MINI PROJECT REPORT

ON

ONLINE VOTING SYSTEM

Submitted in partial fulfillment of requirements to

CS 356 - OOMD LAB

By
Pulapa Sai Kiran (Y20CS147)

JULY 2023
R.V.R & J.C.COLLEGE OF ENGINEERING

(AUTONOMOUS)

(Approved by A.I.C.T.E) NAAC A Grade
Chandramoulipuram : : Chowdavaram
GUNTUR – 522 019

R.V.R & J.C.COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that this mini project work titled "**ONLINE VOTING SYSTEM"** is done by **Pulapa Sai Kiran** in partial fulfillment of the requirements to CS**-356, OOMD Lab** during the academic year 2022-2023.

M.Basha
Lecturer Incharge

Dr. M.Sreelatha
Prof.&HOD, Dept. of CSE

ACKNOWLEDGEMENTS

The successful completion of any task would be incomplete without a proper suggestions, guidance and environment. Combination of these 3 factors acts like backbone to our Project "**ONLINE VOTING SYSTEM".**

We are very much thankful to **Dr. K.Srinivas**, principal of **RVR & JC COLLEGE OF ENGINEERING**, Guntur for having allowed delivering this mini project.

We express our sincere thanks to **Dr.M.Sreelatha**, Head of the Department of Computer Science and Engineering for her encouragement and support to carry out this mini project successfully.

We are very glad to express our special thanks to **M.Basha,** Lecturer- in charge for the mini project, who has inspired us to select this topic, and also for his valuable advicesin preparing this mini project topic.

Finally we submit our reserves thanks to Lab staff in **Department of Computer Science and Engineering** . And to all our **friends** for their cooperation during the preparation.

Pulapa Sai Kiran(Y20CS147)

CONTENTS

DESIGN

Implementation diagrams.

1. PROBLEM STATEMENT

The basic methodology as applied to online voting system would involve giving voter realistic voting tasks to accomplish using a variety of ballot designs. Voting task performance is measured using variables such as accuracy, time, and workload.

In online voting mechanism each voter receives a unique ballot code. The ballot code has an arbitrary length and is generated randomly to help prevent manipulation. Online voting system mails virtual ballot papers, including the ballot code, to the voters before the election.
The voters can then use their email clients to return their votes to the voting server.

The voting server collects the votes and filters out duplicate and invalid votes. Each voter can then check her/his vote online to ensure that her vote has been counted correctly. This case study concerns a simplified system of the online voting system. The online voting system allows the voter at any place to vote. It also provides the specific time to vote. It stores the party symbols according to each party and count the result based on party symbols.

ANALYSIS

2. Requirements Specification:

The following are the functional and non-functional requirements for the system:

ID	Description	Type	Priority
R1	Online Voting System shall be maintain the details of voters	Functional	Must have
R2	Online Voting System shall be verify the voters and check the valid voter.	Functional	Must have
R3	Online Voting System shall be available for voter in specified time period.	Functional	Must have
R4	Online Voting System shall maintain the information about the candidate who is participating in the elections.	Functional	Must have
R5	Online Voting System shall provide the specific timing for the voters to vote.	Functional	Must have
R6	Online Voting System shall maintain the specified symbols of the candidates.	Functional	Must have
R7	Online Voting System shall count the voters from the voting system.	Functional	Must have
R8	Online Voting System the voter shall select the election.	Functional	Must have
R9	Online Voting System shall be easily understandable for any voters.	Non Functional	Could have
R10	Online Voting System shall provide the security from unauthorized voter.	Non Functional	Could have
R11	Online Voting System shall provide fast login in 5-10 sec.	Non Functional	Could have
R12	Online Voting System shall available to more voters to login at a time.	Non Functional	Could have
R13	Online Voting System will allows only ones to vote the voting system.	Non Functional	Could have

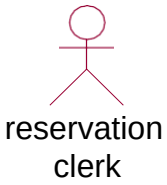
USECASE VIEW

3. IDENTIFICATION OF ACTORS

- Actors are NOT part of the system.
- Actors represent anyone or anything that interacts with (input to or receive output from) the system.
- An actor is someone or something that:
    - Interacts with or uses the system.
    - Provides input to and receives information from the system Is external to the system and has no control over the use cases.
- Actors are discovered by examining:
    - Who directly uses the system?
    - Who is responsible for maintaining the system?
    - External hardware used by the system.
    - Other systems that need to interact with the system.
- The needs of the actor are used to develop use cases. This insures that the system will be what the user expected.

