SYSTEM MODELING BASICS

Author(s): Rajashekar Tuppada
Date written (MM/DD/YY): 10/20/23
Target readers: Technical People
Keywords: UML Design

TITLE	System Modeling Basics		
REVISION HISTORY			
Revision Number	**Date**	**Author(s)**	**Description**
0.1	21 Mar.2024	Rajashekar Tuppada	Initial Draft
0.2	28 Mar 2024	Rajashekar Tuppada	Updated Version
0.3	14 Apr 2024	Rajashekar Tuppada	Corrections Suggested
1.0	21 Apr 2024	Rajashekar Tuppada	Final Version

Table of Contents

1 Guidelines to Author

- Please keep the content short and concise with concentration on project experience/learning.
- Text within '[ ]' can be removed.
- Text within '< >' is to be replaced by the correct value.
- Notes to the Author are in Blue within '[ ]'.
- Document text need to be in "Verdana, 10 pt" for consistency purposes.
- The watermark need to be changed to restricted/ confidential/ highly confidential based on the confidentiality class of the content presented in the document.
- Please retain the copyright notice and the footer as it is.

1.1 Purpose

The purpose of this document is to provide the basic concepts across the System Modeling

2 Introduction

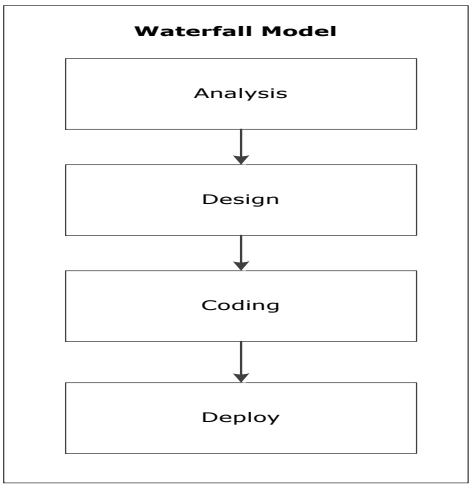
The purpose of this document is to provide the basic concepts across the System Modeling

3 System Business Problem

System is a combination of hardware & software that provides a solution for the business problem. The entities involved in solving a business problem are

Client: Entity which has the defined problem to be solved.
Analyst: Entity who documents the client's problem and relays it to developers.
Developers: Entity who builds software, test and deploy on computer hardware.

The Generic approach for the Solution to the business problem follows the Software Engineering Model (Waterfall Model) which follows the below workflow.



Analysis: Involves the understanding of Business requirements and devising the approach to the Solution.

Design: Providing the Solution approach providing the artifacts to encounter the business requirements**.**

Coding: Developing the defined solution as per the design.

Deploy: Consolidating the code and dropping it into the runnable environment.

In General the Translation happens without the interaction of different phases.

To build a stronger system with solid understanding of the solution and the interaction b/n the different phases is obvious.

4 Rapid Application Development (RAD)

Rapid application development defines the elaborate process involved in providing the solution to the business problem

4.1 Requirements Gathering

Requirement gathering focuses on understanding of the client's business requirement thoroughly before implementing the s/m.

The steps involved in this process are

Discover business process *(Activity diagram):* Analysts interact with the client to gain the better understanding and discover the relevant process by steps.

Performance domain analysis *(High level class diagram):* Analysts interview the client to understand the different entities in the system. The role of object modeler is to list the nouns & verbs for the system which are then the attributes and operations of the classes.

Identify co-operating systems *(Deployment diagram):* Dependency of the new system with existing systems and additional systems (if any). Systems engineer to define the deployment strategy.

Discover system requirements *(Package diagram):* Object Modeler refines the class diagram and defines the process of packaging the system into more rational groups along with the architecture to accomplish client's business requirements.

The end result of this process to deliver the client with the comprehensive high level solution to resolve the Business Requirements.

4.2 Analysis

Analysis is the process of getting in-depth understanding of the requirements of the system providing the clear directions of the system implementation.

Analysis provides the holistic system view envisioning below steps

Identify the actors *(Use case diagram):* Depicts the actors, who initiate each use case and the actors who are benefited with these actions. Also, Analyze the sequence of steps (text description) in each use case diagram *(UCD)*.

* *Actor: System or Person who interacts with the system.*

Refine the class diagram *(Refined class diagram):* Defining the associations, abstract classes, multiplicities, generalizations and aggregations.

Analyze changes of state in objects *(State diagram):* Object modeler changes/refines the model by showing the changes of state wherever necessary.

Define the interactions among objects *(Sequence and Collaboration diagram):* With defined use cases and refined class diagram, this step determines how the objects interact with each other.

Analyze, Integration with co-operating systems*:* Specifications to integrate the co-operating systems.

- Communication type between systems
- Network Architecture
- Database Access
- Protocol involved

At this stage define the deployment diagram along with the Data models.

The end result of this process to deliver the client with the high level object diagrams and the interactions between them.

4.3 Design

Design is the process to elaborate the analysis further to arrive at the realistic system implementation details.

Design & analysis have an appreciable hand shaking until the design is frozen

Develop & refine object diagrams *(Activity diagram):* Programmers generate the object diagrams as required from the class diagrams depicting each operation.

Develop component diagrams *(Component diagram):* Programmers visualize the components and depicts the dependencies amongst these components.

Deployment plan *(Deployment diagram): Strategic* plan for components deployment and integrating them with the co-operating systems.

Design & prototype user interface *(Screen shots of the screen prototypes):* GUI analyst works with the users to develop prototypes of screen that corresponds to group of use cases.

Design tests *(Test scripts):* Developer and/or test specialist from outside the development team uses the use case diagram to develop test scripts for automated test tools.

Documentation *(Document structure):* Documentation specialist in co-ordination with designers arrives at high level document architecture.

The end result of this process to deliver the client with the Low level design details of the objects along with their interactions.

4.4 Development

Development is the process to elaborate the analysis further to arrive at the realistic system implementation details.

Development of the System follows the below steps

Cons struct code *(Code):* Programmers construct the system in line with the available class, object, activity and component diagrams.

Test code *(Test results):* Test the code to meet all the Client Business Requirements and capture the test results. This stage is classified as a unit/system testing.

Construct user interfaces, connect to code & test *(Functioning system):* GUI specialist in co-ordination with developer integrates the UI prototypes (approved) and connects to the code.

Complete documentation *(System documentation):* Documentation experts in co-ordination with programmers complete documentation.

The end result of this process to complete executable Code along with user interfaces to the system to be deployed into the runnable environment.

4.5 Deployment

Deployment is the process of dropping the code with appropriate configurations into the Hardware along with Integrations with co-operative system.

Activities involved in Deployment are as follows

Plan for backup & recovery*:* Systems Engineer plans for system backup & recovery in case of unexpected system crash (crash recovery plan)

Installation*:* Process of packaging and deploying into the environment from where the Code can be executed.

Test the installed System*:* Final stage of testing to ensure system works meeting the Client Business Requirements.

The end result of this process to deploy the code base into the runnable environment and made available for the end user.

5 UML Components

The purpose of the UML components and/or diagram is to present multiple views of the system and the set of multiple views is called a model. UML describes what the system is supposed to do rather than how to implement it.

UML composes of nine basic diagrams

1. ***Class diagram**:* Class is a category/group of things that have similar attributes and common behaviors. Class diagram depicts the template to define the class.
2. ***Object diagram**:* Object is an instance of the class which has specific attributes and behavior. Object diagram depicts the template to define the object.
3. ***Use case diagram (USD)**:* USD is the description of the system behaviour from user's stand point. It's a proven technique for gathering system requirements from user point of view. An actor is an entity that initiates the use case, which can either be a person/system.
4. ***State diagram**:* An object assumes certain state at any given moment of time. State diagram indicate/represent these states of an object. The state diagram starts with symbols to represent 'Start state' and 'End state'.

   E.g.: - person (object)

   States - newborn, infant child, adolescent, teenager or adult.
5. ***Sequence diagram**:* Class diagram and UCD will provide (represent) a static information. In a functioning s/m, however, objects interact with one another and these interactions occur over time. SD will show the time based dynamics of interaction.
6. ***Activity diagram**:* The activities that occur within a use case/within an object behavior typically occur in sequence. This sequence is reps. with activity diagrams.
7. ***Collaboration diagram**:* Objectives and a modeling lang. must have a way to represent this CD will represent this feature.
8. ***Component diagram**:* Currently system evolution is a collective team effort, where different members of the team work on different components of the system. Thus the component diagram is essential to model the system.
9. ***Deployment diagram**: :* Physical architecture of a computer based system which shows the different aspects of the system where the components reside.

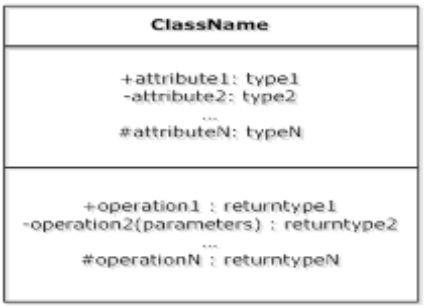
5.1 Class Diagram

Class Diagram represents the static view of the system with several classes connected with relationships.

Objective

Define attributes and methods for different classes alongside binding the classes with relationships to form the class diagram.

Template for Class



- ClassName: Name of the class.
- attribute1..N: Properties of a class.
- type1..N: Attribute types (string, int, float or user defined data).
- operation1..N: Operation indicates the behavior of the class on the attributes.
  returntype1..N: Specifies the return type of the method (void, string, int, float or user defined).

