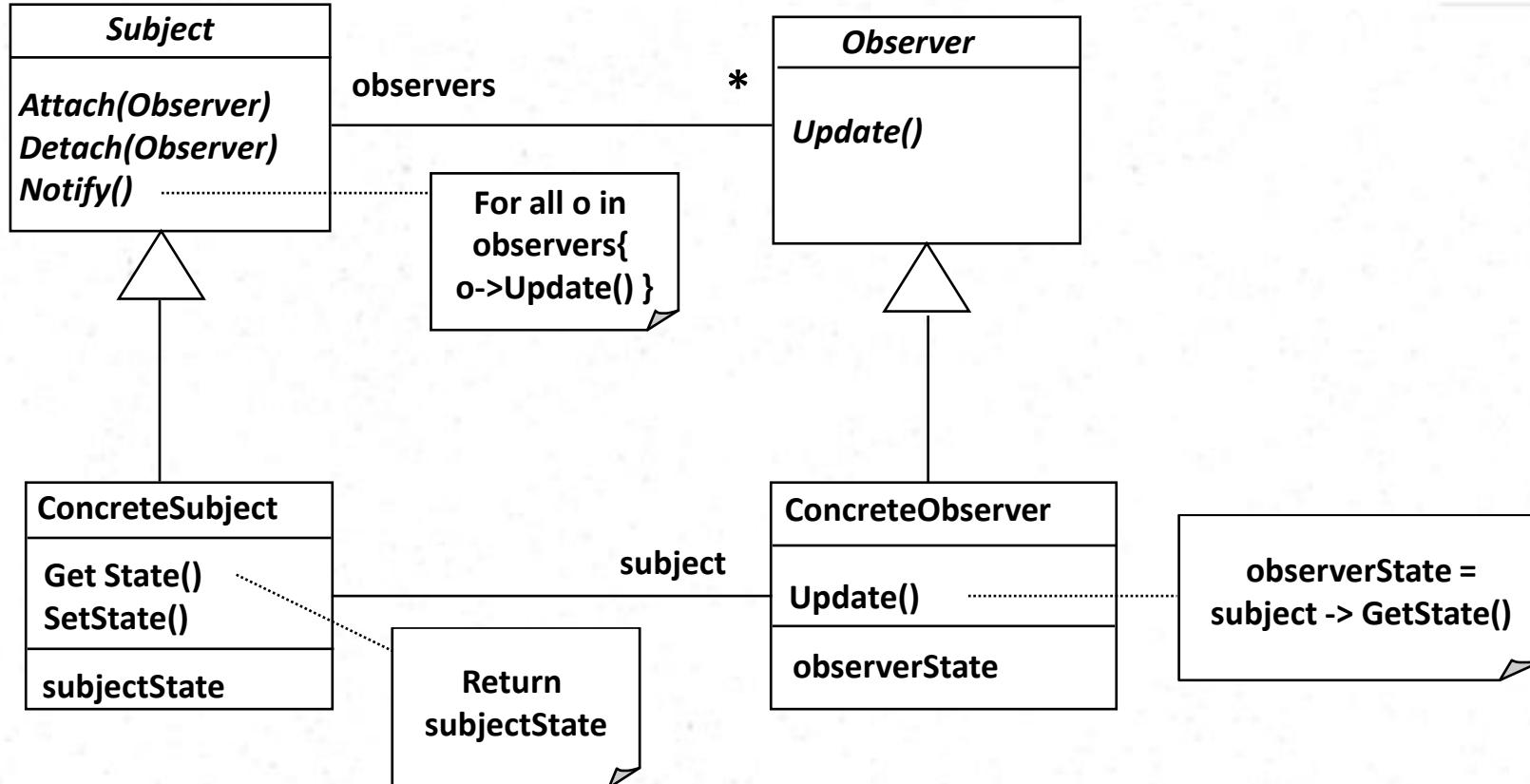
	A	B	C	D
X	15	35	35	15
Y	10	40	30	20
Z	10	40	30	20



→ Change notification
..... Requests, modifications

Application data

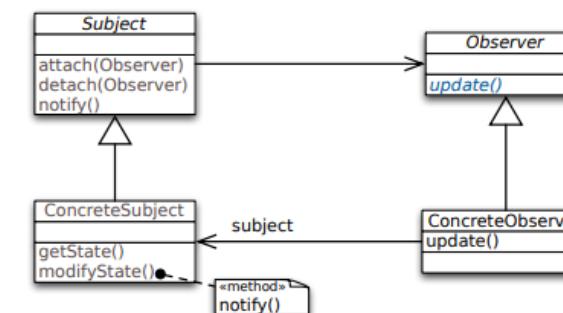
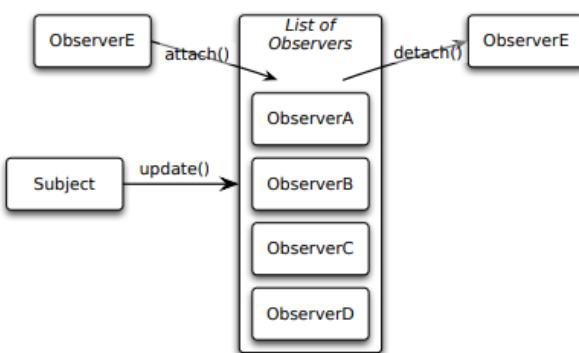
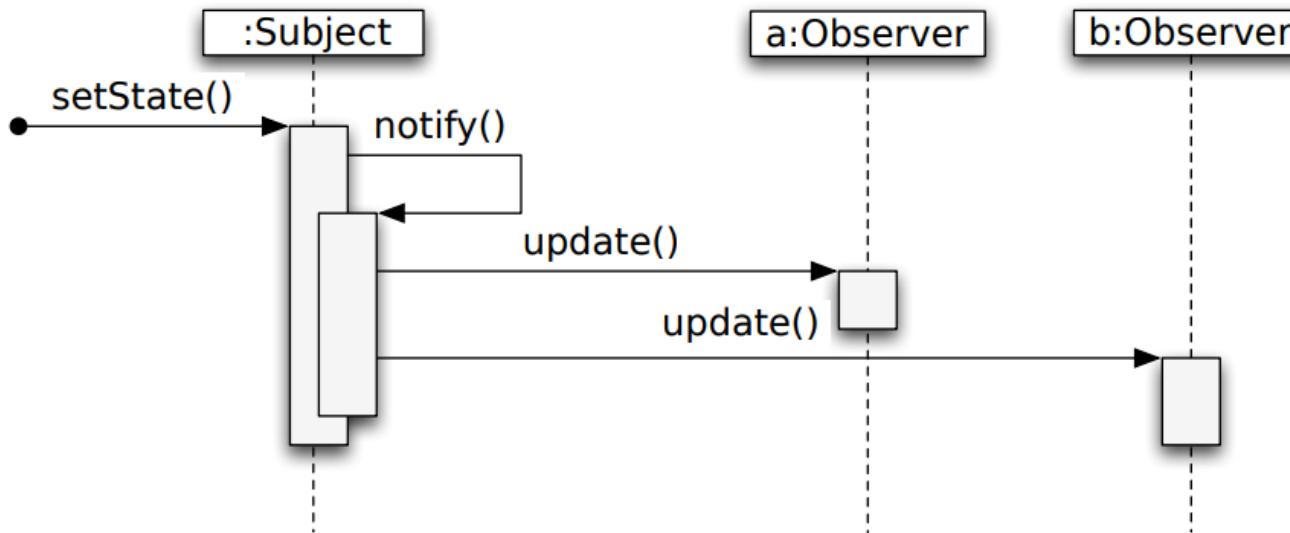
The Structure of Observer



Observer : Participants

- Subject
 - Knows its observers
 - Any number of Observer objects may observe a subject
 - provides an interface for attaching and detaching Observer objects
- Observer
 - defines an updating interface for objects that should be notified of changes in a subject
- ConcreteSubject
 - stores state of interest to ConcreteObserver objects
 - sends a notification to its observers when its state changes
- ConcreteObserver
 - maintains a reference to ConcreteSubject object
 - stores state that should stay consistent with subject's
 - implements the Observer updating interface to keep its state consistent with the subject's.

Observer



Observer Pattern Usage

- **When to use this pattern?**

You should consider using this pattern in your application when multiple objects are dependent on the state of one object as it provides a neat and well tested design for the same.

- **Real Life Uses:**

- It is heavily used in GUI toolkits and event listener. In java the button(subject) and onClickListener(observer) are modelled with observer pattern.
- Social media, RSS feeds, email subscription in which you have the option to follow or subscribe and you receive latest notification.
- All users of an app on play store gets notified if there is an update.

Activity- 1

- RDA of Sri Lanka is going to develop a system to model the cities of the country. Consider the partial requirement of the system given below and answer the questions.
 - The system intends to find distance between cities by maintaining the set of cities as graphical structure. The application needs to maintain multiple views of the distances between cities via a map. One view provides a table of distances between cities as texts, another view maintains the same information as a 2D graph and another view displays this information as a 3D graph. Assume that the City class provides operations for obtaining the distance between any two cities. When new roads are built, the distances between cities can be changed accordingly.
- a) What is the most appropriate design pattern to implement the above requirement? Justify your answer.
Observer Pattern. The subject(city road data of the application) is changing rapidly. So the three representation need to be notified and updated every time.
- b) Draw the class diagram for your selected design pattern in part a).

Observer Pattern Pros & Cons

Pros:

- Supports the principle to strive for loosely coupled designs between objects that interact.
- Allows you to send data to many other objects in a very efficient manner.
- No modification is needed to be done to the subject to add new observers.
- You can add and remove observers at anytime.

Cons

- If not used carefully the observer pattern can add unnecessary complexity.
- The order of Observer notifications is undependable.
- Observable protects crucial methods which means you can't even create an instance of the Observable class and compose it with your own objects, you have to subclass.

Structural Patterns

Decorator Pattern

Decorator Definition

Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.



As a Modern Girl Role



As a warrior character



As a Traditional Character

Decorator : Intent and Motivation

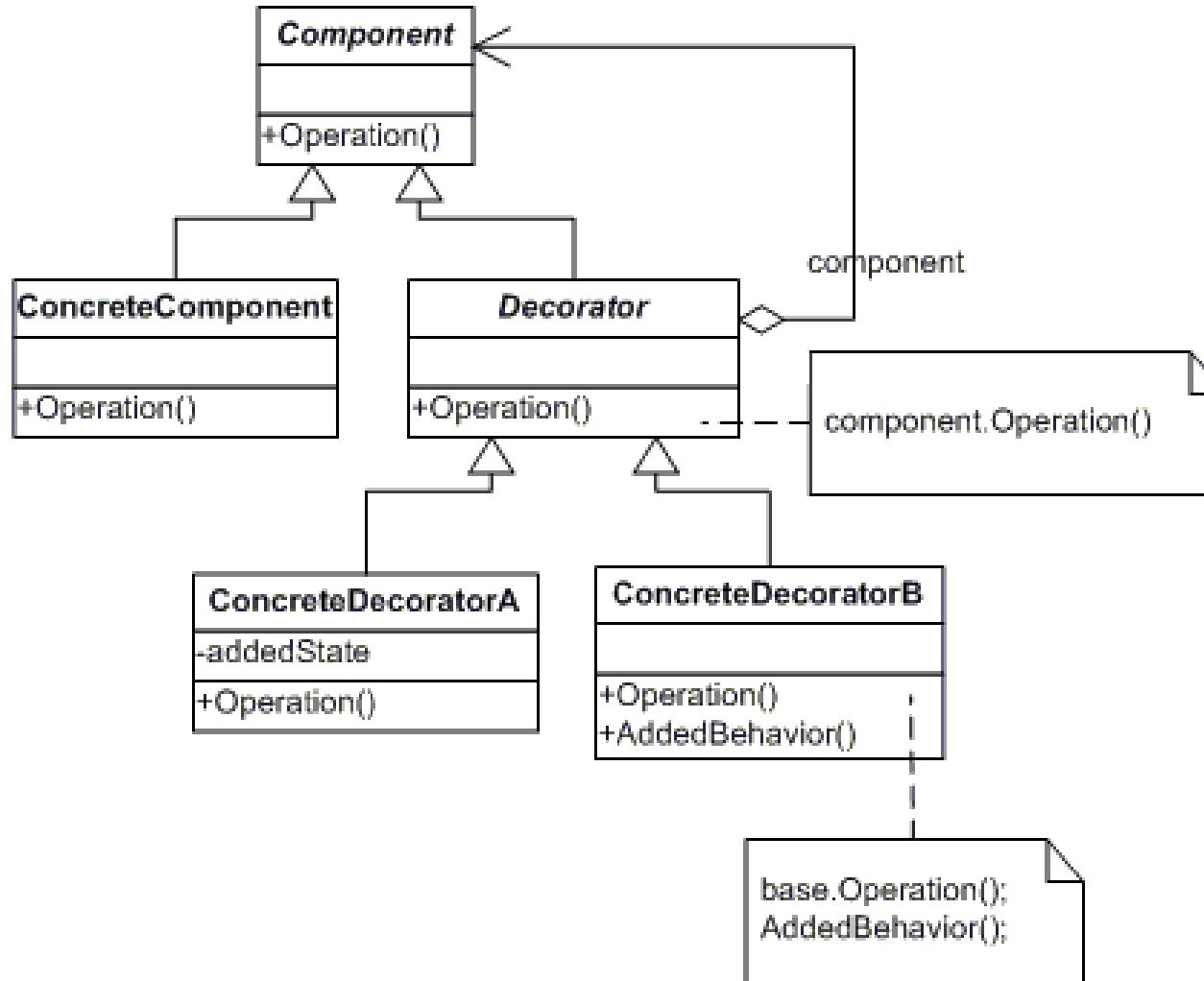
Intent

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality.