Graphical Depiction

- An actor is a stereotype of a class and is depicted as a "stickman" on a use-case diagram.



Naming: The name of the actor is displayed below the icon.

Questions that help to identify actors

1. Who is interested in a certain requirement?
2. Where is the system used within the organization?
3. Who will benefit from the use of the system?
4. Who will supply the system with information, use this information, and remove this information?
5. Who will support and maintain the system?
6. Does the system use an external resource?
7. Does one person play several different roles?
8. Do several people play the same role?
9. Does the system interact with a legacy system?

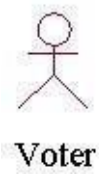
Using the above questions we have identified two actors in student attendance management and faculty evaluation application. They are

1 .Administrator

2. Voter

Admin: Administrator is a person who is responsible le for registration of voter ,preparation of ballot form , updating the number of votes and prepare the results.

Voter: The person who register for voting can vote in the particular period of time.



4. IDENTIFICATION OF USE-CASES AND SUB USE-CASES

Use case is a sequence of transactions performed by a system that yields a measurable result of values for a particular actor. The use cases are all the ways the system may be used.

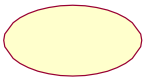
Graphical Depiction:

- The basic shape of a use case is an ellipse:

NewUseCase

Naming

- A use case may have a name, although it is typically not a simple name. It is often written as an informal text description of the actors and the sequences of events between objects. Use case names often start with a verb.
- The name of the use case is displayed below the icon.



booking

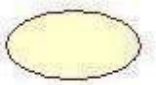
Questions that help to find use cases

1. What are the tasks of each actor?
2. Will any actor create, store, change, remove or read information in the system?
3. What use cases will create, store, change, remove, or read this information?
4. Will any actor need to inform the system about sudden, external changes?
5. Does any actor need to be informed about certain occurrences in the system?
6. What use cases will support or maintain the system?
7. Can all functional requirements be performed by the use cases?

By applying above questions to bus reservation system application the following use cases are identified .They are

1) Login: This usecase provides the voter to login into the system.

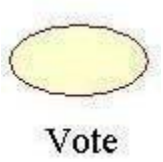
UML notation:



Login

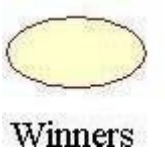
2) Voting: This use case provides to vote for a particular contestant.

UML notation:

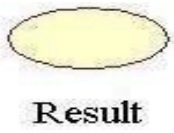


3) See the results: This use case is started by admin .It display the winners.

UML notation:

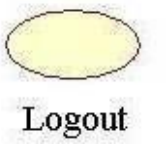


4) Publish the result: This use case is started by admin.It provides to publish the results. UML notation:



5) Logout: This use case provides the voter to logout into the system.

. UML notation:



5. FLOW OF EVENTS

Use case:	**Login/Logoff**
Id:	1
Brief description:	This use case describes how a user logs into the Online Voting System. The actors starting this use case are voter and the administrator of the system
Primary actor	voter, Administrator
Secondary actor	none
Pre condition:	None.
Flow of events:	1. The system validates the actor's password and logs him/her into the system. 2. The system displays the Main Form and the use case ends.
Post condition:	none
Alternate flow:	1.Invalid Name / Password If in the basic flow the system cannot find the name or the password is invalid, an error message is displayed. The actor can type in a new name or password or choose to cancel the operation, at which point the use case ends

Use case:	**Voting**
Id:	2
Brief description:	The voters can answer to the questions listed on the Internet or voting for a result on the basis of the items listed bellowing list box .
Primary actor	voter, Administrator
Secondary actor	none
Pre condition:	The voter login to the Online voting system so that he or she can do the voting
Flow of events:	1. Voter login into the voting system . 2. Voter select the thesis that the user want to do the voting . 3. Voter Click the result and finish the voting
Post condition:	The statistics of the voting system can only be used after other people do the voting .
Alternate flow:	none