Naming Convention

ClassName: Class name starts with upper case letter. *(Eg. Customer)*

Attribute: Attribute starts with small letter with word more than one wherein subsequent words are started with uppercase. *(Eg. +customerName for public attribute)*

Operation: Opertaion starts with small letter with word more than one wherein subsequent words are started with uppercase. *(Eg. -getCustomerName for private operation)*

Attributes and Operations will be prefixed with +/-/# to indicate public/private /protected. These define visibility/accessibility of the attributes/operations from other classes.

1. *public:* usability extends to other classes(+).
2. *private:* usability restricted within the class and not accessible from other classes(-).
3. *protected:* usability only from the classes which inherits from original class(#).

Additional features to attributes

Constraints: free form text enclosed in braces. (one/more rule that class follows)

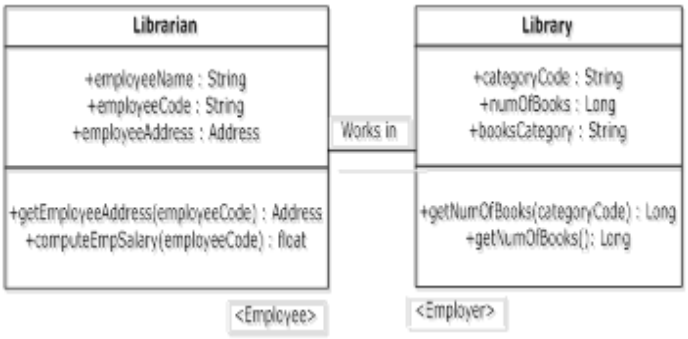
Notes: attached to attributes/operations and give additional info to a class (can be graphic as well as text)

Derivation of class

In the model, *Nouns* form the *ClassName*, *Verbs* form the *Operation* and *attributes* emerge as *nouns* relating to class.

A. Associations:

The Conceptual relationship between two classes is defined as an Association. Association is indicated by a line connecting to two classes, with the name of the association just above the line.

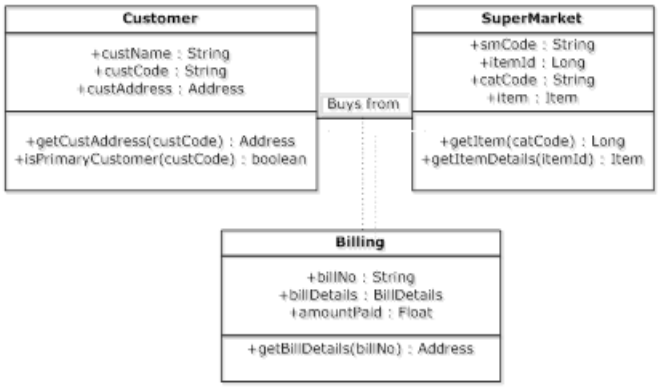


When one class associates with another, each one usually plays a role within that association. *(Eg. Employee and Employer)*

***Address* is user defined data type.

Association can be more complex wherein several classes can connect to one class.

Sometimes an association has to follow a rule. This is indicated by putting the constraint near the association line.



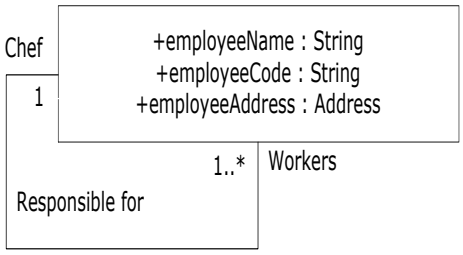
Similar to Class, an association can have attributes and operations. In this case we have an association class. Association class (Billing) represents similar to class but connected with dotted line to the association line.

***Item* is user defined data type.

B. Multiplicity:

Multiplicity is a special type of association which shows the no. of objects from one class that relate with no. of objects in an associated class. The different types of relationships are

- One–to-one
- One-for-many
- One-to-one or more
- One-to-zero or one
- One-to-a bounded interval (one to-two through twenty)
- One-to-exactly n
- One-to-set of choices (one–to-five or eight)

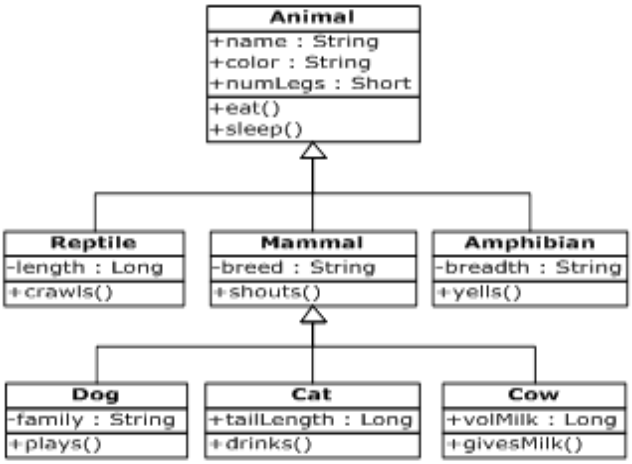


UML uses (*) to reps. more and to reps. many.

Sometimes a class is an association by itself. This can happen when a class has objects that can play variety of roles. These associations are called reflexive associations.

Inheritance & Generalization

Inheritance is an object oriented principle of abstracting the properties/ behavior available in one class into the other class. This hierarchy of abstraction in object oriented paradigm is referred as inheritance.



Animal has the properties and behavior which are available for the animals Reptile, Mammal and Amphibian with the properties and behavior of their own. Similarly Dog, Cat and Cow are mammals, wherein the properties and behavior of Mammal are available for them.

A class (child/subclass) can inherit attributes and operations from another class (parent/super class), parent class is more general than the child class.

In the above cited example, Animal is a parent/super class with Reptile, Mammal and Amphibian being the child/sub class. Mammal is a parent/super class with Dog, Cat and Cow being the child/sub class.

In generalization, a child is a substitutable part for a parent i.e. anywhere the parent appears, child may appear. Reverse is not true.

*From association point of view: inheritance is a kind of association.

Discover inheritance

1. Identify set of attributes and operations of a class which are general across the set of classes and embed them in the parent/super class, while child/sub class can inherit the parent class along with its own attributes and behaviors.
2. Inheritance can also be applied where two/more classes have common attributes and/or operations.

3. A class can inherit attributes and operations from another class. The inheriting class is the child of the parent class it inherits from. Abstract classes are intended only as bases for inheritance and provide no objects on their own.
   **Classes that provide no objects are said to be abstract classes indicated by italics.*

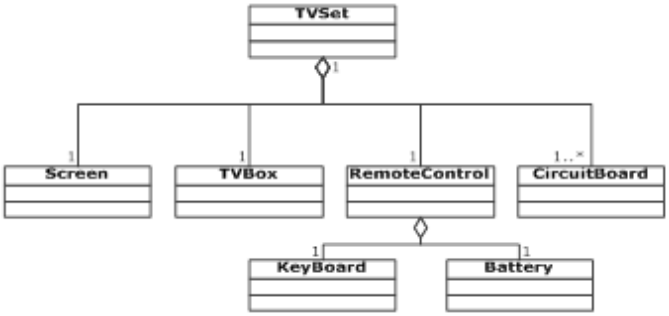
Aggregations

A class can contain no. of component classes which is defined as special type of relationship called *aggregation*. The components and the class they constitute are in a *part- whole* association.

Aggregation

Aggregation is represented as hierarchy with the whole class at the top and the component below. A line joins a whole to a component with an open diamond on the line near the whole.

TVSet is the whole class at the top with the components Screen, TVBox, RemoteControl and CircuitBoard.



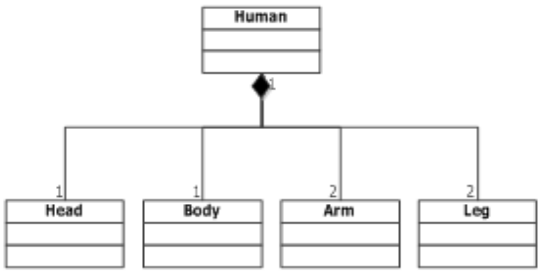
Also, RemoteControl class with Components aggregation of Keyboard and Battery.

Sometimes the set of possible components in an aggregation falls into *OR* relationship.

To model this, use constraint - the word or within braces on a dotted line that connects two part-whole lines.

Composite

Composite is a strong type of aggregation. Each component in a composite can belong to just one whole. Symbol for composite is same as the symbol for an aggregation except the diamond is filled.



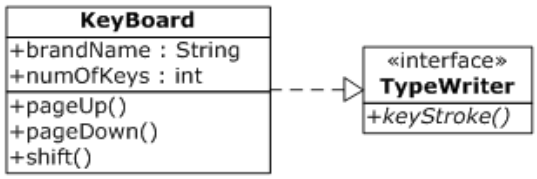
Human being (object) has head, body, arms and legs.

A component association is one in which each component belongs to exactly one whole.

Interfaces and Realizations

When the classes are not related to parent but their behaviors might include some of the same behaviors with the same signatures. In such Code for one of the classes and reuse them in others.

An interface is a set of operations that specifies some aspect of the class behavior and a set of operation a class presents to other classes. Interface is modeled similar to class with no attributes, only operations. Another way is with a small circle joined with line to a class.



KeyBoard is a class which inherits the operations from interface. keyStroke() is a common operation made available in interface which KeyBoard implements along with its own attributes and operations.

To diff. from class <<interface>> or "1"

Relationship b/n class and an interface is called Realization. It is indicated by dashed line and open triangle.

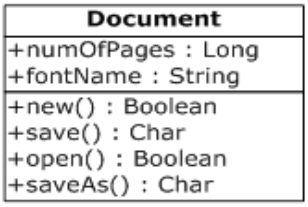
Class Diagram: Case Study

Requirement: Class Diagram for Creating Text Document.