Motivation

- We want to add individual responsibilities to individual objects, not to a class.

Decorator : Structure



Decorator

- **Component:** A component can be an object which is to be decorated as well as an object which decorates. The component to be decorated is wrapped by a component which decorates. Component can be interface or an abstract class
- **Concrete Component:** An implementation of Component for which we are going to add new responsibilities. This is the component to be decorated.
- **Decorator:** A Decorator IS-A Component as well as a Decorator HAS-A Component. i.e. Decorator implements a Component as well as it has instance variable of Component. This is the class which serves as abstract class for all concrete decorators.
- **Concrete Decorator:** This can extend behavior or state of a Component. In above example, ConcreteDecoratorA adds newMethod() and ConcreteDecoratorB adds newMember.

Decorator : Applicability

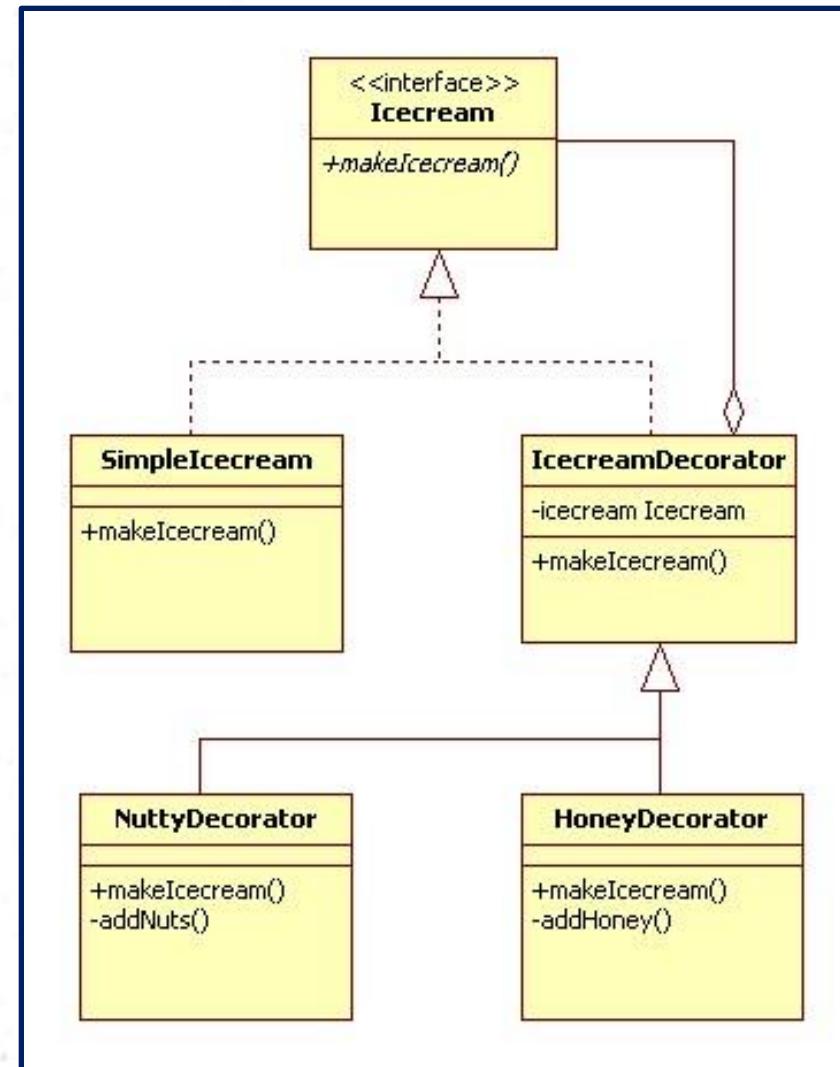
Use Decorator;

- To add responsibilities to individual objects dynamically and transparently (without affecting others)
- For responsibilities that can be withdrawn
- When extension by sub classing is impractical.

Eg: class definition hidden/unavailable for sub classing

Decorator pattern steps

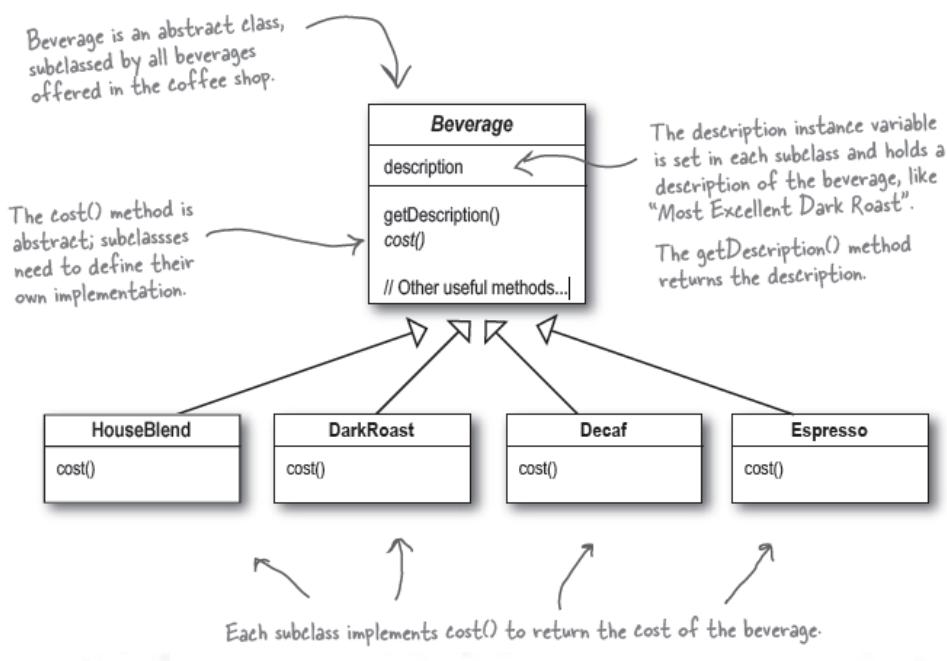
1. Create an interface for the class to be decorated – Icecream
2. Implement that interface with basic functionalities – SimpleIceCream
3. Create an abstract class that contains an attribute type of the interface. - IcecreamDecorator
4. The concrete decorator class will add its own methods. – NuttyDecorator, HoneyDecorator



Activity- 2

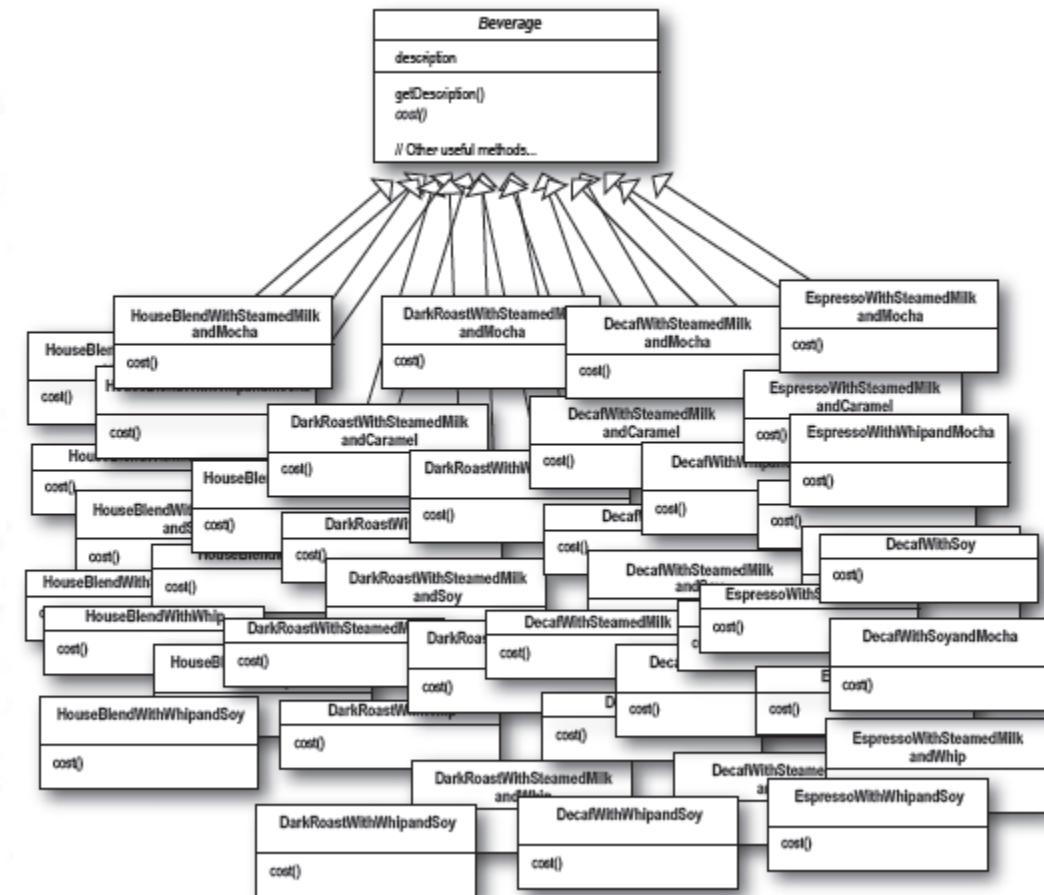
Starbuzz Coffee has made a name for itself as the fastest growing coffee shop around. Because they've grown so quickly, they're scrambling to update their ordering systems to match their beverage offerings.

When they first went into business they designed their classes like this...



Activity -2

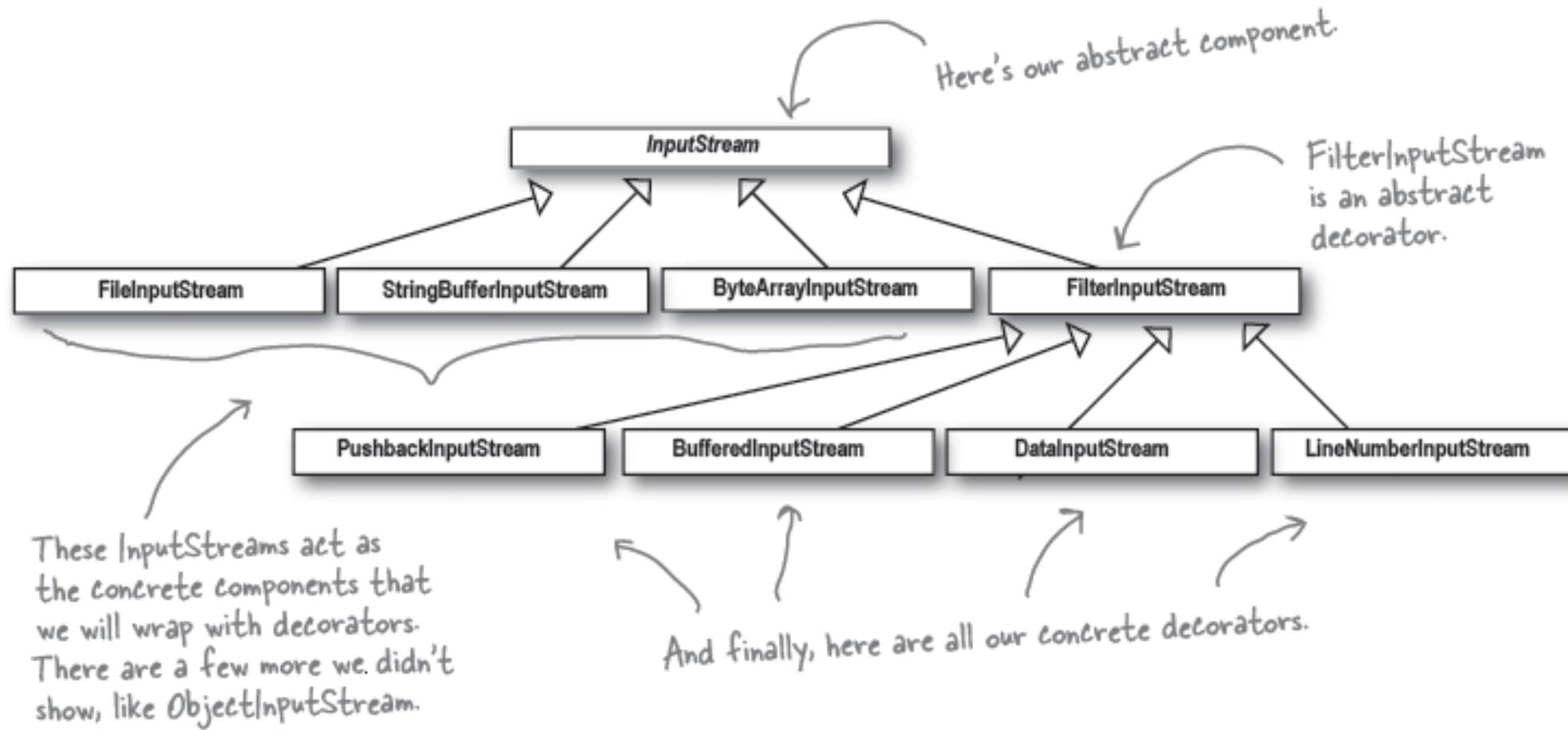
- In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk. Starbuzz charges a bit for each of these. Here's their first attempt...
- Use the Decorator pattern to make this design better.



Activity - Decorator : Participants

- Component (Beverage)- Defines the interface for objects that can have responsibilities added to them dynamically
- ConcreteComponent (HouseBlend, darkRost..)- Defines an object to which additional responsibilities can be attached.
- Decorator (CondimentDecorator) - Maintains a reference to a Component object that defines an interface that conforms to Component's interface
- ConcreteDecorator (Milk, Mocha...)- Adds Responsibilities to the component

Decorator in Java I/O classes



Decorator Pros and Cons

Pros

- A more flexible way to add or remove responsibilities at runtime.
- Rather than having a high complex class, we can define a simple class and add functionality incrementally with Decorator objects.