Use case:	**Publish the Result**
Id:	4
Brief description:	The administrator of the system can publish the result of the system
Primary actor	voter, Administrator
Secondary actor	none
Pre condition:	1. The administrator login into the Online Voting System. 2. The voter has the authority to publish the result of the votingthesis
Flow of events:	1. Voter login into the Online Voting System. 2. Voter select the thesis of the online voting System. 3. The voter publish the result of the thesis of the project. 4.The voting result can be seen on the web by clicking the thesis ofthe system
Post condition:	The voter can see the result of the online voting system after the result is published.
Alternate flow:	1. The login voter doesn't have the authority to publish the Result so that the systemprovide the voter with an information that he or she is not capable of this function

6. USE-CASE DIAGRAM

Use-case diagrams graphically represent system behavior (use cases). These diagrams present a high level view of how the system is used as viewed from an outsider's (actor's) perspective. A use-case diagram may contain all or some of the use cases of a system.

A use-case diagram can contain:

- Actors ("things" outside the system)
- Use cases (system boundaries identifying what the system should do)
- Interactions or relationships between actors and use cases in the system including the associations, dependencies, and generalizations.

Use-case diagrams can be used during analysis to capture the system requirements and to understand how the system should work. During the design phase, you can use use-case diagrams to specify the behavior of the system as implemented.

RELATIONS:

Association Relationship:

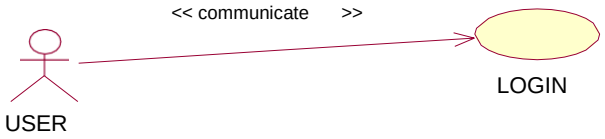
An association provides a pathway for communication. The communication can be between use cases, actors, classes or interfaces. Associations are the most general of all relationships and consequentially the most semantically weak. If two objects are usually considered independently, the relationship is an association

By default, the association tool on the toolbox is uni-directional and drawn on a diagram with a single arrow at one end of the association. The end with the arrow indicates who or what is receiving the communication.

Bi-directional association:

If you prefer, you can also customize the toolbox to include the bi-directional tool to the use-case toolbox.

In An ASSOCIATION Relationship, we can provide Stereotype COMMUNICATE also as shown below



Dependency Relationship:

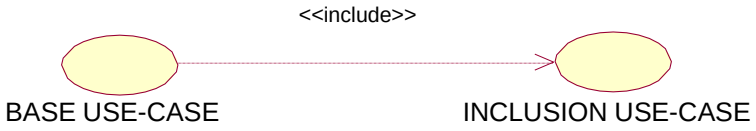
A dependency is a relationship between two model elements in which a change to one model element will affect the other model element. Use a dependency relationship to connect model elements with the same level of meaning. Typically, on class diagrams, a dependency relationship indicates that the operations of the client invoke operations of the supplier.

We can provide here

1. Include Relationship.
2. Extend Relationship

- There are two types of relationships that may exist between use cases: *include relationship* and *extend relationship.*
- Multiple use cases may share pieces of the same functionality. This functionality is placed in a separate use case rather than documenting it in every use case that needs it
- *Include* relationships are created between the new use case and any other use case that "uses" its functionality.

An include relationship is a stereotyped relationship that connects a base use case to an inclusion use case. An include relationship specifies how behavior in the inclusion use case is used by the base use case.



Extended Relationship:

An extend relationship is a stereotyped relationship that specifies how the functionality of one use case can be inserted into the functionality of another use case. Extend relationships between use cases are modeled as dependencies by using the Extend stereotype.

An *extend* relationship is used to show

- Optional behavior
- Behavior that is run only under certain conditions such as triggering an alarm
- Several different flows that may be run based on actor selection
- An *extend* relationship is drawn as a dependency relationship that points from the extension to the base use case

The extend relationship sample demonstrates how you can use an extend relationship to connect use cases. The sample illustrates two important aspects of extend relationships:

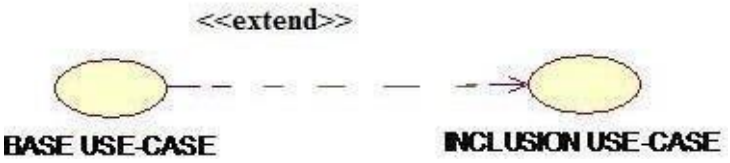
· An extend relationship shows optional functionality or system behavior.