Step1: Derivation of Classes

First step is to identify nouns (Class Names) would be Document, Page, Header, Footer, Character, Table and Picture. Also, identifying the verbs (behavior) for each of these classes.

Class: Document

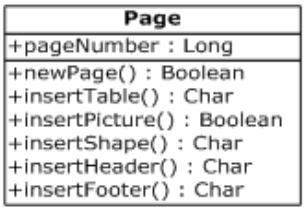


Document will be a central class with pages and ability to new, save, open and saveAs.

- Attribute – numOfPages and fontName.
- Operations – new(), save(), open() and saveAs().

** Page will itself be a class.*

Class: Page

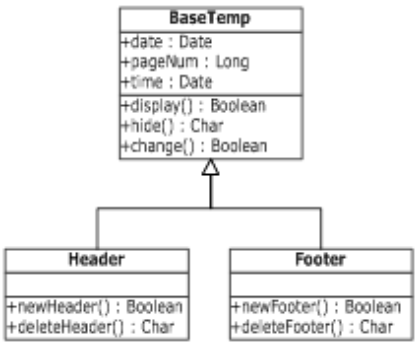


Page will be a class with ability to new, insert table, insert header, insert footer etc.

- Attribute - pageNumber
- Operations - newPage (), insertHeader() and insertFooter() along with other operations as well.
  ** Header and Footer can be classes as well.*

Classes: Header & Footer

In this case we have common attributes (like date, time, page number etc.) and common operations as well (like display, hide, change etc.). Hence, we can have a properties inherited from the base class (Inheritance). Define parent class which holds the common attributes and/or behavior.



Header and Footer classes (child) are the inherited from the BaseTemp (parent).

- Common Attributes - date, time and pageNum.
- Common Operations - hide (), display() and change().

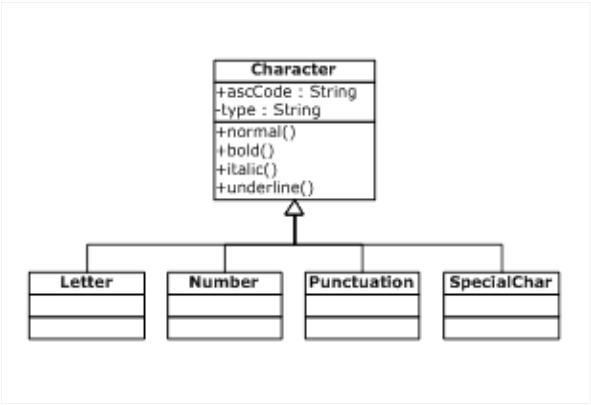
Header Class:

- Operations - newHeader (), deleteHeader().

Footer Class:

- Operations - newFooter(), deleteFooter().

Class: Character



Character Class (parent) with the child classes as Letter, Number, Punctuation and SpecialChar.

- Attribute – ascCode and type
- Operations - normal (), bold(), italic() and underline().

Class: Table

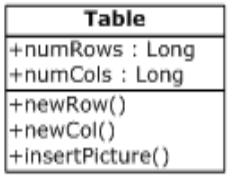
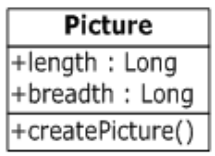


Table is a class which defines the properties of the Table along with their behavior.

- Attribute – numRows and numCols.
- Operations - newRow(), newCol() and insertPicture().

Class: Picture

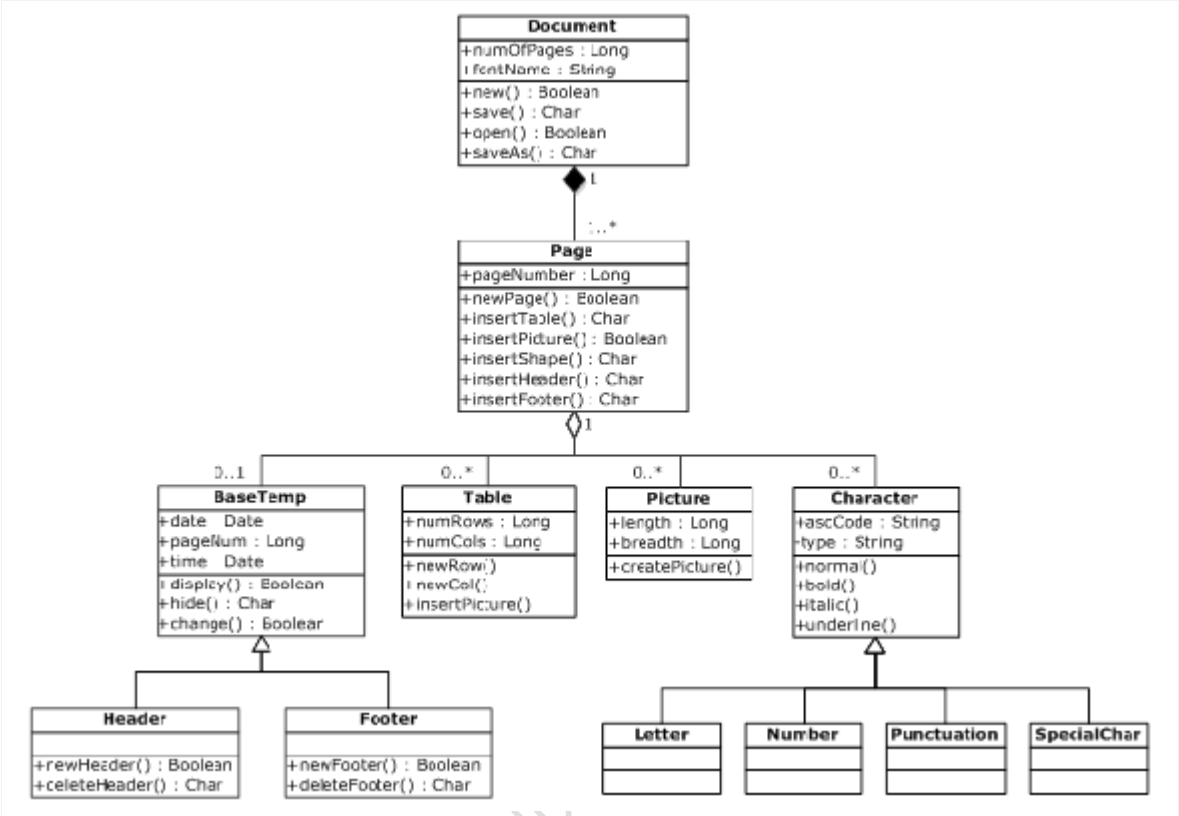


Picture is a class which defines the properties of the Table along with their behavior.

- Attribute – length and breadth.
- Operations - createPicture().

Step 2: Establishing Relationship Diagram

The complete class diagram for creation of document is arrived at by establishing the appropriate relationships between the different classes.



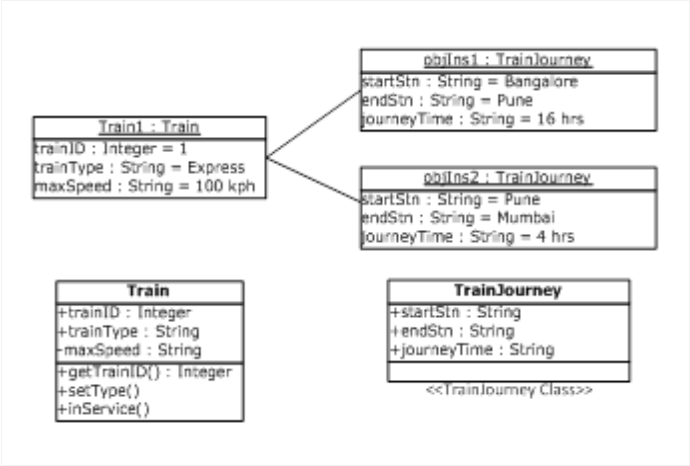
5.2 Object Diagram

Object diagram is derived from a class diagram which represents an instance of a class diagram. Similar to class diagram, object diagram represent the static view of a system. However this static view is a snapshot of the system at a particular moment. The difference is that a class diagram represents an abstract model consisting of classes and their relationships. But an object diagram represents an instance at a particular moment which is concrete in nature.

Both diagrams are made up of same basic elements but in different form. In class diagram, elements are in abstract form to represent the blue print and in object diagram the elements are in concrete form to represent the real world object.

Object Diagram: Case Study

Requirement: Depicting object instances for Train and TrainJourney.



Train1 is the name of the instance of the Train class separated by colon and underlined to indicate this is an object. Likewise 'objIns1' and 'objIns2' are the instances of 'TrainJourney class.

'objIns1' is a snapshot of an object at one moment and 'objIns2' at another.

Train1 object is associated with two journey objects.

5.3 Use Case Diagram

Use case provides a static view of the system to the client. The initial discussion of the flow during requirements collection reveal actors and high level use cases that describe functional requirement in general terms. Subsequent discussion will enable deep dive into the requirements.

Use cases are the collection of these scenarios about the system. Each scenario describes a sequence of events which are initiated by an entity (person, another system or hardware/time) referred as actors. The result of the sequence should be useful to actor or another actor.

Use cases can be reused as applicable.

Inclusion is to use the stops from one use case as part of the sequence of steps to another use case.

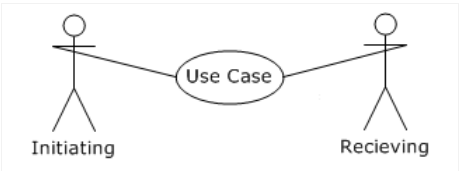
Extension is to create a new use case by adding steps to existing use case.