Cons

- Decorators can result in many small objects in our design, and overuse can be complex.
- Decorators can complicate the process of instantiating the component because you not only have to instantiate the component but wrap it in a number of decorators

Fcade Design Pattern

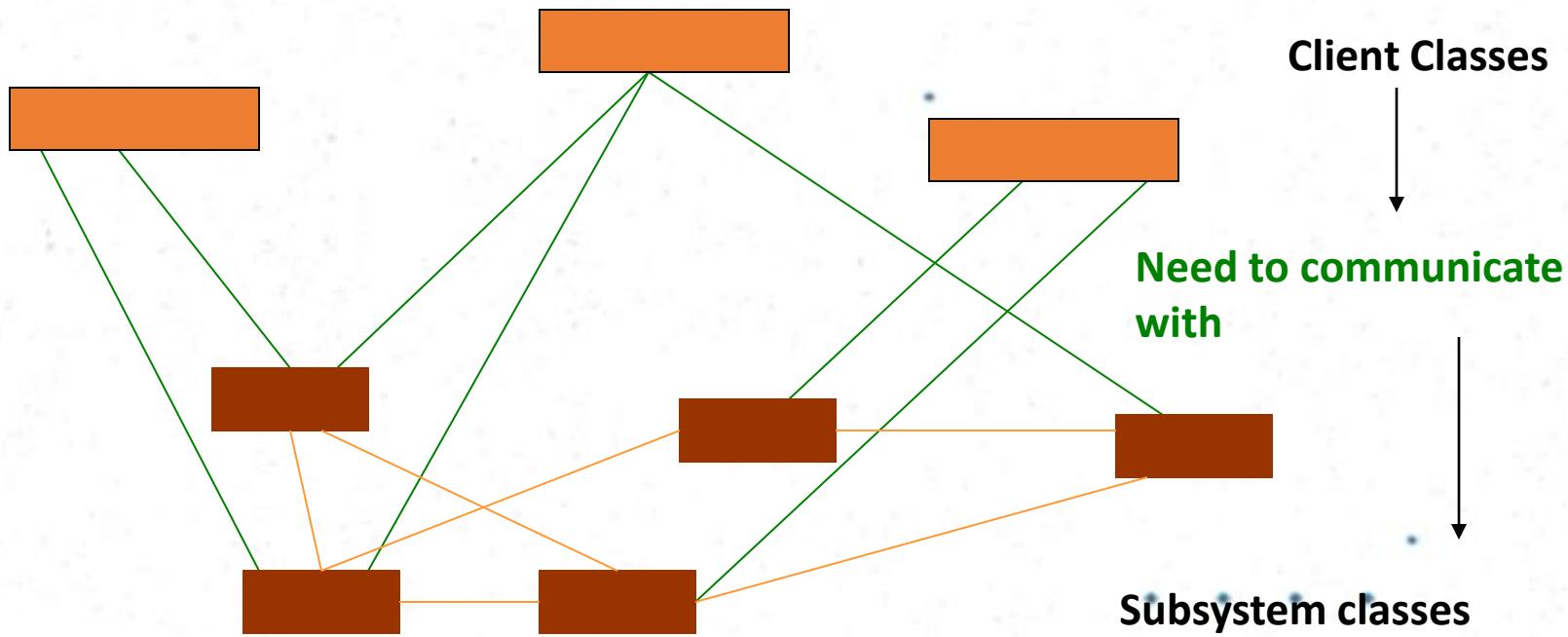
Facade Definition

Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher level interface that makes the subsystem easier to use.

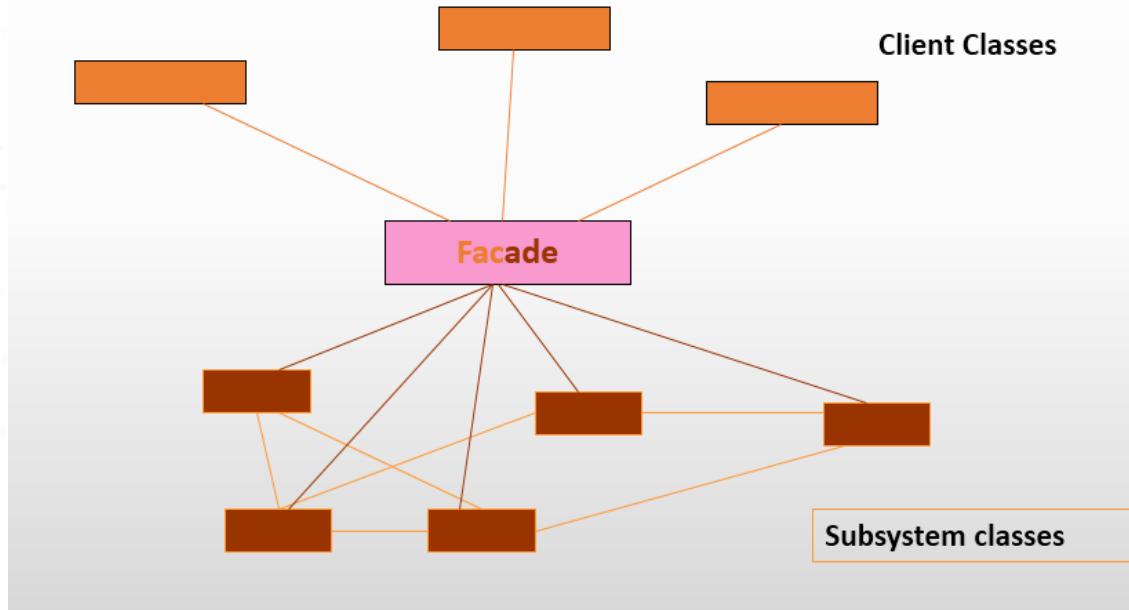
The Façade Design Pattern

- **Intent**
 - Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use
- **Motivation**
 - Simplifying system architecture by unifying related but different interfaces via a Façade object that shield this complexity from clients.
- **Clients**
 - Sending requests to the Façade, which forwards them appropriately to the subsystem components.
 - Façade may do some adaptation code.
 - Clients do not need to know, or ever use the subsystem components directly.

Facade Pattern: Problem



Facade Pattern: Solution



- Create a class that accepts many different kinds of requests, and “forwards” them to the appropriate internal class (Subsystems classes).
- You Might need to write a separate Interface class for each (all implementing the same **interface**), but the users (Client classes) of your Facade class don’t need to change

Facade

Name: Facade design pattern

Problem description:

Reduce coupling between a set of related classes and the rest of the system.

Solution:

A single **Facade** class implements a **high-level interface** for a subsystem by invoking the methods of the lower-level classes.

Example. A **Compiler** is composed of several classes: **LexicalAnalyzer**, **Parser**, **CodeGenerator**, etc. A caller, invokes only the **Compiler (Facade)** class, which invokes the contained classes.

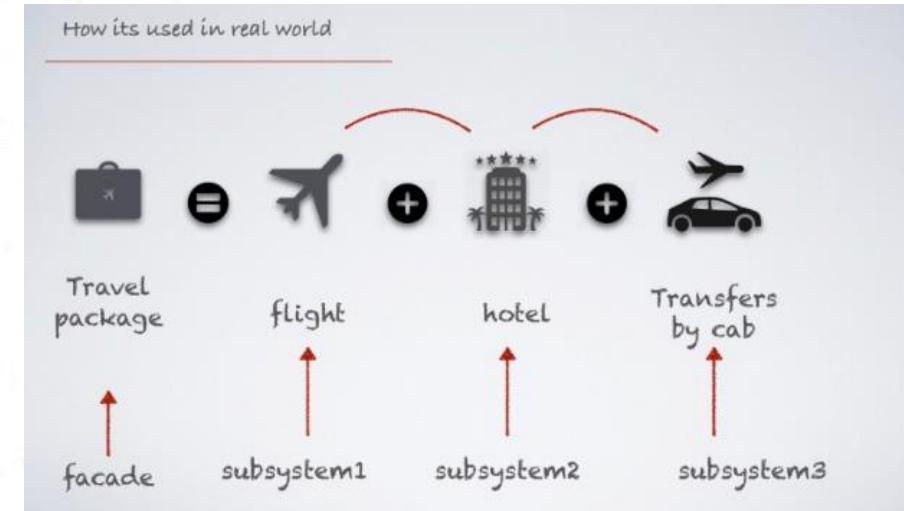
Facade Applicability

- Contrary to other patterns which decompose systems into smaller classes, Façade combines interfaces together to unified same one.
- Separate subsystems from clients via yet another unified interface to them.
- Levels architecture of a system, using Façade to separate the different subsystem layers of the application.
- . . .

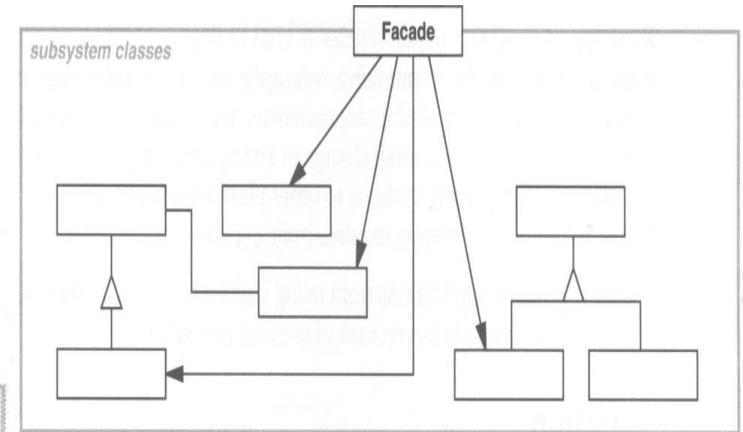
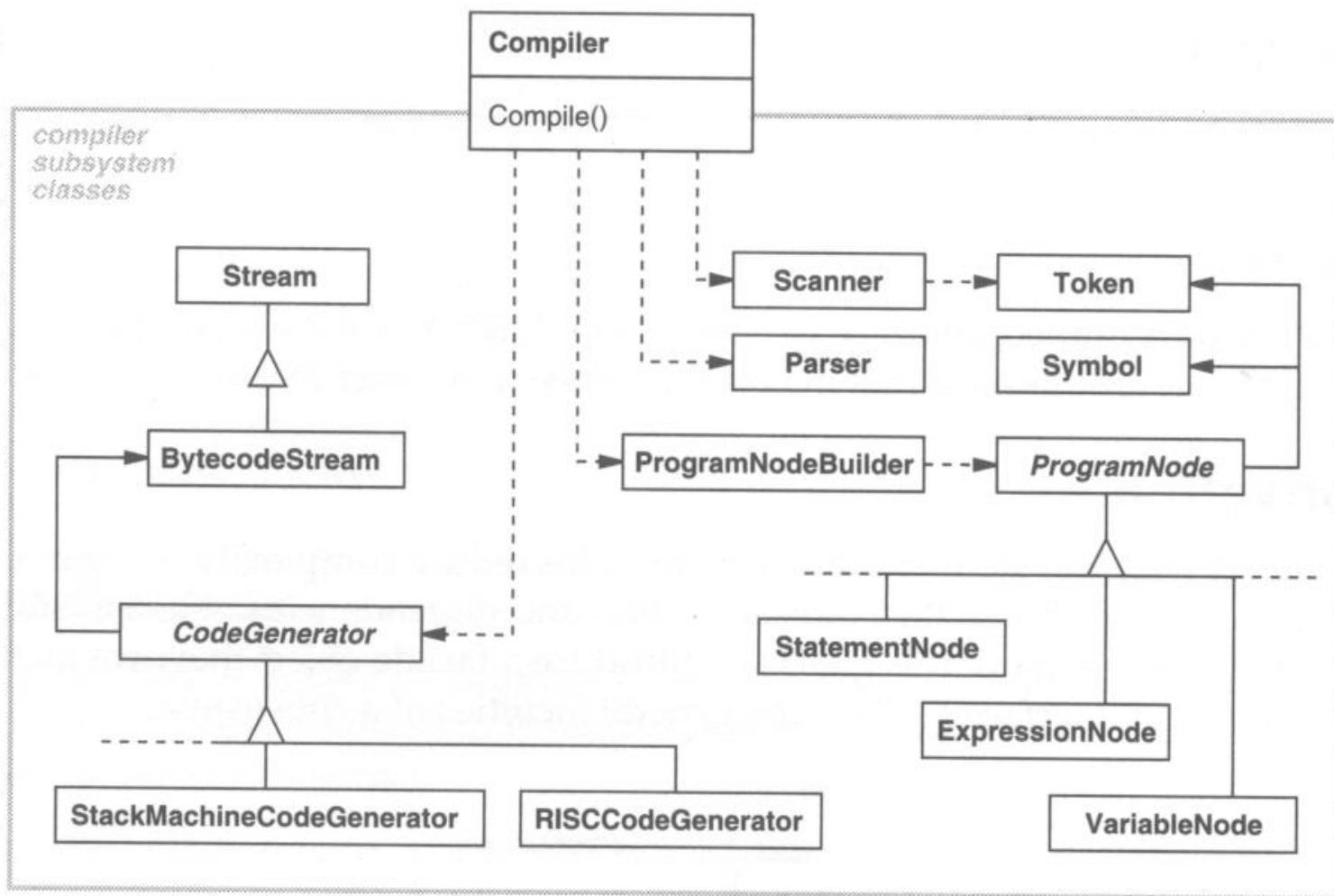
Activity -3

Let us consider an example of booking a package.