· A base use case does not need to acknowledge any specific extended use cases Finally

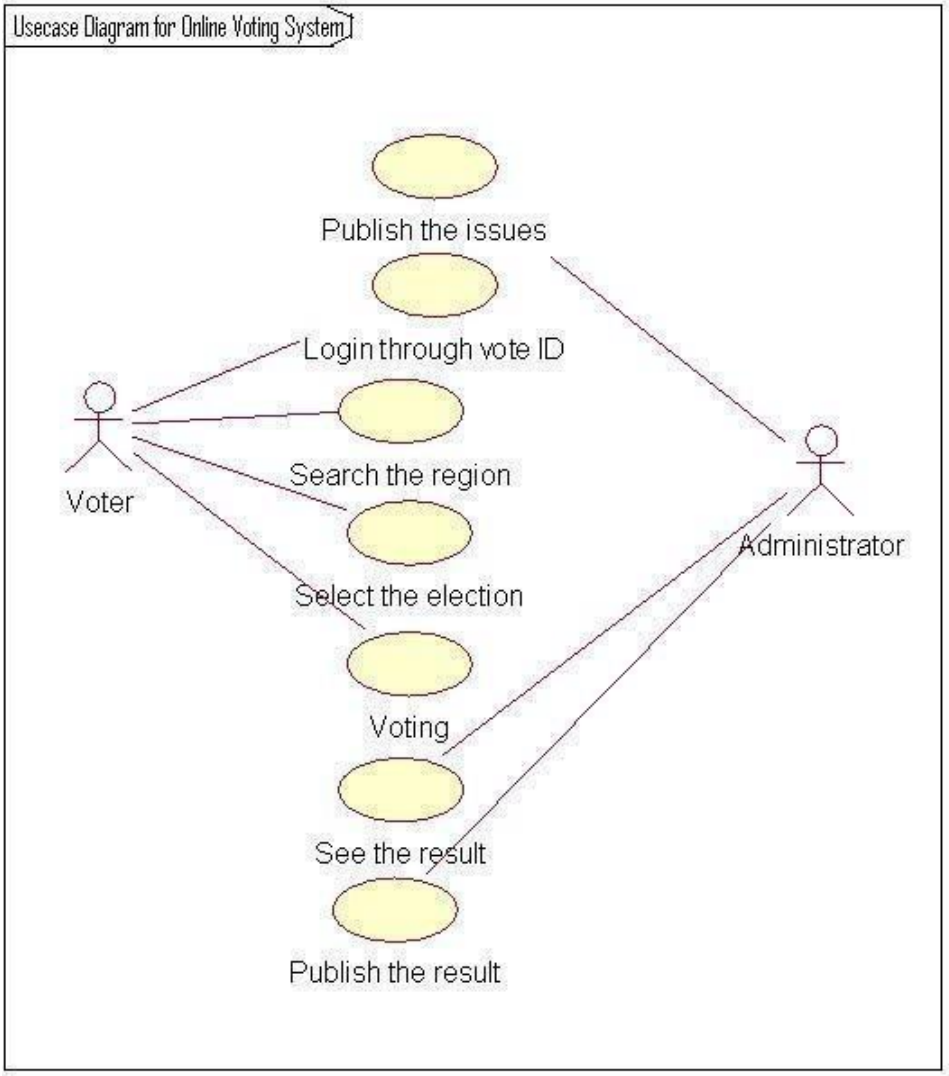
we can conclude

«extend» is used when you wish to show that a use case provides additional functionality that may be required in another use case.

«include» applies when there is a sequence of behavior that is used frequently in a number of use cases, and you want to avoid copying the same description of it into each use case in which it is used.



USECASE DIAGRAM FOR ONLINE VOTING SYSTEM



7. ACTIVITY DIAGRAM

Activity diagrams are “OO flowcharts”. They allow you to model a process as a collection of activities and transitions between those activities. Activity diagrams are really just special cases of state charts, where every state has an entry action that specifies some process or function that occurs when the state is entered.

An activity diagram can be attached to any modeling element for the purpose of modeling the behavior of that element. Activity diagrams are typically attached to:

- Use cases;
- Classes;
- Interfaces;
- Components;
- Nodes;
- Collaborations;
- Operations and methods.