Class diagrams- static view

Static view helps an analyst communicate with the client. The dynamic view helps an analyst communicate with the developers and helps them to create programs.

Use Case model

Actor initiates the use case and receives information from the use case.



Association line connects between actor & use case representing communication between them.

Use Case Relationships:

1. Inclusion
2. Extension
3. Generalization
4. Grouping

Inclusion enables to reuse one use case steps inside other use case.

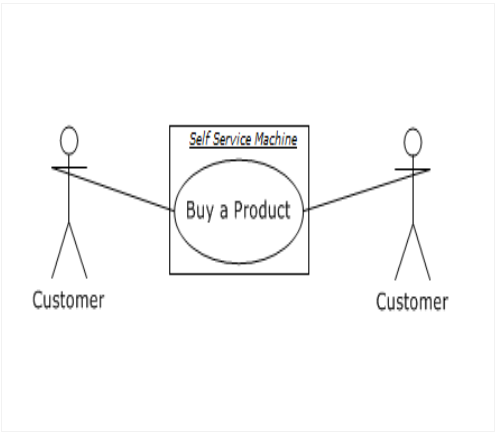
Extension allows to create new use case by adding steps to the existing use case.

Generalization classes can inherit from other class and so can use cases. In use case inheritance, the child use case inherits from parent and adds it onto behavior can apply child wherever we apply parent. Generalization relationship can exist b/n actor as well as use cases.

Grouping multiple use cases organized as one single unit. Scenario where in one system consists of several sub systems. Related use cases organized into package.

Use Case Diagram: Case Study

Requirement: Create Use case relationship for a Self Service Machine for the functionality 'Purchasing product by customer'.



Main use case: *Buy a product*

Actor: *Customer*

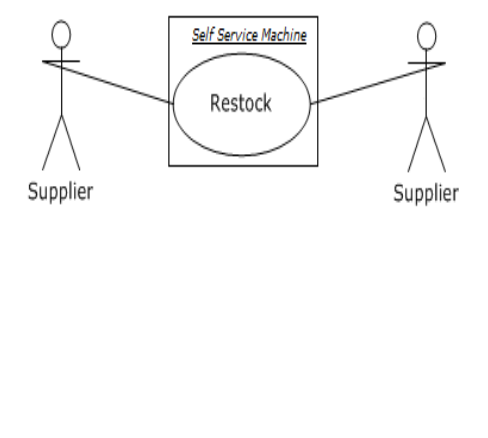
Scenario sequence:

- Inserts money into self-service machine.
- Selects one or more products.
- Self-service machine provides selected product to the customer.

In this case, it is possible that the self-service machine is out of one or more products or machine doesn't have exact money to return back to the customer. In either of

these cases, Self-service should display appropriate message accordingly to the customer and provide options for him to make selection/return money back. This is a use case from customer/user view point.

However, there are users to 'restock the self-service machine *(Supplier)'* and to 'collect money from the self-service machine *(Collector)'*. This implies to create at-least 2 more use cases: "*Restock*" and "*Collect Money*".



Use case: *Restock*

Actor: *Supplier*

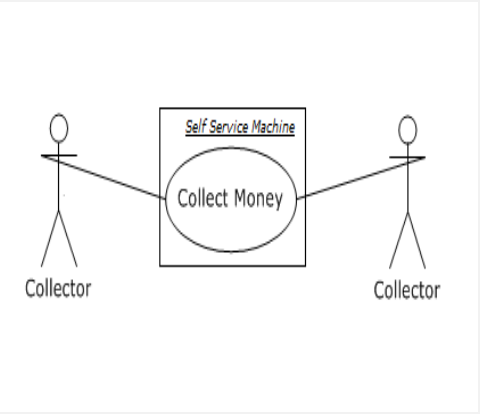
Scenario sequence:

Actions that the *Supplier* perform

- Unsecure self-service machine.
- Open self-service machine.
- Fill the brands into compartment to capacity.
- Close self-service machine.
- Secure self-service machine.

Precondition - time elapsed since last appraisal of products

Post condition- new set of potential sales.



Use case: *Collect Money*

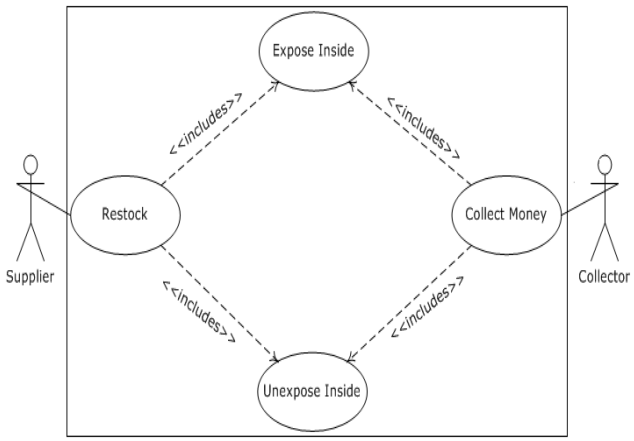
Actor: *Collector*

Scenario sequence: *Collector* has to perform similar actions that of *Supplier*, but *Collector* will deal with money rather than products. *Collector* may be same person as *Supplier*.

- Post condition- money in hands of collector.

For both *Supplier* and *Collector*: Unsecure, Open, Close and Secure are the common actions.

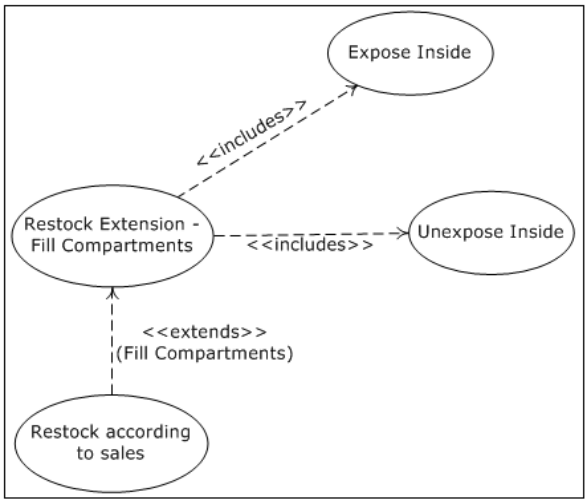
We could combine '*Unsecure*' and '*Open*' into a single use case '*Expose Inside*' and '*Secure*' and '*Close*' into a single use case '*Unexpose Inside*'.



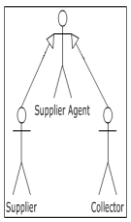
Both use cases '*Expose Inside*' and '*Unexpose Inside*' can be then included using <<includes>> relationship for '*Restock*' and '*Collect Money*' use cases.

'*Restock*' use case can be basis of another use case '*Restock according to sales*'.

In this case, *Supplier* may fill up brands with new products according to the sales of the products (extension relationship).



Also, Generalization can exist for the actors *Supplier* and *Collector* when both are executed by the same entity *(Supplier Agent).*



Then *Supplier* and *Collector* would be children for *Supplier Agent.*

5.4 State Diagram

State diagram deals with the behavioral elements that show how parts of the UML model can change over time. As system interacts with other systems or users, the objects in the system will undergo change depending on the interactions to accommodate with these changes. State Diagram represents this state changes within the system.

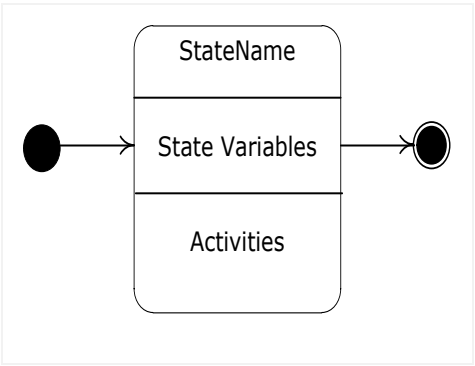
State diagrams provide the analysts or the designers to understand the behavior of the objects in the s/m. developers need to know this behavior to implement in the s/m.

The Simple state change scenario is when *switch pulled*, a light changes from '*off state*' to '*on state*'. The other scenario is, in screen saver mode *on key stroke* or *mouse movement* computer '*leaves*' screen saver mode and returns to '*working-active*' mode.

Symbol	Description
State	Rounded rectangle - states
	Transition – solid arrow head
	Initial State
	Final State

UML state diagram shows the states that the object can be in, along with the transitions b/n the states, shows the starting point and ending point of the sequence of state changes.

It shows the state of single object.



StateName – Defines name of the State.

State Variables – defines the variables associated with State (Eg: timeout, counters, date etc.).

Activities – defines the different activities in this state

- entry: what happens when s/m enters the state.
- exit: defines the exit state (leaves the s/m)
- do: defines the functionality within this state.

State details and transitions:

Indicate an event that causes a transition to occur (a trigger event) and the computation (the action) that executes and makes the state change happen.

To add events and actions, write near the transition line using a slash to separate the trigger event from an action.

Sometimes an event causes a transition without an associated action or transition occurs because state completes an activity (rather due to an event). This type of transition is called *triggers less transition*.

Guard condition- Guard Condition is the defined criteria for the transition to take place when this condition is met (Eg: Screen 'saving mode' is activated at defined time interval)

Some of the states can be more complex represented by a rounded rect. It is possible to have states inside states. These states are called sub states (S*equential and Concurrent*).

Sequential sub states are sequence of states which occur one after another.

Concurrent sub states must consist of two or more sequential sub states, which occur at same time represented with dotted line between concurrent states.