- Usually when you try to book a package , the ticket booking system interacts with many of subsystems.
- The various sub-systems may be flight , hotel and cab booking .
- In addition this may also interact with many other sub systems.
- In this case instead of client having the overhead of interacting with various other subsystems , we can introduce a facade layer which interacts will all these subsystems.
- Finally once it get the response from all the subsystems, it aggregates all these response and send the response back to the client.



Facade - Example



Facade Participants

- Façade (Compiler)
 - Knows which subsystem classes are responsible for a request
 - Delegates client requests to appropriate subsystem objects
- Subsystem classes (Scanner, Parser, ProgramNode, etc.)
 - Implement subsystem functionality
 - Handle work assignment by the Façade object
 - Have no knowledge of the Façade – I.e., no reference upward.

Facade Summary

- Ideally our facade provides a unified interface for interacting with all other subsystems .
- And also the facade layer which we have introduced acts a higher level interface which interacts with other subsystems.

Structural Patterns

Flyweight

Flyweight : Problem

- Consider writing a application like Microsoft Word. It allows user to mix all kinds of objects - sequences of characters, pictures, tables, etc.
- Given the complexity, implementers must consider modeling **every character as an Object**. This sounds right from an object-oriented perspective, however in reality, is **too expensive**. Imagine allocating memory for every character in a large document!

Definitions

- Flyweight
 - A shared object that can be used in multiple contexts simultaneously.
- Intrinsic
 - State information that is independent of the flyweights context.
Shareable Information.
- Extrinsic
 - State information that depends on the context of the flyweight and cannot be shared.

Flyweight : Example



Flyweight (Object Structural)

- **Intent:**
 - Use sharing to support large numbers of fine-grained objects efficiently.
- **Motivation:**
 - Some applications could benefit from using objects throughout their design, but a naïve implementation would be prohibitively **expensive**.

Flyweight: Background

- An image is a 2-D array of pixels, where each pixel specifies an intensity value that typically ranges from 0 (dark) to 255 (light).
 - A monochrome TV image with dimensions 768 by 576 contains over 1 million pixels.
 - A colour TV image of this size contains over 3 million pixels.
- Image processing applications typically use objects to represent images, but using an object for each pixel in an image is not the best approach. Why ??

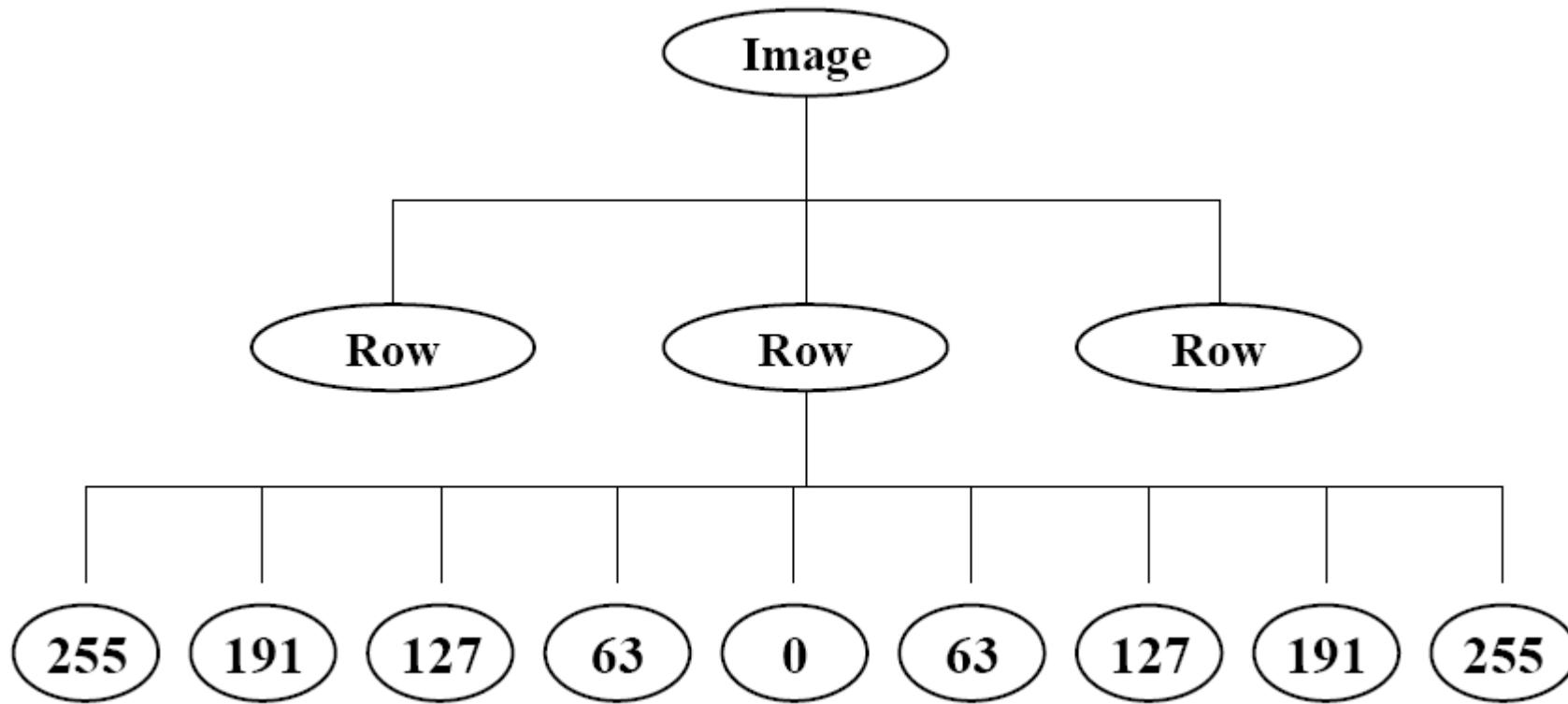


Flyweight: Motivation

- The problem with a pure OO design is its cost:
 - even small images require **many pixels objects**, which consume **lots of memory** and may incur unacceptable **run-time overheads**.
- **The flyweight pattern describes how to create a large number of objects without prohibitive cost.**
- A flyweight is a **shared object** that can be used in multiple contexts **simultaneously**:
- A flyweight acts like an ordinary object in each context . it is indistinguishable from a normal unshared object.

Flyweight: Logical Structure

- Logically, every pixel in an image is an object:



Singleton Definition – Covered in OOP

Ensure a class only has one instance and provide a global point of access to it.

Flyweight: Physical Structure

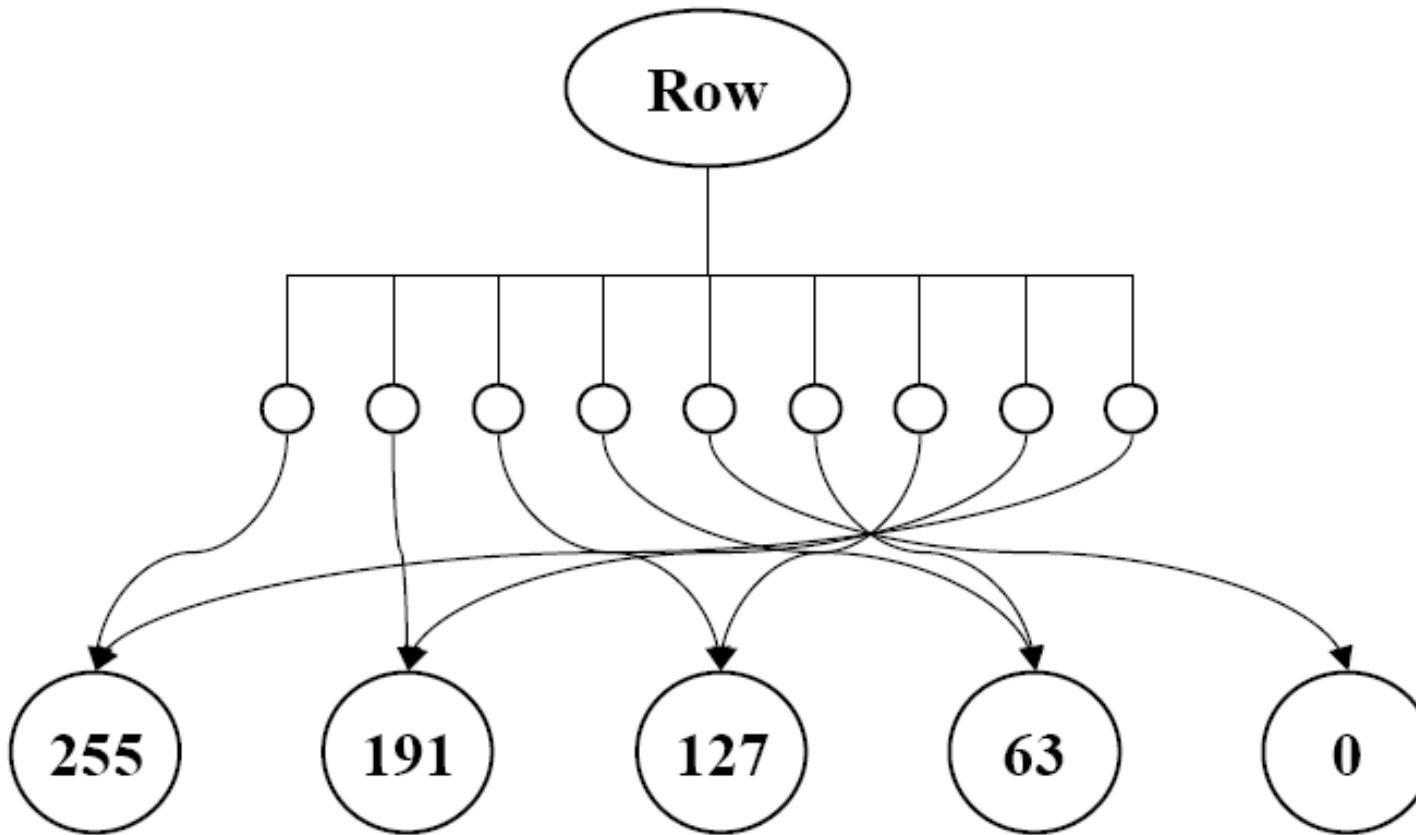
- Physically, there is one shared flyweight object representing all pixels of a given intensity:
 - a flyweight representing a pixel **only stores the intensity, not the location** of the pixel;
 - it appears in different contexts throughout the image.
- Pixels with the same characteristics all refer to the same instance in a shared pool of flyweight pixels:
 - e.g. the first and last pixels on the previous slide.

Flyweight: Physical Structure

Client:

References:

Flyweights:

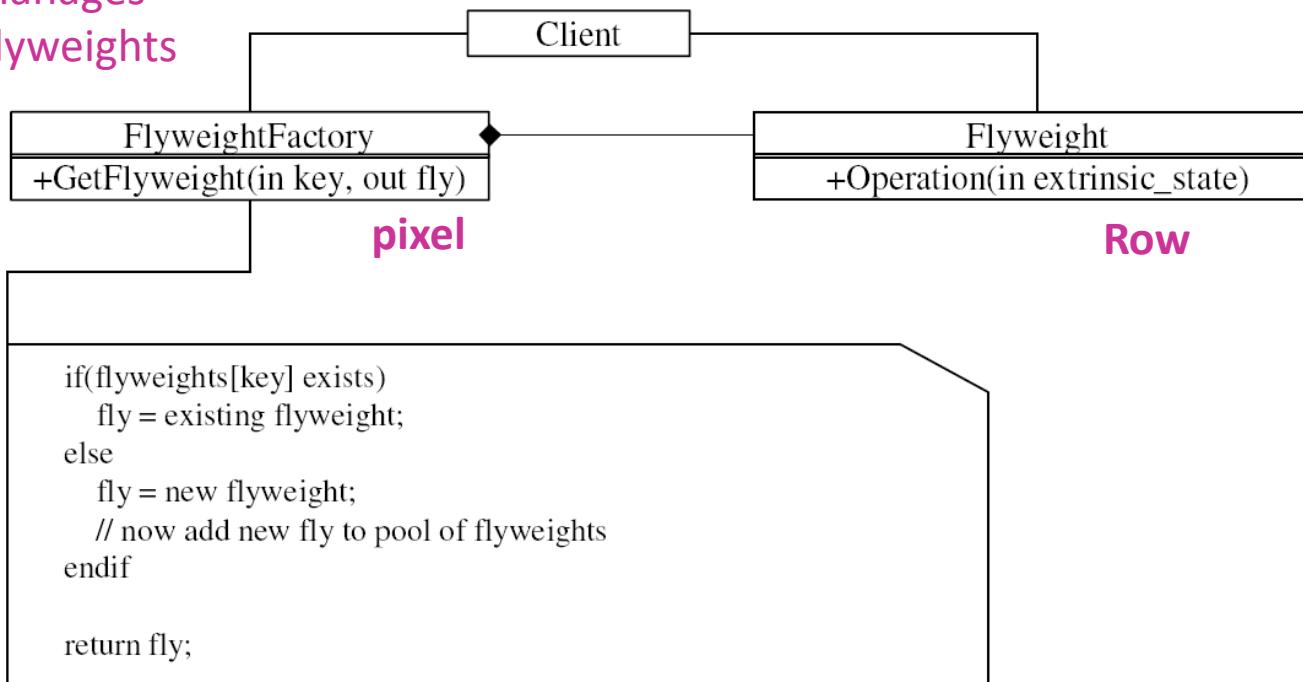


Flyweight: Physical Structure

- The Flyweight pattern relies upon a distinction between intrinsic and extrinsic state.
- Intrinsic state is stored in the flyweight; it consists of information that is independent of the flyweight's context, thereby making it sharable .
 - e.g. the intensity of the pixel.
- Extrinsic state varies with the context, and so cannot be shared; it must be passed to the flyweight whenever it is needed .
 - e.g. the row and column position of the pixel.