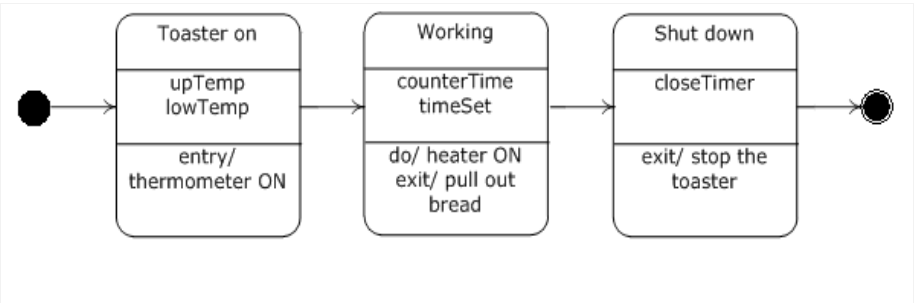
- A message that triggers a transition in the receiving objects state diagram is called a *Signal*. Sending a signal is similar to creating a signal object and transmitting it to the receiving object. Signal object has properties that are represented as attributes therefore signal is an object, it's possible to create inheritance hierarchies of signals.

State diagram: Case Study

Requirement: Define a State Diagram for the Toaster functionality. The Sequence of operations are

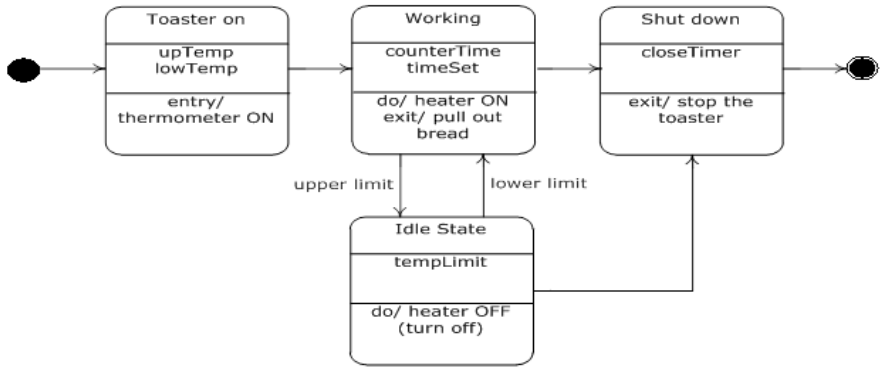
1. Turn on the toaster.
2. Put in bread.
3. Wait for several minutes.
4. Pull out the bread.

Initial state diagram: The initial state diagram represents the sequence of states that the toaster be in,



As an exceptional scenario, '*To prevent burning out*' of the bread the system has to check the temperature for the *upper* and *lower* limits.

When the temperature reaches an *upper* limit, it has to go an idle state and stay in '*idle state*' until the temperature reaches *lower* limit and kick back again. The '*idle state*' is added to



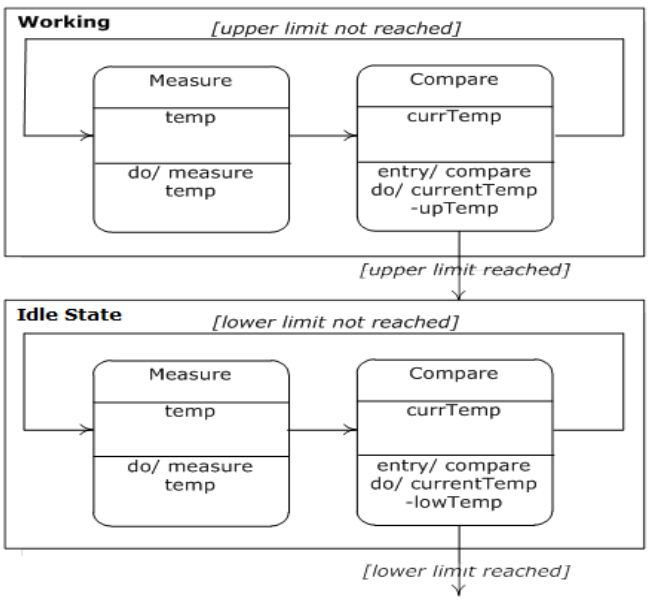
depict this transition.

Transition between '*Working*' state and '*idle state*' depends on comparison of the temperature limits with the standard value. For which, '*Working*' state needs to have sub states to perform this operation.

Sub states for working and idle states are '*Measure*' and '*Compare*' with defined state attributes and operations.

Working sub states: 'Measure' measures the current temperature and '*Compare*' compares with the standard temperature and transitions the state to '*idle state*' when the temperature reaches *upper* limit.

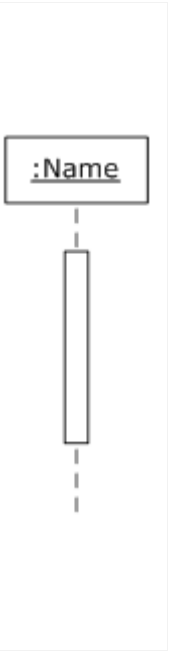
Similarly, the transition happens from 'idle state' to 'Working' state or 'Shut down' when the *lower* limit is reached.



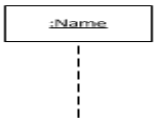
5.5 Sequence diagrams

Sequence diagram shows the communication between objects. It adds a '*time*' parameter to show the interaction between the objects in a specified sequence. The sequence takes '*time*' to go from beginning to end.

Sequence diagram consists of an 'objects' represented with named rectangles (name underlined), 'messages' represented by solid arrows and 'time' represented by vertical progression.

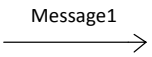
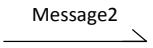


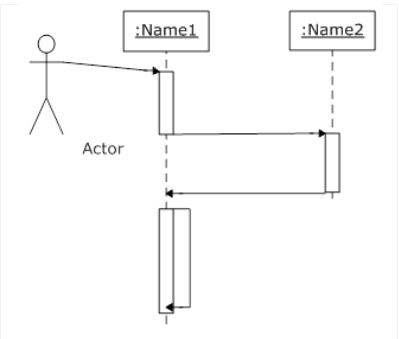
- 'Activation' represents execution of the operation the object carries out.
- 'Length' of the activation signifies its duration.

Symbol	Description
	Object representation with the dashed line called as 'Life line'
	The narrow vertical rectangle along the life line is called as 'Activation'.

Messages: A message is an information that traverses from one object's life line to another objects life line. An object can send a message to itself, i.e. from its life line to back to its life line which is referred to as '*Recursion*'.

The basic message types are

Message Type	Description	Symbol
Simple	Transfer control from one object to other.	
Synchronous	When an object sends synchronous message, it waits for a response to this message from the other object before it proceeds with further operations.	
Asynchronous	When an object sends asynchronous message, it do not wait for a response to this message from the other object. However it proceeds with further operations.	



This diagram reps. the time in the vertical direction. Time starts at the top and progresses toward the bottom. The message that's closer to the top occurs earlier in time than a message that occurs closer to the bottom.

Actor initiates the sequence (not part of SD)

In SD, the objects are laid out from left to right a/c the top.

Each objects life line is a dashed line extending downward from the object. A solid line with the arrow head connects from one life line to another and reps. a message from one object to another. Time starts at top and proceeds downward.

Ways of creating sequence:-

SD – instance SD

-generic SD

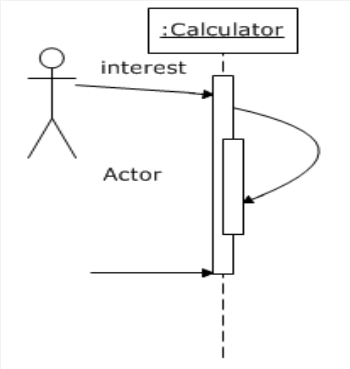
Generic SD provide opportunities to reps. if statements and while loops. Enclose each condition for an "if" s/m in square brackets. Same with "white" but prefix the left bracket with an asterisks'.

It's possible in SD to create an object. When this occurs, a created object is reps. in usual way- named rectangle.

Diff is it is not reps. at top but along vertical dim. So that its location corresponds to the time when its created. The message that creates the object is labeled create ().

Sometimes an object has an operation that invokes itself. This is called recursion.

E.g.:- suppose one of the objects in the s/m is calculator, and suppose one of its operations to perform compound interest. In order to compute compound interest for a time frame that encompasses several compounding periods, the objects interest- computation operation has to invoke itself a no. of times (as shown below)



Reps. Recursion in a SD

Sequence diagram example

Case: Self servicing m/c

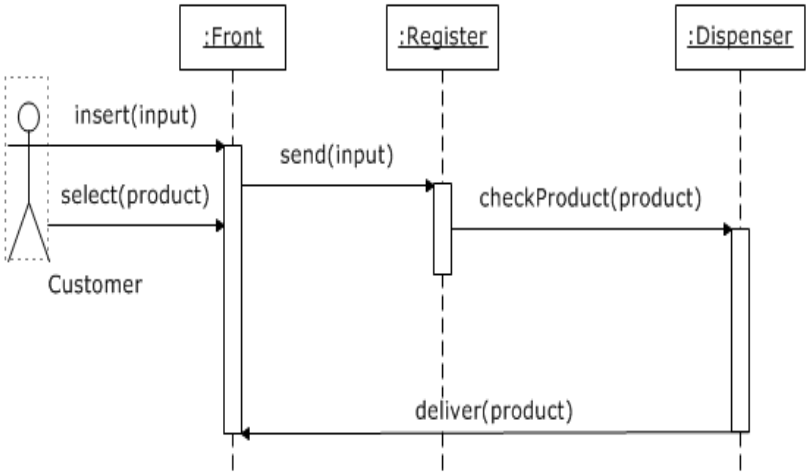
Objects identified:

- The front – interface- that m/c presents to the customer
- Money register- as part of the m/c where moneys are collected.
- Dispenser- disperser selected product to the customer