Flyweight: (Simplified) Structure

Creates &
manages
Flyweights



Flyweight: Participants

- **Flyweight (Pixel):**
 - Declares an interface through which flyweights can receive and act on extrinsic state.
 - Implements the Flyweight interface and adds storage for intrinsic state, if any.
- **Flyweight Factory:**
 - Creates and manages flyweight objects.
 - Ensures that flyweights are shared properly.
- **Client (ImageRow):**
 - Maintains a reference to one or more flyweights.
 - Computes or stores the extrinsic state of flyweights.

Activity

Imagine that you need to create a Car Management System to represent all of the cars in a city. You need to store the details about each car (make, model, and year) and the details about each car's ownership (owner name, tag number, last registration date).

If you are to use the Flyweight design pattern for the above system, identify the intrinsic and extrinsic states of the system.

Applicability

- Use the Flyweight when *all* of the following are true:
 - Application has a large number of objects.
 - Storage costs are high because of the large quantity of objects.
 - Most object state can be made extrinsic.
 - Many groups of objects may be replaced by relatively few once you remove their extrinsic state.
 - The application doesn't depend on object identity.

Flyweight: Collaborations

- When a client requests a flyweight object the FlyweightFactory object supplies an existing instance if possible, or creates one if necessary.
- Clients should not instantiate Flyweights directly. They must obtain Flyweights exclusively from the FlyweightFactory to ensure that duplicates are not created, and that Flyweights are shared properly.
- FlyweightFactory is related to Singleton. A FlyweightFactory for pixels would create and manage exactly 256 objects (one per intensity value).
- Flyweight state is separated into intrinsic and extrinsic state. Intrinsic state is stored in the flyweight; extrinsic state is stored or computed in the client objects.
- Clients pass the extrinsic state to the flyweight when they invoke its methods.

Flyweight: Consequences

- Flyweights may introduce run-time costs associated extrinsic state, especially if it was formerly stored as intrinsic state.
- Storage savings depend on several factors:
 - The reduction in the total number of instances from sharing.
 - The amount of intrinsic state per object (savings increase with the amount of shared state).
 - Whether extrinsic state is computed or stored.
- The more flyweights are shared, the greater the storage savings.
- The greatest savings occur when the objects use substantial quantities of both intrinsic and extrinsic state, and the extrinsic state can be computed rather than stored.

Singleton (Covered in OOP)

Singleton -Problem

- We need to make sure that only one instance of a class can be created.
- We want that instance to be easy to access anywhere in the application.
- That instance should not be resource intensive and it should be able to create only when it is needed.

Singleton – Intent and Motivation

- **Intent-** Ensure a class has only one instance (i.e. object), and provide a global point of access to it.
- **Motivation -**
 - Some classes need exactly one instance:
e.g. the Administrator boundary class should only ever have one instance because there should only be one administrator of the system.
 - Otherwise, by instantiating more than one would run into problems such as incorrect program behavior, overuse of resources or inconsistent results. To avoid them, we use Singleton.

Singleton - Implementation

- To create an object from any class, we need to call the constructor of that class.

E.g.: new MyClass();

- Can instantiate this class more than once.

```
public MyClass {  
    public MyClass() {}  
}
```

Singleton - Implementation

Q1 - How will you implement singleton design pattern in java?

Step 1 : Declare a default private constructor.

```
public MyClass {  
    private MyClass () {}  
}
```

Can we create at least one object from this class?

Singleton - Implementation

- Since the constructor is private, instantiating an object from MyClass is impossible.
- It's a chicken and egg problem.

How to solve it?



Singleton - Implementation

- Create a ‘public static’ method to return an object of type MyClass.

```
public MyClass {  
  
    private MyClass() {}  
  
    public static MyClass getInstance() {  
        return new MyClass();  
    }  
}
```

- Since it is a public method any class can access it. Since it is a static method (Class method) , can access it through class name without instantiating an object.

e.g. : MyClass.getInstance

Singleton- Implementation

- Still the above code does not satisfy the Singleton problem since it can initialize more than one object.
- To complete the code, follow the steps below:

Step 2 : Declare a private static variable to hold single instance of class.

Step 3 : Declare a public static function that returns the single instance of class.

Step 4 : Do “lazy initialization” in the accessor function.

Let's rename MyClass
to Singleton.

```
public class Singleton {  
    private static Singleton uniqueInstance;
```

We have a static
variable to hold our
one instance of the
class Singleton.

// other useful instance variables here

```
private Singleton() {}
```

Our constructor is
declared private;
only Singleton can
instantiate this class!

```
public static Singleton getInstance() {  
    if (uniqueInstance == null) {  
        uniqueInstance = new Singleton();  
    }  
    return uniqueInstance;  
}
```

The getInstance()
method gives us a way
to instantiate the class
and also to return an
instance of it.

// other useful methods here

Of course, Singleton is
a normal class; it has
other useful instance
variables and methods.

What is meant by Lazy Initialization?

Lazy initialization happens when the initialization of the **Singleton class instance** is delayed till its static `getInstance()` is called by any client program.

Singleton: Structure

Singleton

```
-static instance: Singleton&
// the singleton instance
```

```
+static getInstance(): Singleton&
// returns the unique instance

#Singleton() <<constructor>>
// the protected constructor
// make it private if there are no subclasses
```

Singleton: Participants & Collaborations

Participants:

- Singleton - creates a unique (static) instance of itself.
- Note that Singleton defines a getInstance() operation that lets clients access its unique instance.

Collaborations:

- Clients can only access the Singleton instance through the getInstance() operation.

Singleton: Implementation so far

- Create a public static method that is solely responsible for creating the single instance of the class.
 - the getInstance method can be overridden so that it assigns a subclass of Singleton to this variable.
- Should have a **private or protected** constructor, so when trying to instantiate Singleton directly cause compiler errors.
- The instance is not constructed until the first request to access it.
- If sub classing is not required then the constructor should be **private** for greater encapsulation.

Singleton: Implementation

Q2 - In which conditions the above code could lead to multiple instances?

```
public class Singleton {  
    private static Singleton uniqueInstance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance()  
    {  
        if (uniqueInstance == null) //1  
        {  
            uniqueInstance = new Singleton(); //2  
        }  
  
        return uniqueInstance; //3  
    }  
}
```

Singleton: Implementation – Multithreading

```
public static Singleton getInstance()
{
    if (uniqueInstance == null) //1
    {
        uniqueInstance = new Singleton(); //2
    }

    return uniqueInstance; //3
}
```

- If one thread enters ‘getInstance()’ method and finds the ‘uniqueInstance’ to be null at step 1.
- Then it enters the IF block.
- Before it can execute step 2 another thread enters the method and executes step 1 which will be true as the uniqueInstance is still null,
- Then it might lead to a situation (Race condition) where both the threads executes step 2 and create two instances of the Singleton class.

Singleton: Implementation – Multithreading

Q3 - How do we solve the issues in multi-threading?

1. By making getInstance() a synchronized method.

```
public class Singleton {  
    private static Singleton uniqueInstance;  
  
    // other useful instance variables here  
  
    private Singleton() {}  
  
    public static synchronized Singleton getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
}
```

By adding the synchronized keyword to getInstance(), we force every thread to wait its turn before it can enter the method. That is, no two threads may enter the method at the same time.

Singleton: Implementation – Multithreading

Q4 - What is the drawback of synchronizing the getInstance() method?

- It will decrease the performance of a multithreaded system as only single thread can access the getInstance() method at a time.
- Once we have set the uniqueInstance variable to an instance of Singleton, we have no further need to synchronize this method.
- Therefore, after the first time through, synchronization is totally unneeded overhead.

Singleton: Implementation – Multithreading

2. An eagerly created instance

```
public class Singleton {  
    private static Singleton uniqueInstance = new Singleton();  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        return uniqueInstance;  
    }  
}
```

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

We've already got an instance, so just return it.

- JVM creates the `uniqueInstance` variable when the class is loaded and before any thread accesses the static instance variable.

What is meant by Eager Initialization?

Eager initialization happens when we eagerly initialize the private static variable to hold the single instance of the class at the time of its declaration / when the class is loaded.

Singleton: Implementation – Multithreading

3. Use 'double – checked locking'

```
public class Singleton {  
    private volatile static Singleton uniqueInstance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (uniqueInstance == null) {  
            synchronized (Singleton.class) {  
                if (uniqueInstance == null) {  
                    uniqueInstance = new Singleton();  
                }  
            }  
        }  
        return uniqueInstance;  
    }  
}
```

Check for an instance and
if there isn't one, enter a
synchronized block.

Note we only synchronize
the first time through!

Once in the block, check again and
if still null, create an instance.

* The volatile keyword ensures that multiple threads
handle the uniqueInstance variable correctly when it
is being initialized to the Singleton instance.

Singleton: Known Uses & Related Patterns

- **Known Uses:**

- Singleton is useful whenever having more than one instance of a class is undesirable.
- For example:
 - Classes implementing Thread Pools
 - Database connection pools,
 - Caches
 - Objects used for logging
 - Objects that act as device drivers to devices such as printers and graphic cards etc.
- Generally use Singleton so as to prevent any class from accidentally creating multiple instances of such resource consuming classes.

Singleton: Known Uses & Related Patterns

- **Known Uses:**
 - Singleton can be generalized:
 - e.g. a 'couplet' has exactly two instances, with a single point of access to these instances;
 - a 'key' can be used to specify which instance is required.
- **Related Patterns:**
 - Many patterns can be implemented using the Singleton pattern. See Abstract Factory, Builder, and Prototype, all of which are creational patterns.

Thank you

Software Engineering (IT2020) - 2022

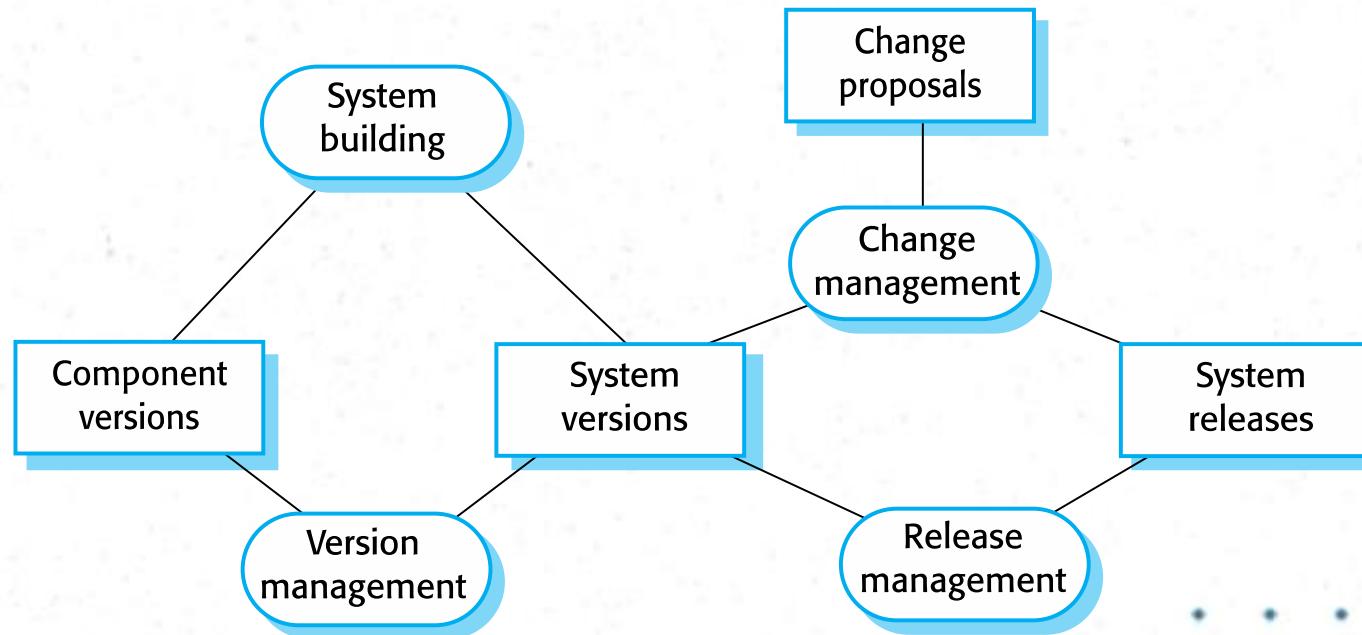
Lecture 8 - Supportive Processes

.....

what is configuration management?

- **Change management**: managed way to decide which change ideas to implement and when.
- **Version management**: keep track of multiple versions of components and ensure that changes by different developers do not disturb each other.
- **System building**: collect and assemble correct versions of required components and then compile.
- **Release management**: prepare for external releases and keep track of external releases.

Configuration management activities



Agile development and CM

- Agile development, where components and systems are changed several times per day, is impossible without using CM tools.
- The definitive versions of components are held in a shared project repository and developers copy these into their own workspace.