Instance SD:

Sequences:-

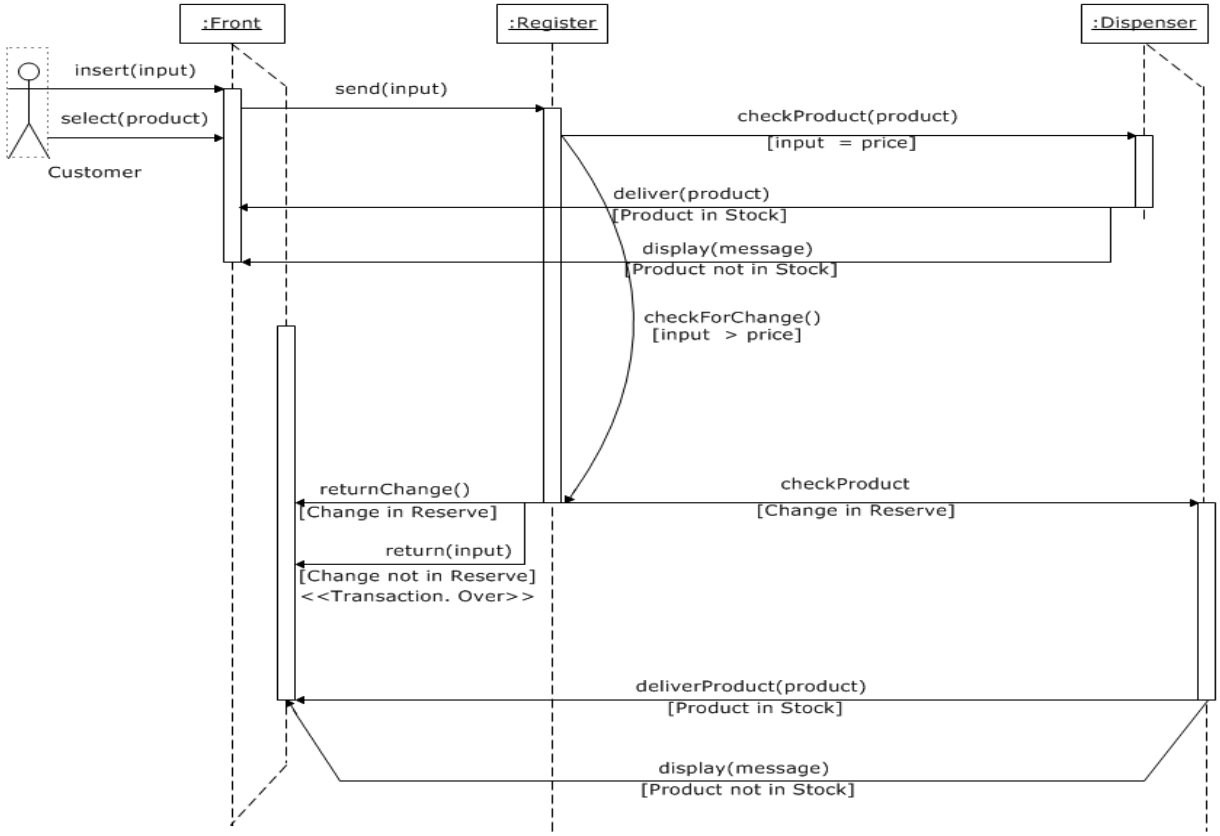
1. A customer inserts money into the money slot
2. Customer makes selection
3. Money travels to the register
4. Register checks to see whether the selected product is in the dispenser
5. Register updates its cash reserve
6. Register has a dispenser deliver the product to the font of the m/c

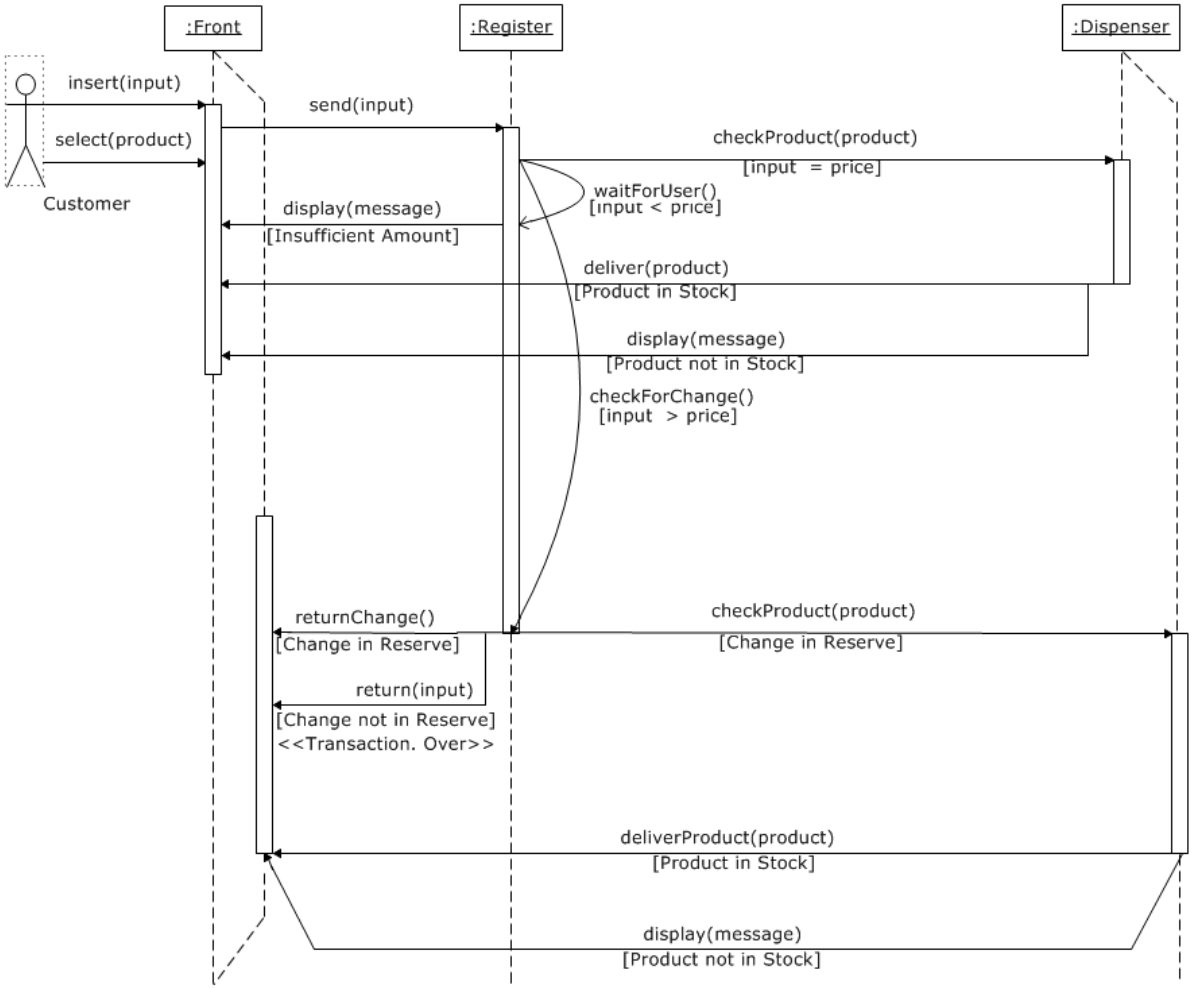


Ref: in use case diagram- 2 more scenarios were introduced to reps. out of product case and incorrect amount of money case. If you consider all of a use cases scenario – generic SD

Out-of-product sequence:-

1. After selecting the "SOLD OUT" brand the message flashes
2. Prompt for another selection must be displayed
3. Customer has an option of pushing a button that returns money
4. If the customer selects an in-stock brand, everything proceeds as in the best case scenario if the I/p amount is correct. If not m/c follows incorrect money sequence.
5. If the customer selects another sold-out brand, the process repeats until the customer selects an in-stock brand or pushes a button that returns money.


:Front
:Register
:Dispenser
insert(input)
select(product)
Customer
send(input)
checkProduct(product)
[input = price]
deliver(product)
[Product in Stock]
display(message)
[Product not in Stock]
checkForChange()
[input > price]
returnChange()
[Change in Reserve]
checkProduct
[Change in Reserve]
return(input)
[Change not in Reserve]
<<Transaction. Over>>
deliverProduct(product)
[Product in Stock]
display(message)
[Product not in Stock]



Out-of-product scenario

Incorrect-amount-of-money sequences:

1. Register checks with customers input with the price of the product
2. If the amount is greater than price, the diffn is calc and register checks is cash reserve
3. If the difference exists in the class reserve, the register returns the change to customer and everything proceeds as before.
4. Otherwise, the register returns I/p money and displays a message that prompts the customer for the correct amount.
5. If the amount is less than price, the register does nothing and waits for an additional money to be inserted.

1. Collaboration diagram

Collaboration diagrams show the object interaction. It shows the objects along with the messages that travel from one to another.

SD and CD are similar. They are semantically equivalent. We can turn SD to CD and vice versa.

- Main difference is that SD is arranged according to "time", CD according to "space"

A collaboration diagram is an extension of an object diagram. In addition to the associations among objects, CD shows the messages the objects send each other.

Reps.:-

1. Messages are resp. with arrows that points to receiving object, near the association line b/n the objects
2. Label near arrow reps. message is
3. Message indicates the receiving object to execute one of its operation
4. A pair of parenthesis ends message. Inside parenthesis put parameters the opn. Works on.