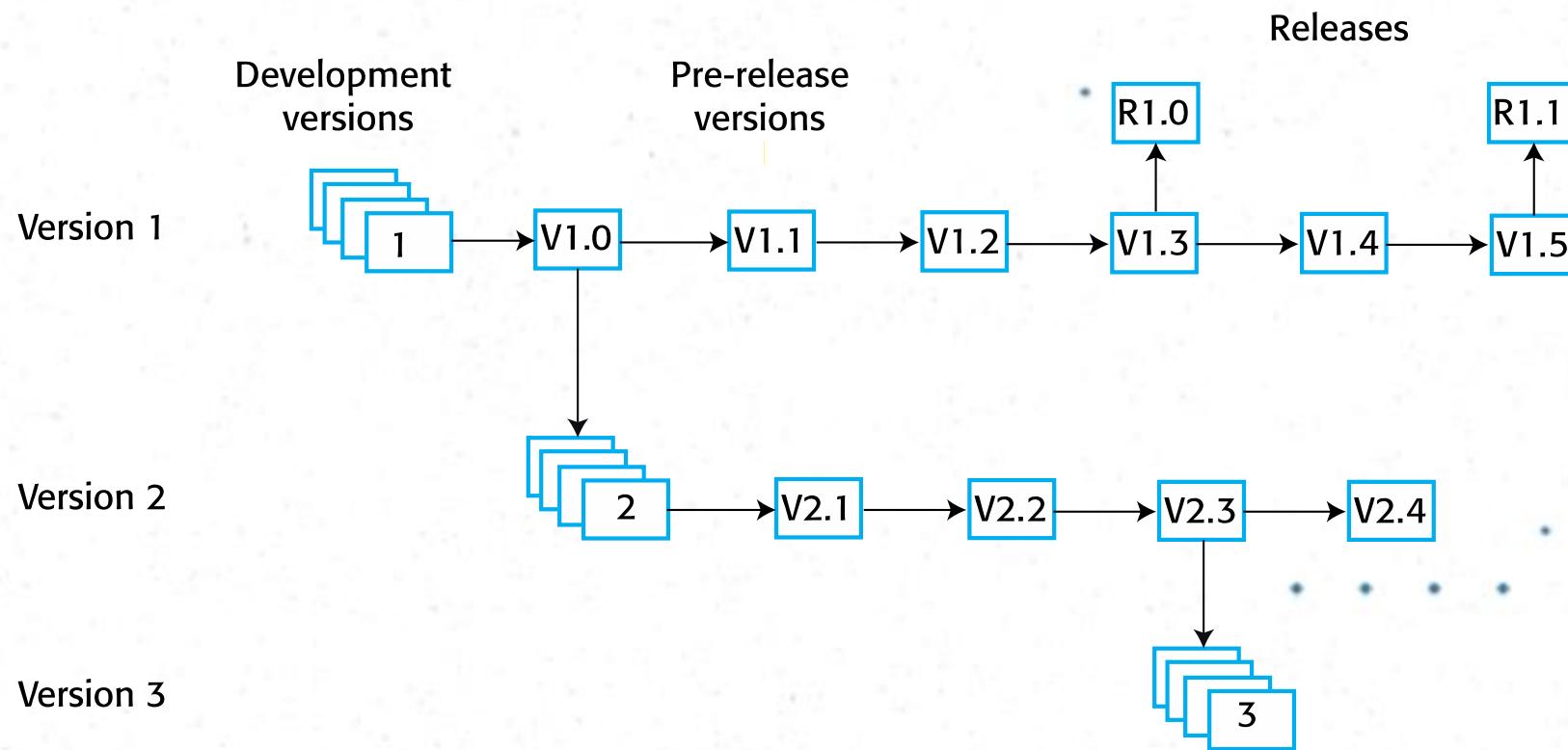
Development phases

- A development phase where the development team is responsible for managing the software configuration and new functionality is being added to the software.
- A system testing phase where a version of the system is released internally for testing.
 - No new system functionality is added. Changes made are bug fixes, performance improvements and security vulnerability repairs.
- A release phase where the software is released to customers for use.
 - New versions of the released system are developed to repair bugs and vulnerabilities and to include new features.

Multi-version systems

- For large systems, there is never just one ‘working’ version of a system.
- There are always several versions of the system at different stages of development.
- There may be several teams involved in the development of different system versions.

Multi-version system development



Version management

Version Control

- Allows different projects to use the same source files at the same time
- Isolates work that is not ready to be shared by the rest of the project
- Isolates work that should never be shared
- Allows software engineers to continue development along a branch even when a line of development is frozen
- Therefore version management can be thought of as the process of managing codelines and baselines.

Baselines

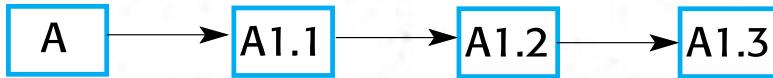
- A work product becomes a baseline only after it is reviewed and approved.
- A baseline is a milestone in software development that is marked by the delivery of one or more configuration items.
- Once a baseline is established each change request must be evaluated and verified by a formal procedure before it is processed.

Codelines and baselines

- A codeline is a sequence of versions of source code with later versions in the sequence derived from earlier versions.
- Codelines normally apply to components of systems so that there are different versions of each component.
- A baseline is a definition of a specific system.
- The baseline therefore specifies the component versions that are included in the system plus a specification of the libraries used, configuration files, etc.

Codelines and baselines

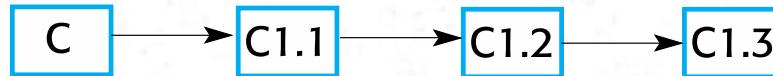
Codeline (A)



Codeline (B)



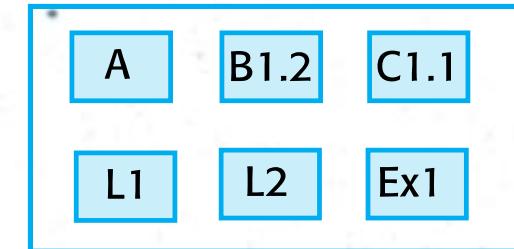
Codeline (C)



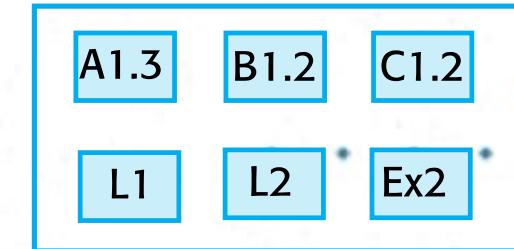
Libraries and external components



Baseline - V1



Baseline - V2



Mainline

Key features of version control systems

- Version and release identification
- Change history recording
- Support for independent development
- Project support
- Storage management

Centralized version control

- Developers check out components or directories of components from the project repository into their private workspace and work on these copies in their private workspace.
- When their changes are complete, they check-in the components back to the repository.
- If several people are working on a component at the same time, each check it out from the repository. If a component has been checked out, the VC system warns other users wanting to check out that component that it has been checked out by someone else.

Distributed version control

- A ‘master’ repository is created on a server that maintains the code produced by the development team.
- Instead of checking out the files that they need, a developer creates a clone of the project repository that is downloaded and installed on their computer.
- Developers work on the files required and maintain the new versions on their private repository on their own computer.
- When changes are done, they ‘commit’ these changes and update their private server repository. They may then ‘push’ these changes to the project repository.

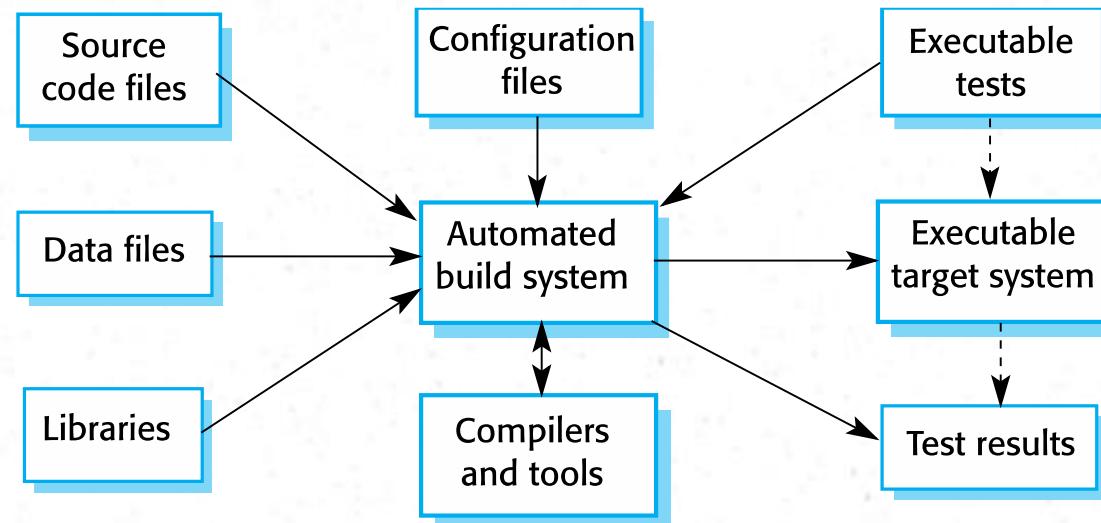
Open source development

- Distributed version control is essential for open source development.
 - Several people may be working simultaneously on the same system without any central coordination.
- As well as a private repository on their own computer, developers also maintain a public server repository to which they push new versions of components that they have changed.
 - It is then up to the open-source system ‘manager’ to decide when to pull these changes into the definitive system.

System building

System building

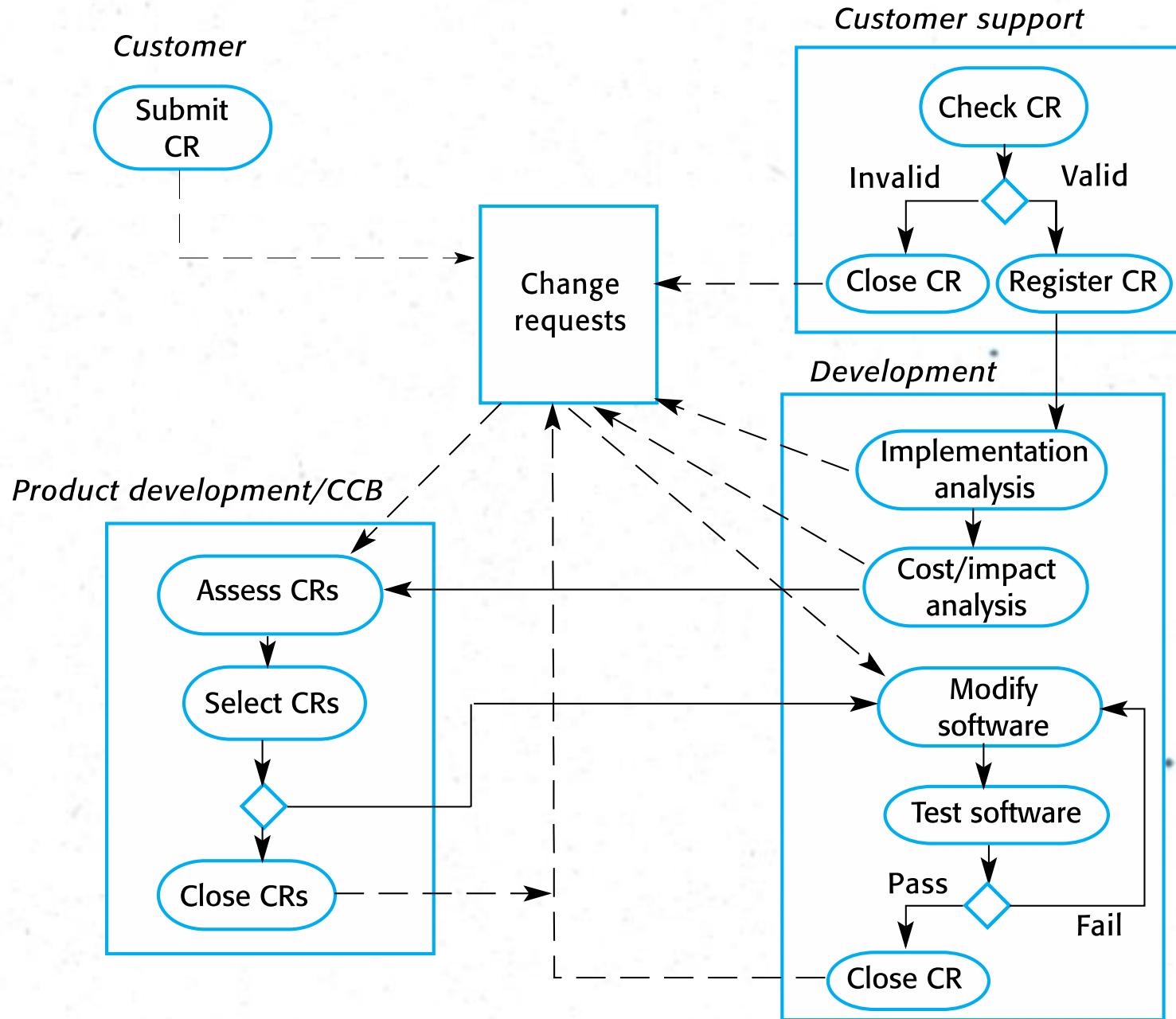
- The process of compiling and linking software components into an executable system.
- System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, etc.



Change management

Change management

- Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired and systems have to adapt to changes in their environment.
- Change management is intended to ensure that system evolution is a managed process and that priority is given to the most urgent and cost-effective changes.
- The change management process is concerned with analyzing the costs and benefits of proposed changes, approving those changes that are worthwhile and tracking which components in the system have been changed.