To reps. SD in CD

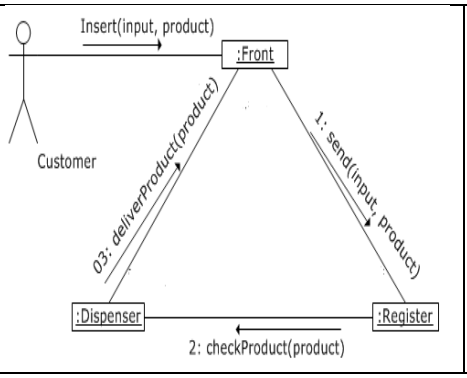
1. Add a no. to the label of the message, with no. corresponding to sequence order "colon" separates no. from message.

Collaboration diagrams: example

Case study: self-service m/c

Best case scenario:-

1. Customer inserts money in the m/c and makes selection of one/more products present in the m/c
2. When the register gets the money (correct amt, product present), selected product is delivered to customer
3. Dispenser delivers the product to the front of the m/c and the customer gets the product.



Worst case:

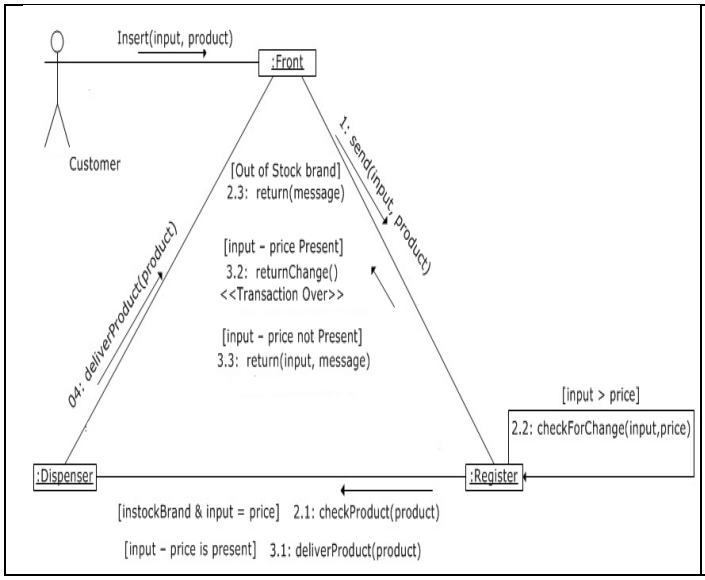
2 scenarios – out of product

-Incorrect-amount-of-money

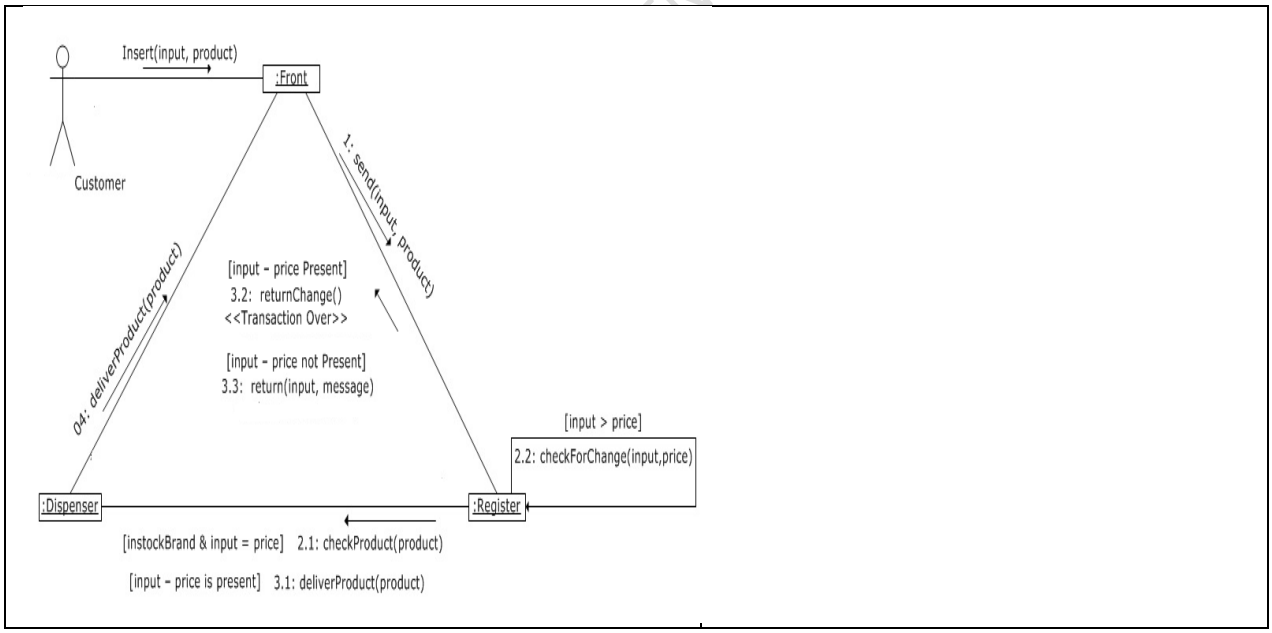
Reps:-

1. Conditions – square brackets preceded with label (message)
2. Co-ordinate with numbering
3. Modeling if conditions: messages have same number and decimal pt. is added with numbers. This is called "nesting"

Out-of-product scenario:-



Incorrect-amount-of-money scenario:



Topics:-

1. State diagram models the states of single object
2. Class, object or use-case diagram models a s/m or at least part of s/m

3. A "trigger less transition" is a transition that occurs because of activities within a state rather than in response to an event
4. While condition – square brackets and precede with left bracket [*] asterisk
5. To reps. message in the collaboration diagrams- arrow near the association line pointing to receiving object.

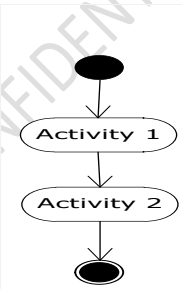
2. Activity Diagram

Activity diagram is basically a flow chart to represent the flow form one activity to another activity. The activity can be described as an operation of the system.

Activity Diagram uses 'Control Flow' to transition from one activity to another. The flow can be sequential, branched or concurrent.

Symbol	Description
	Rounded rectangle
	Control Flow
	Initial State
	Final State
	Decision
	Transition (Fork)
	Transition (Join)

State diagram shows the transition between the states wherein Activity diagram is an extension of state diagram which highlight the activities.



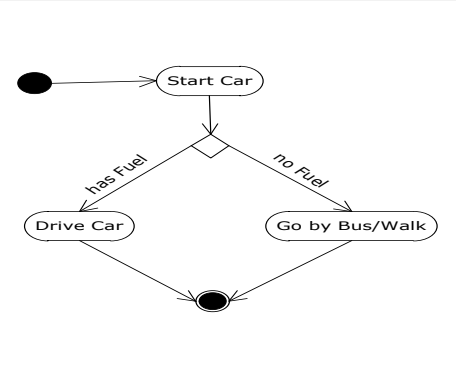
The processing within an activity goes to completion and then an automatic transition to the next activity occurs. Control Flow show the transition of one activity to the next.

Building an activity diagram

Activity diagram primarily transitions the activities through control flow based on decisions, concurrent paths, signals and swim lanes.

Decisions

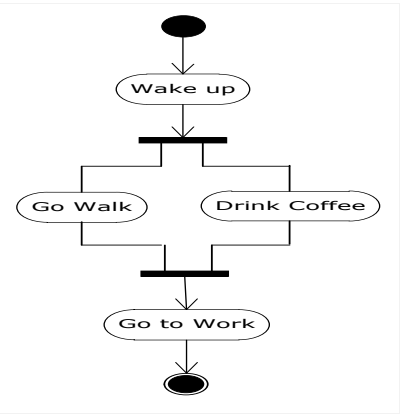
Decisions are the scenarios where-in we need to decide to perform definite activity based on certain scenario.



Imagine a scenario where you try to 'start your car' (activity), the possible paths coming out of this activity is that the "Car starts" or "Car doesn't start". In either case you need to perform certain activity, either you 'drive the car' (activity <<Car starts>>) or 'Go by bus/walk' (activity <<Car doesn't start>>)

Concurrent paths

Concurrent Paths are the scenarios where-in we need to separate a transition into two or more separate paths to perform different activities concurrently (at the same time) termed as 'Transition (Fork)' and join back termed as 'Transition (Join)' together to perform single activity.

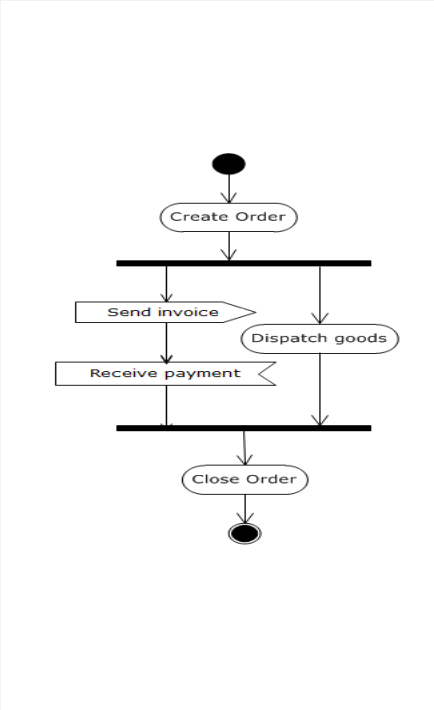


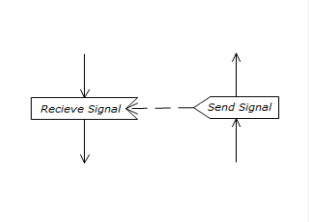
Imagine a scenario when you 'wake up' (activity), you would 'go walk' (activity) along with 'drinking a coffee' (activity) at the same time (concurrent) then you 'go to work' (activity). The concurrent path is represented as shown here.