The change management process

A partially completed change request form (a)

Change Request Form	
Project: SICSA/AppProcessing	Number: 23/02
Change requester: I. Sommerville	Date: 20/07/12
Requested change: The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.	
Change analyzer: R. Looek	Analysis date: 25/07/12
Components affected: ApplicantListDisplay, StatusUpdater	
Associated components: StudentDatabase	

A partially completed change request form (b)

Change Request Form

Change assessment: Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

Change priority: Medium

Change implementation:

Estimated effort: 2 hours

Date to SGA app. team: 28/07/12 **CCB decision date:** 30/07/12

Decision: Accept change. Change to be implemented in Release 1.2

Change implementor: **Date of change:**

Date submitted to QA: **QA decision:**

Date submitted to CM:

Comments:

Factors in change analysis

- The consequences of not making the change
- The benefits of the change
- The number of users affected by the change
- The costs of making the change
- The product release cycle

Derivation history

```
// SICSA project (XEP 6087)
//
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
//
// Object: currentRole
// Author: R. Looek
// Creation date: 13/11/2012
//
// © St Andrews University 2012
//
// Modification history
// Version Modifier Date Change Reason
// 1.0      J. Jones  11/11/2009 Add header Submitted to CM
// 1.1      R. Looek  13/11/2012 New field Change req. R07/02
```

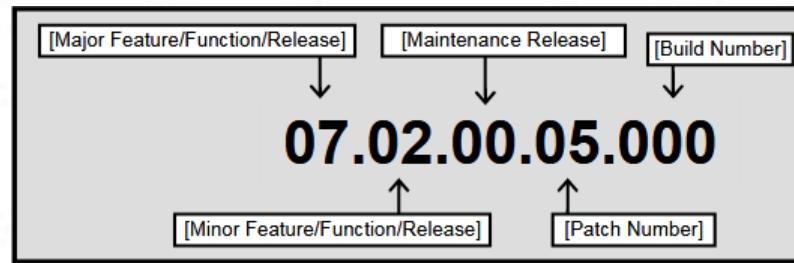
Change management and agile methods

- In some agile methods, customers are directly involved in change management.
- They propose a change to the requirements and work with the team to assess its impact and decide whether the change should take priority over the features planned for the next increment of the system.
- Changes to improve the software improvement are decided by the programmers working on the system.
- Refactoring, where the software is continually improved, is not seen as an overhead but as a necessary part of the development process.

Release management

Release management

- A system release is a version of a software system that is distributed to customers.
- Types of releases
 - Major releases - deliver significant new functionality
 - Minor releases - repair bugs and fix customer problems
- Release Numbering
 - Several Notations
 - *major.minor[.build[.revision]]*
 - *major.minor[.maintenance[.build]]*



Release components

- Executable Code
- configuration files
- Data files, such as files of error messages, that are needed for successful system operation;
- Installation program
- Electronic and paper documentation describing the system;
- Packaging and associated publicity that have been designed for that release.

Factors influencing system release planning

Factor	Description
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. Minor system faults may be repaired by issuing patches (usually distributed over the Internet) that can be applied to the current release of the system.

Release creation

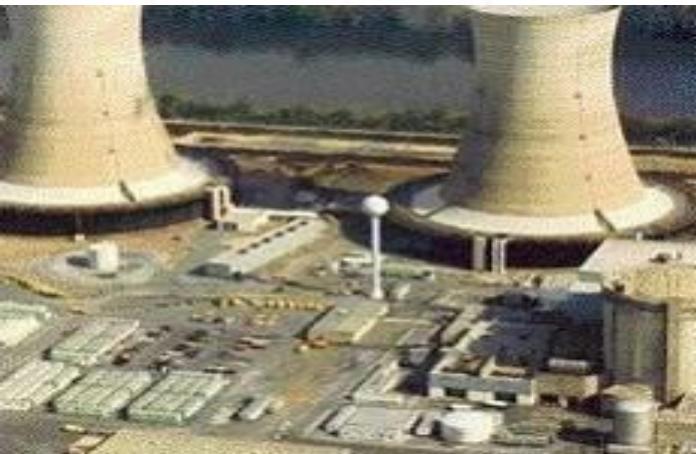
- Identify items
 - The executable code of the programs and all associated data files must be identified in the version control system.
- Set Configurations
 - Configuration descriptions may have to be written for different hardware and operating systems.
- Configuration Guidelines
 - Update instructions may have to be written for customers who need to configure their own systems.
- Installation Scripts
 - Scripts for the installation program may have to be written.
- Documentation on Web
 - Web pages have to be created describing the release, with links to system documentation.
- Create the release
 - When all information is available, an executable master image of the software must be prepared and handed over for distribution to customers or sales outlets.

Release planning

- As well as the technical work involved in creating a release distribution, advertising and publicity material have to be prepared and marketing strategies put in place to convince customers to buy the new release of the system.
- Release timing
 - If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it.
 - If system releases are too infrequent, market share may be lost as customers move to alternative systems.

Software as a service (SaaS)

Traditional Software



Build Your Own

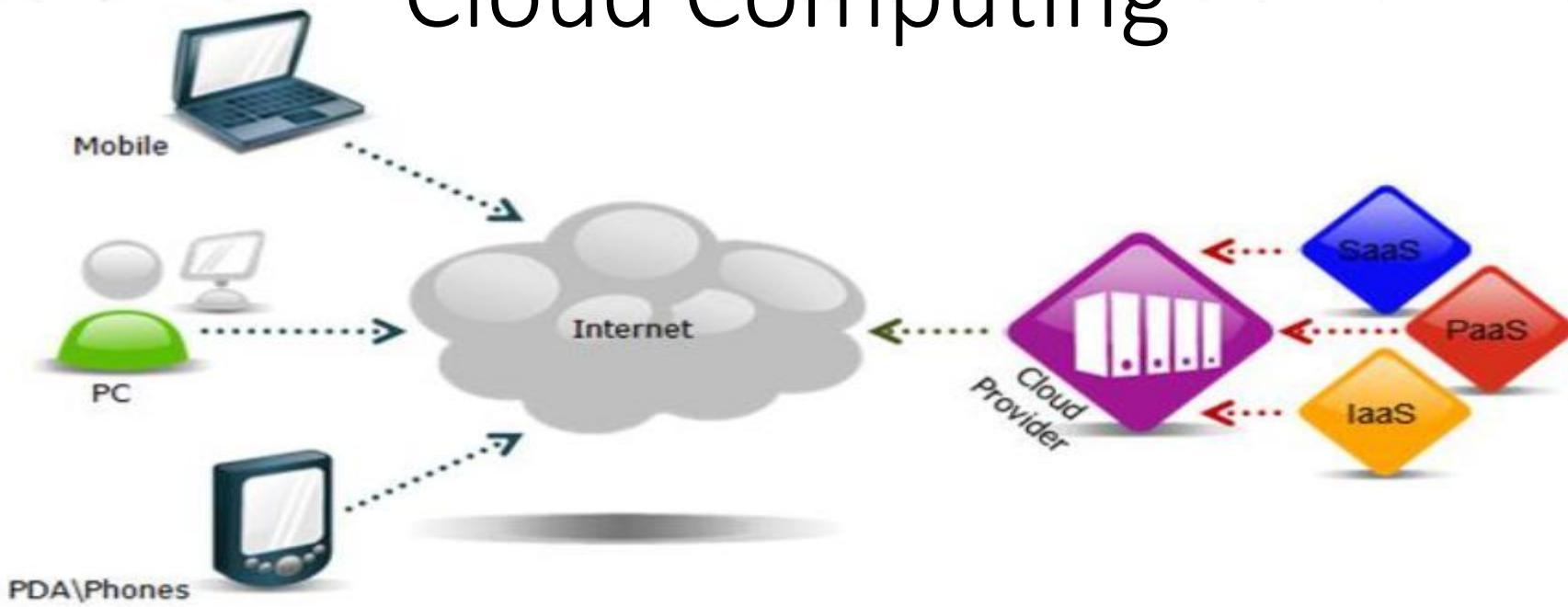
On-Demand Utility



Plug In, Subscribe

Pay-per-Use

Cloud Computing



Traditional packaged software

- Designed for customers to install, manage and maintain.
- Architect solutions to be run by an individual company in a dedicated instantiation of the software

Software as a service

- Designed from the outset up for delivery as Internet-based services
- Designed to run thousands of different customers on a single code

Traditional packaged software

- Infrequent, major upgrades every 18-24 months, sold individually to each installed base customer.
- Version control
- Upgrade fee
- Streamlined, repeatable functionality via Web services, open APIs and standard connectors

Software as a service

- Frequent, "digestible" upgrades every 3-6 months to minimize customer disruption and enhance satisfaction.
- Fixing a problem for one customer fixes it for everyone
- May use open APIs and Web services to facilitate integration, but each customer must typically pay for one-off integration work.

Software as a service

- Delivering software as a service (SaaS) reduces the problems of release management.
- It simplifies both release management and system installation for customers.
- The software developer is responsible for replacing the existing release of a system with a new release and this is made available to all customers at the same time.

Chapter 25 Configuration management -
Sommerville

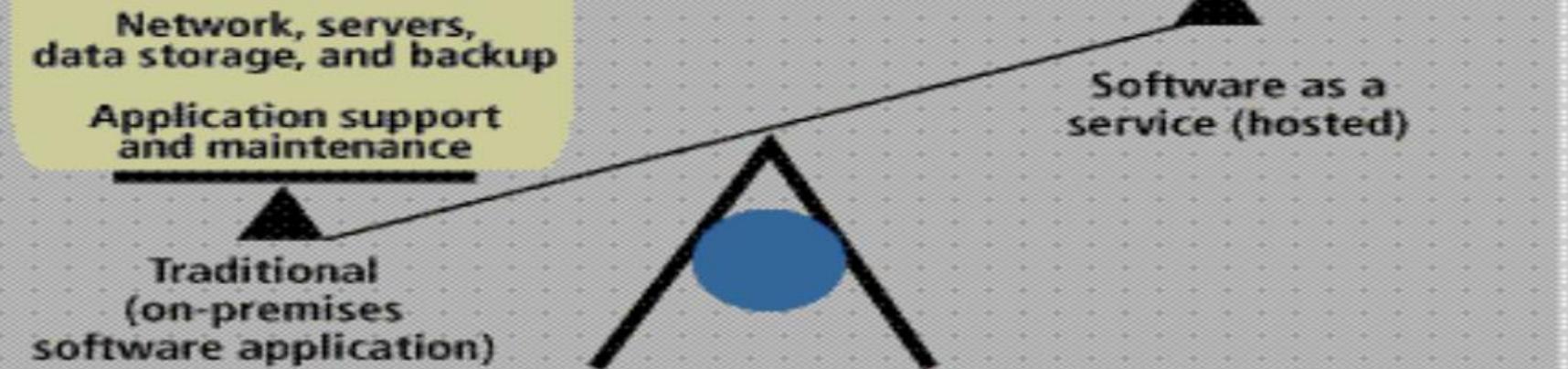
Software as a service

Balanced View

When weighing software as a service against a traditional application, take all costs—direct and indirect—into account

- Licenses
- Customization
- Implementation
- IT support personnel
- Training for custom application
- Network, servers, data storage, and backup
- Application support and maintenance

- Subscription fees
- Configuration fees
- Internet access
- End-user support usage fees
- Training on a standard application



Factors influencing system release planning

Factor	Description
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. Minor system faults may be repaired by issuing patches (usually distributed over the Internet) that can be applied to the current release of the system.

Software as a service

- Delivering software as a service (SaaS) reduces the problems of release management.
- It simplifies both release management and system installation for customers.
- The software developer is responsible for replacing the existing release of a system with a new release and this is made available to all customers at the same time.

CM terminology

Term	Explanation
Baseline	A baseline is a collection of component versions that make up a system. Baselines are controlled, which means that the versions of the components making up the system cannot be changed. This means that it is always possible to recreate a baseline from its constituent components.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Codeline	A codeline is a set of versions of a software component and other configuration items on which that component depends.
Configuration (version) control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Configuration item or software configuration item (SCI)	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. There are often different versions of a configuration item. Configuration items have a unique name.
Mainline	A sequence of baselines representing different versions of a system.

CM terminology

Term	Explanation
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Repository	A shared database of versions of software components and meta-information about changes to these components.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions always have a unique identifier.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.

Incident management

Incident Management

Why do people contact IT support?

Something Doesn't Work

Information

Want Something



Why not just help desk?

- Traditional IT help desk environments can't respond to rapidly expanding user expectation and are not designed for the complete scale of modern IT.
- They rely on standalone tools and manual activities, making it incredibly difficult to engage effectively with users.
- Incident management modernizes and transforms your IT help deck, dramatically increasing efficiency while improving service levels.
- Engage online and real-time visibility of help desk performance.

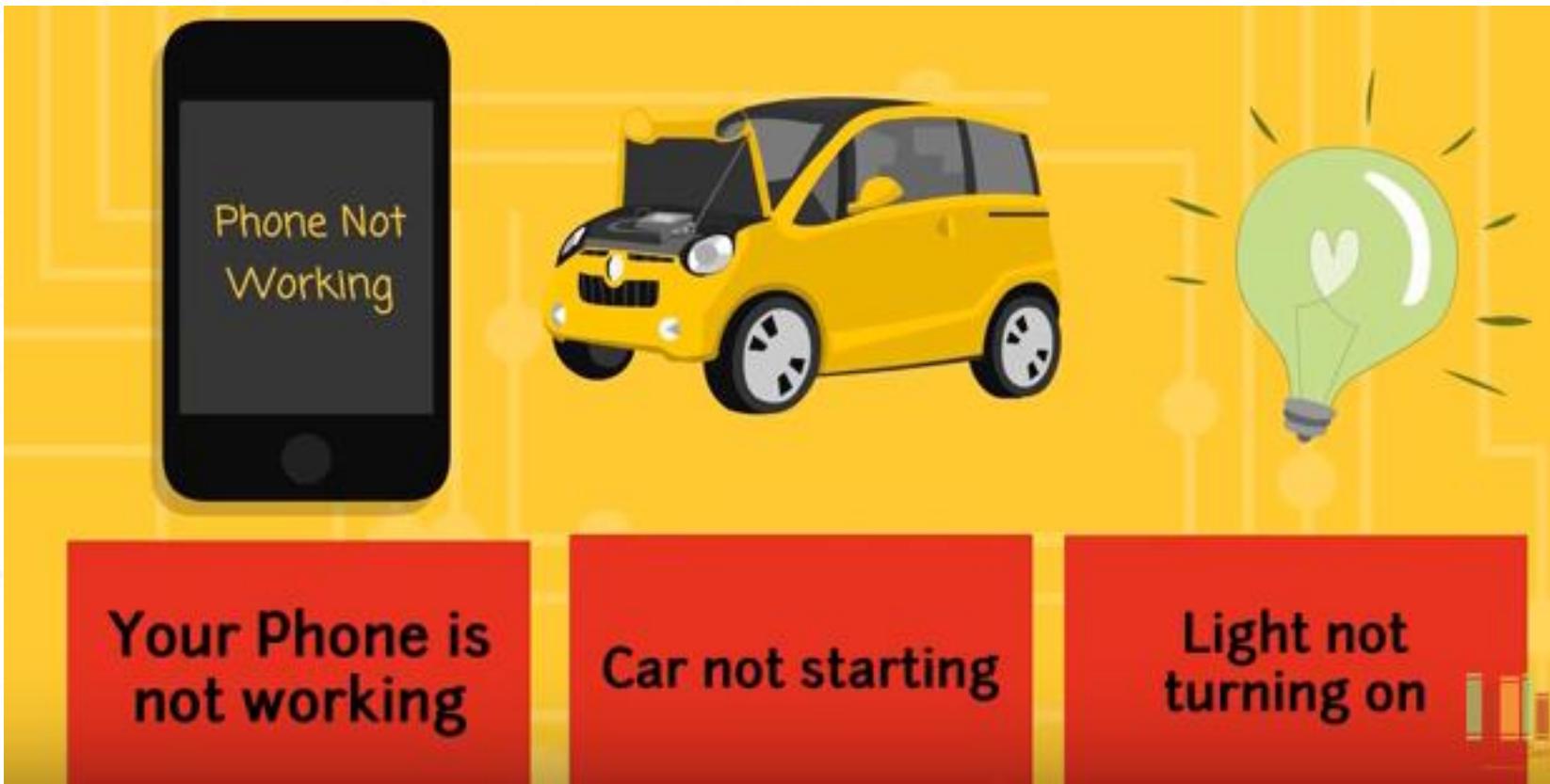
Incident Management

- **Incident** is an unplanned interruption to an IT service or reduction in the quality of an IT service.
- **Incident Management** is the process for dealing with all incidents; this can include failures, questions or queries reported by the users (usually via a telephone call to the Service Desk), by technical staff, or automatically detected and reported by event monitoring tools.

Examples of Incident



Cont..



Primary goal

- The primary goal of the Incident Management process is to **restore normal service operation** as quickly as possible, after an incident has occurred to that service, and minimize the adverse impact on business operations.

Goals of incident Management

- Restore the service as quickly as possible
- Minimum disruption to users' work
- Management of an incident during its entire lifecycle
- Support of operational activities

The Incident Process

- Incident identification
- Incident logging
- Incident categorization
- Incident prioritization
- Incident response
 - Initial diagnosis
 - Incident escalation
 - Investigation and diagnosis
 - Resolution and recovery
 - Incident closure

Incident identification

- The first step in the life of an incident is incident identification.
- Incidents come from users in whatever forms the organization allows.
- Sources of incident reporting include walk-ups, self-service, phone calls, emails, support chats, and automated notices, such as network monitoring software or system scanning utilities.

Cont..

- The service desk then decides if the issue is truly an incident or if it's a request.
- Requests are categorized and handled differently than incidents, and they fall under request fulfillment.

Incident logging

- Once identified as an incident, the service desk logs the incident as a ticket.
- The ticket should include information, such as the user's name and contact information, the incident description, and the date and time of the incident report

Logging

- Self Service Portal
- Email
- Telephone
- Walk ins

I 1303 040 Chair is Broken

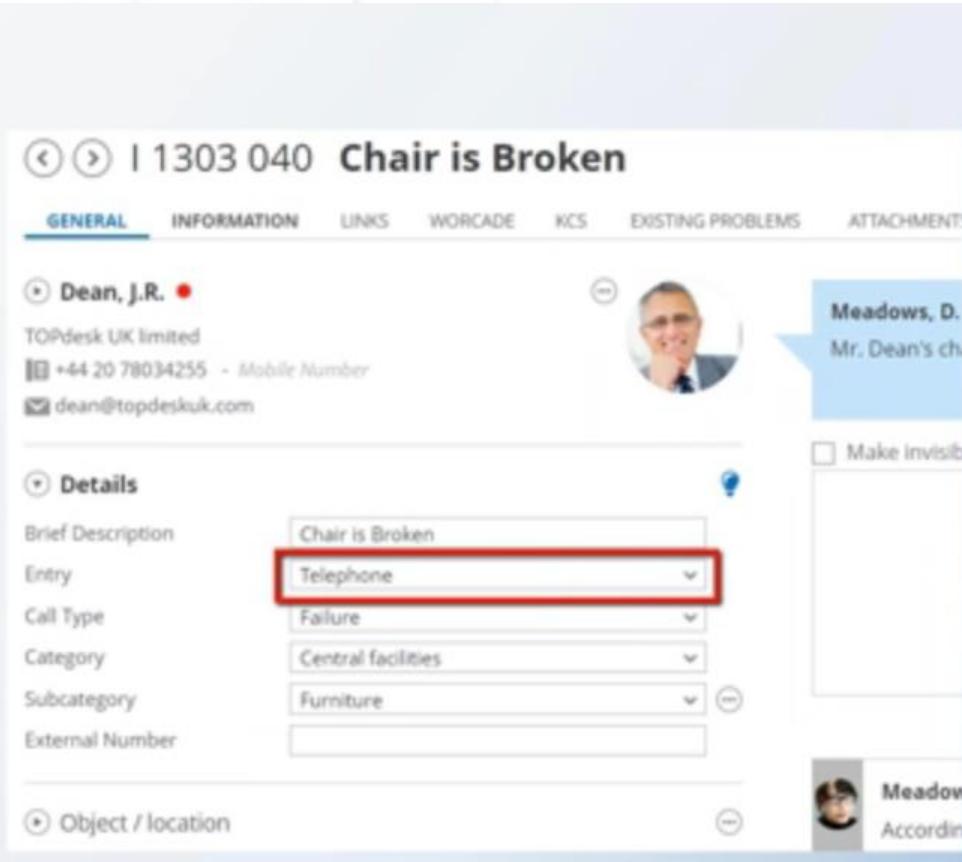
GENERAL INFORMATION LINKS WORCADE KCS EXISTING PROBLEMS ATTACHMENTS

Dean, J.R. •
TOPdesk UK limited
+44 20 78034255 - Mobile Number
dean@topdeskuk.com

Details

Brief Description	Chair is Broken
Entry	Telephone
Call Type	Failure
Category	Central facilities
Subcategory	Furniture
External Number	

Object / location



Incident categorization

- Categorization involves assigning a category and at least one subcategory to the incident.
- This action serves several purposes. First, it allows the service desk to sort and model incidents based on their categories and subcategories.
- Second, it allows some issues to be automatically prioritized. For example, an incident might be categorized as “network” with a sub-category of “network outage”.

Categorisation

- Recognise, differentiate and understand

Details

Brief Description	Chair is Broken
Entry	Telephone
Call Type	Failure
Category	Hardware
Subcategory	Laptop
External Number	

Incident prioritization

- An incident's priority is determined by its impact on users and on the business and its urgency.
- Urgency is how quickly a resolution is required; impact is the measure of the extent of potential damage the incident may cause.
 - **Low-priority incidents**
 - **Medium-priority incidents**
 - **High-priority incidents.**

Prioritisation

- Priority matrix
- Impact
- Urgency

Planning

Impact	Organisation
Urgency	Critical
Priority	P1
Duration	1 hour
Target Date	April 28, 2017
On Hold	[checkbox]

Priority matrix

Impact	Organisation	Location	Department	Person
Urgency				
Critical	P1	P2	P2	P3
Normal	P2	P3	P2	P3
Low	P3	P4	P4	P5

Incident response

- **Initial diagnosis:** This occurs when the user describes his or her problem and answers troubleshooting questions.
- **Incident escalation:** This happens when an incident requires advanced support, such as sending an on-site technician or assistance from certified support staff
- **Investigation and diagnosis:** These processes take place during troubleshooting when the initial incident hypothesis is confirmed as being correct. Once the incident is diagnosed, staff can apply a solution, such as changing software settings, applying a software patch, or ordering new hardware.

Cont..

- **Resolution and recovery:** This is when the service desk confirms that the user's service has been restored to the required SLA level.
- **Incident closure:** At this point, the incident is considered closed and the incident process ends.

Incident response

Initial Diagnosis

- Can I fix this straight away?

Closing a call

- Completed
- Closed

Processing

Operator Group	Facilities management	...
Operator	Bauer, F.	...
Status	▼	

Processing

Operator Group	Facilities management	...
Operator	Facilities management	...
Status	Solved	▼
Completed	<input checked="" type="checkbox"/> January 4, 2017	1:57 PM
Closed	<input checked="" type="checkbox"/> January 6, 2017	4:16 PM

Incident Template

Template
Bond Trading Access Denied

Name: Bond Trading Access Denied Application: Global

Table: Incident [incident] User: System Administrator

Active: Group:

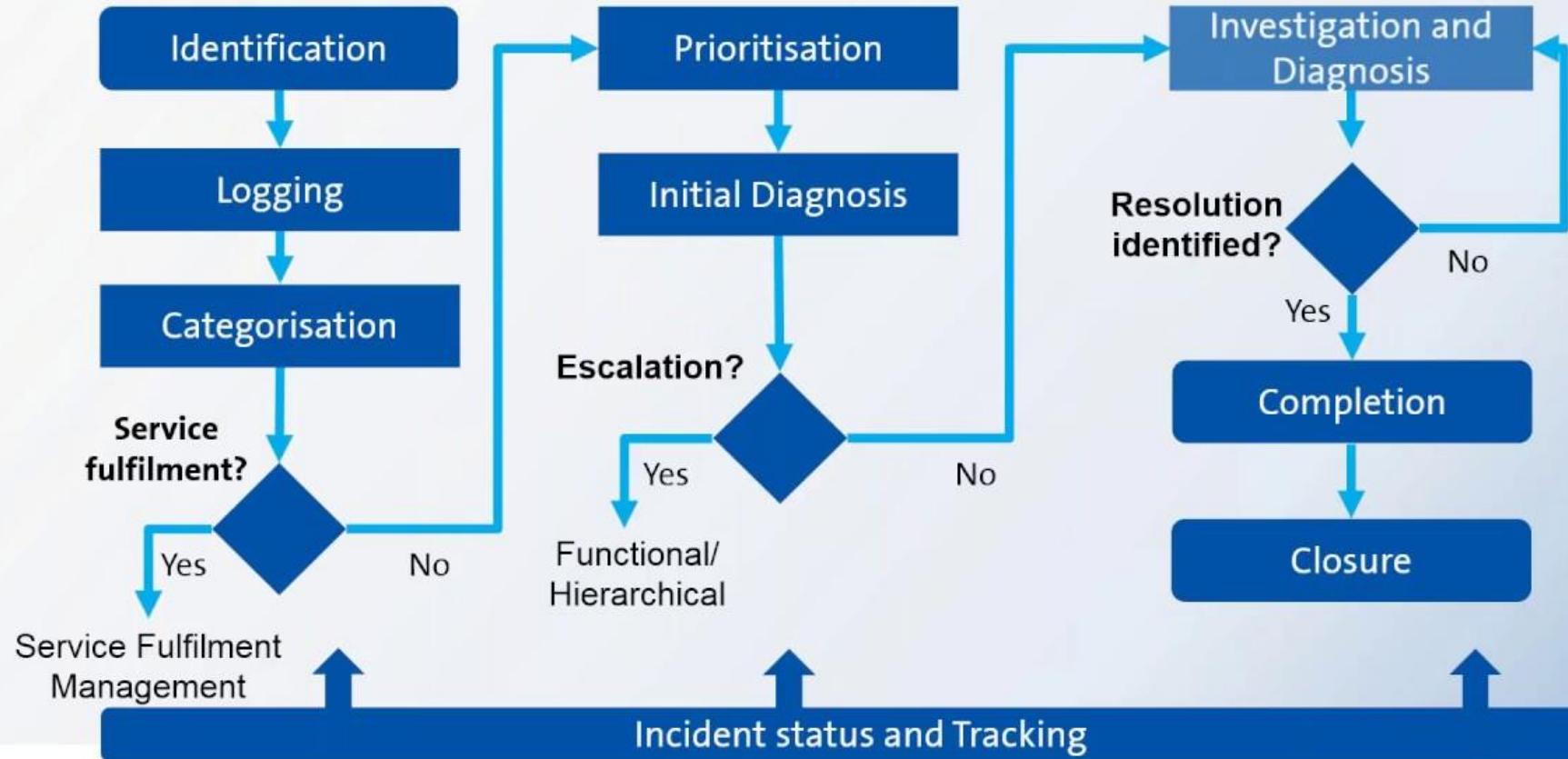
Global:

Short description: Bond Trading Access Denied

Template:

Category	Inquiry / Help	X
Description	The user was denied access to the Bond Trading application.	X
Impact	2 - Medium	X
Urgency	3 - Low	X
Configuration item	Bond Trading	X

Incident Management Process



Service Management Simplified | **TCPdesk**

Incident statuses

Incident statuses mirror the incident process and include:

- New
- Assigned
- In progress
- On hold or pending
- Resolved
- Closed

Benefits

- Reduced business impact of incident by timely resolution
- Proactive identification of possible enhancements
- Improved monitoring
- Better staff utilization
- Better customer satisfaction

References

- <http://www.bmcsoftware.com.au/guides/itil-incident-management.html>
- <https://www.slideshare.net/agnihotry/itil-incident-managementfor-beginners>