Transition (Fork) enables the control flow to perform concurrent activities and Transition (Join) enables the control flow to converge to a single activity.

Signals

Signals are the scenarios where-in the activity takes place post receiving defined signal from the other activity. The activity in this case will send a signal for the other activity, which receives the signal and starts its activity.





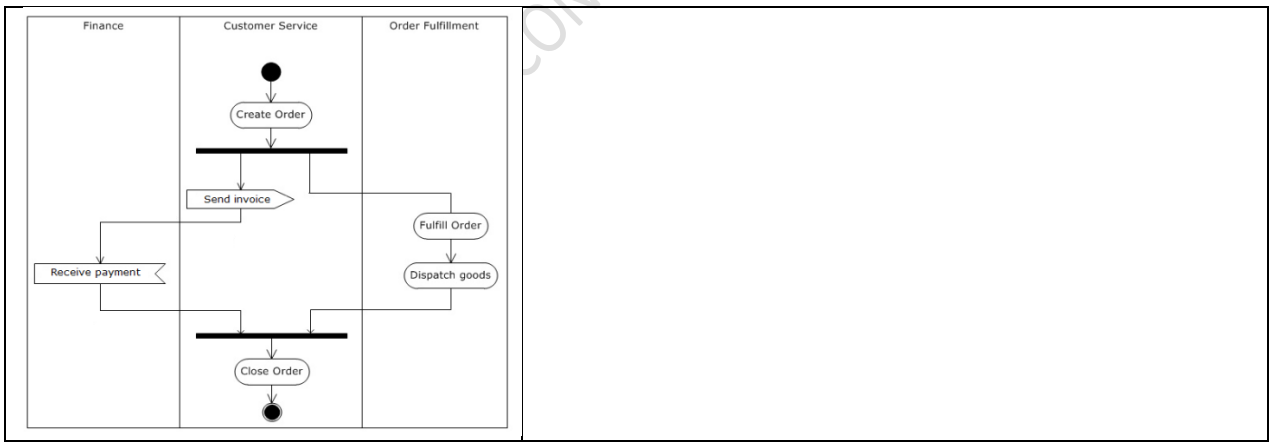
During a sequence of activities, it's possible to send a signal, upon receiving signal next activity takes place.

Imagine a Scenario where you 'create an Order' (activity), you would 'send the invoice' (activity). Post dispatching invoice you would 'receive payment' (activity).

In this case, 'Send invoice' will send the signal to the 'Receive Payment' post invoice is sent.

Swim lanes

Swim lanes are the Scenarios where-in the set of activities aligned to the different roles which are separated into parallel segments with role. Each segment is referred to as swim lane.



More about interfaces and components:

In this we deal with real world entities: software component. A software component is a physical part of a system. It resides in computer but not in the mind of the analyst.

The relationship b/n a component & a class-

Component is a software implementation of a class. Class reps. an abstraction of a set of attributes and operations. A component may be an implementation of one/more class.

Components and their relationships should be modeled such that

1. Clients can see the structure in the finished s/m
2. Developers have a structure to move forward to work.
3. Technical writers who have to provide documentation and help files can understand what they are writing about
4. You're ready for reuse

When dealing with components, we have to deal with their interfaces. Interface is the objects "face" to the outside world. So that other objects can ask the object to execute its opns. which are hidden with encapsulation.

An interface is a set of operations that specifies something about a class behavior. It is a set of opns class reps. to other classes for a modeler, an interface for a class is similar to interface for a component. As is the case with a class and its interface, the relation b/n a component and its interface is called realization.

We can replace one component with another if the new component conforms to the same interface as the old one. You can reuse the same component and access though its interfaces.

In modeling, we deal with three kinds of components:

1. Deployment components
   -from the basis for executable systems (DLL's , executables, beans)
2. Work product components
   -from which the deployment components are created (data files & source code files)
3. Extension components – created as result of a running s/m.

Note:-

Instead of representing a conceptual entity such as a class or a state, a component diagram reps a real world item- software component.

Software components reside in computer, not in the mind of analyst. A component is accessible thro its interface. The relation b/n a component & interface is called realization. When one component access the services of another, it uses an import interface, the component that realizes the interface with those services provides an export interface.

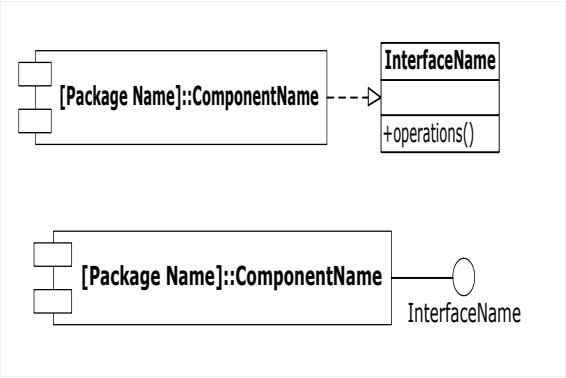
3. Component diagram

Component diagram is used to model physical aspects of a system which includes elements like executable, libraries, files, documents etc. which resides in a node. These diagrams

are used to visualize the organization and relationships among components in a system. The component is represented as below.

Component diagram does not describe the functionality of the system but it describes the components used to make those functionalities. Component diagram is composed of components, interface and relationships.

A component and the interface it realizes can be represented in two ways.



To show the interface as a rectangle that contains interface related information. It's connected to the component by the dotted line and empty triangle that visualize realization.

To show interface is with small circle connected to the component by a solid line, which reps the realization relationship.

Component diagram: Case Study

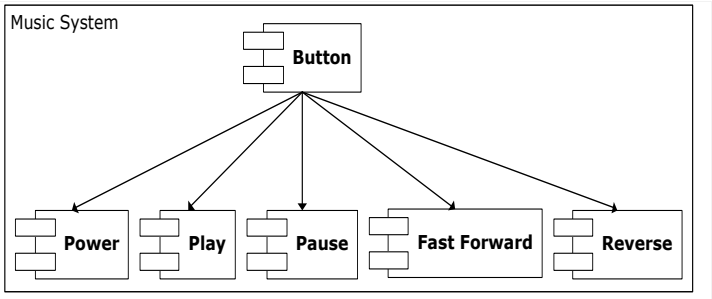
Case: music player

Components:

Controls: play, stop, eject etc.

Controls will be realized by buttons.

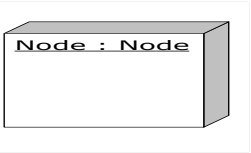
Buttons are separate components and can be reps by UML diagram as shown.



All the components belong to one single global component called button. But the actions/behavior performed is different. We obtain these actions by programming them.

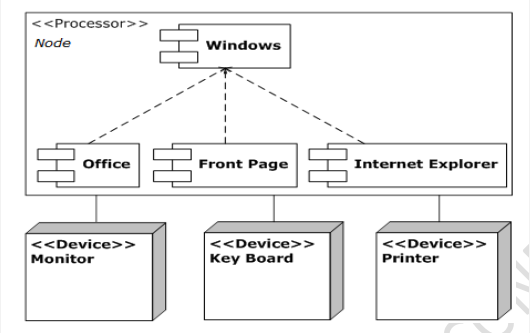
4. Deployment diagram

Deployment diagrams are used to visualize the topology of the physical components of a system where the software components are deployed. Deployment diagrams are used for describing the hardware components where software components are deployed. Component diagrams and deployment diagrams are closely related. Component diagrams are used to describe the components and deployment diagrams shows how they are deployed in hardware.



Node is used to reps the deployable component. Each Node deploys software components in the system. To indicate deploys components, we show them in dependency relationships with a node.

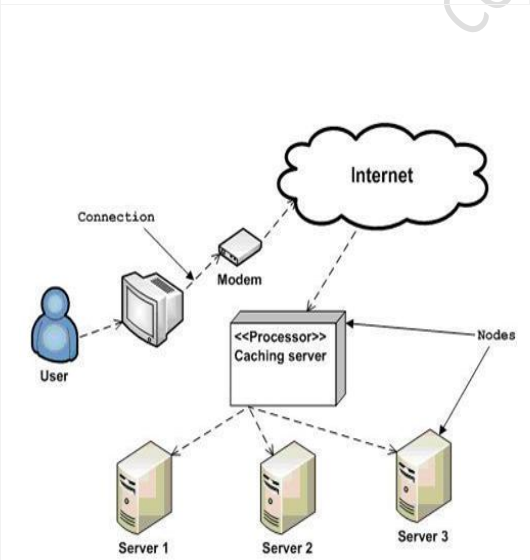
Deployment Diagram: Case Study



Case Study1:

The deployment diagram depicts the nodes Monitor, Keyboard, Pinter and Processor are the physical/hardware components.

The Software components Windows, Office, Front Page and Internet Explorer are deployed in the Processor (Node).



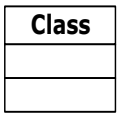
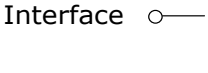
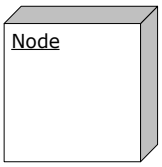
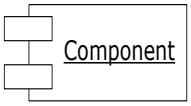
Case Study2:

A web based application (Software Component) deployed in a clustered environment using server 1, server 2 and server 3(Hardware Components). The user is connecting to the application using internet. The control is flowing from the caching server to the clustered environment.

The nodes in this case are Monitor, Modem, Caching server and Server. Servers are the physical/hardware components on which the Software component

A. UML Diagram Symbol Set

1. Structural elements

Class	Interface	Use Case	Deployment Node	Component
				

2. Behavioral elements

State	Sequence	Collaboration	Activity