Action states

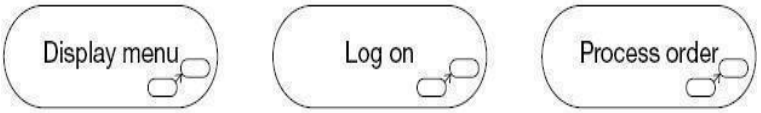
Activity diagrams contain action states and sub activity states. Action states are the finest granularity building block of activity diagrams, and represent actions – tasks that can’t be broken down into subtasks. Action states are represented as boxes with rounded ends. This is illustrated as

- Atomic – can’t be broken down into smaller pieces;
- Uninterruptible – once the piece of work starts it always progresses to the finish;
- Instantaneous – the work of an action state is generally considered to take an insignificant amount of time.

Each activity diagram has two special states – the start state and the stop state. The start state marks the beginning of the workflow, and the stop state marks the end – their symbols are shown as

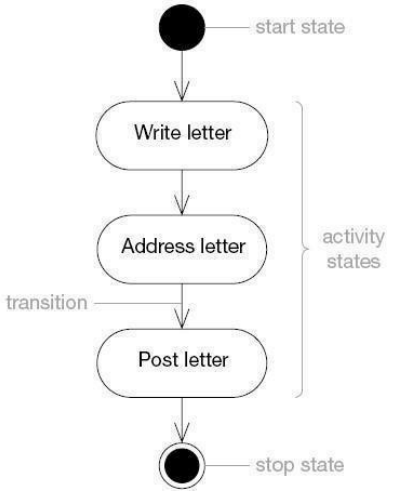
Sub activity

Sub activity states are non-atomic – they can be broken down into other sub activity and action states. They may be interrupted, and may take a finite amount of time. They have a special syntax shown in Figure below



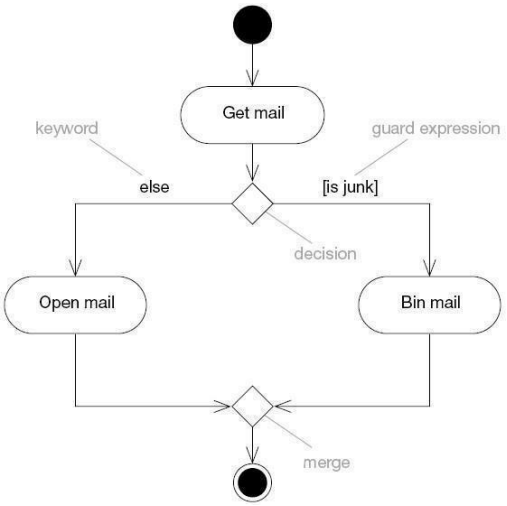
Transition

Whenever an action or sub activity state finishes its work, there is a transition out of the state into the next state. This is known as an automatic transition.



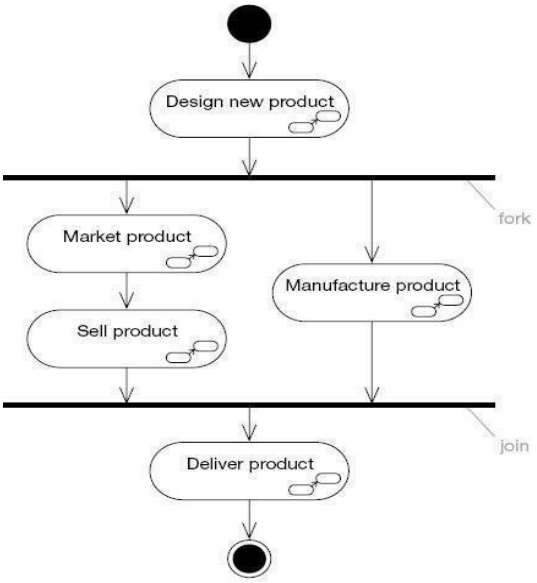
Decision

A decision specifies alternative paths based on some Boolean guard expression. The UML syntax for a decision is a diamond shape, just like in a normal flowchart. The same symbol is also used for a merge, when the two alternative paths reunite.



Forks and joins

Activity diagrams are very good for modeling concurrent flows of work. You can split a path into two or more concurrent flows using a fork, and then synchronize these concurrent flows using a join. Forks have exactly one incoming transition and two or more outgoing transitions. Joins have two or more incoming transitions and exactly one outgoing transition.



UML does not give any detailed semantics for swim lanes. You can therefore use them to partition activity diagrams in any way you like! Swim lanes are commonly used to represent:

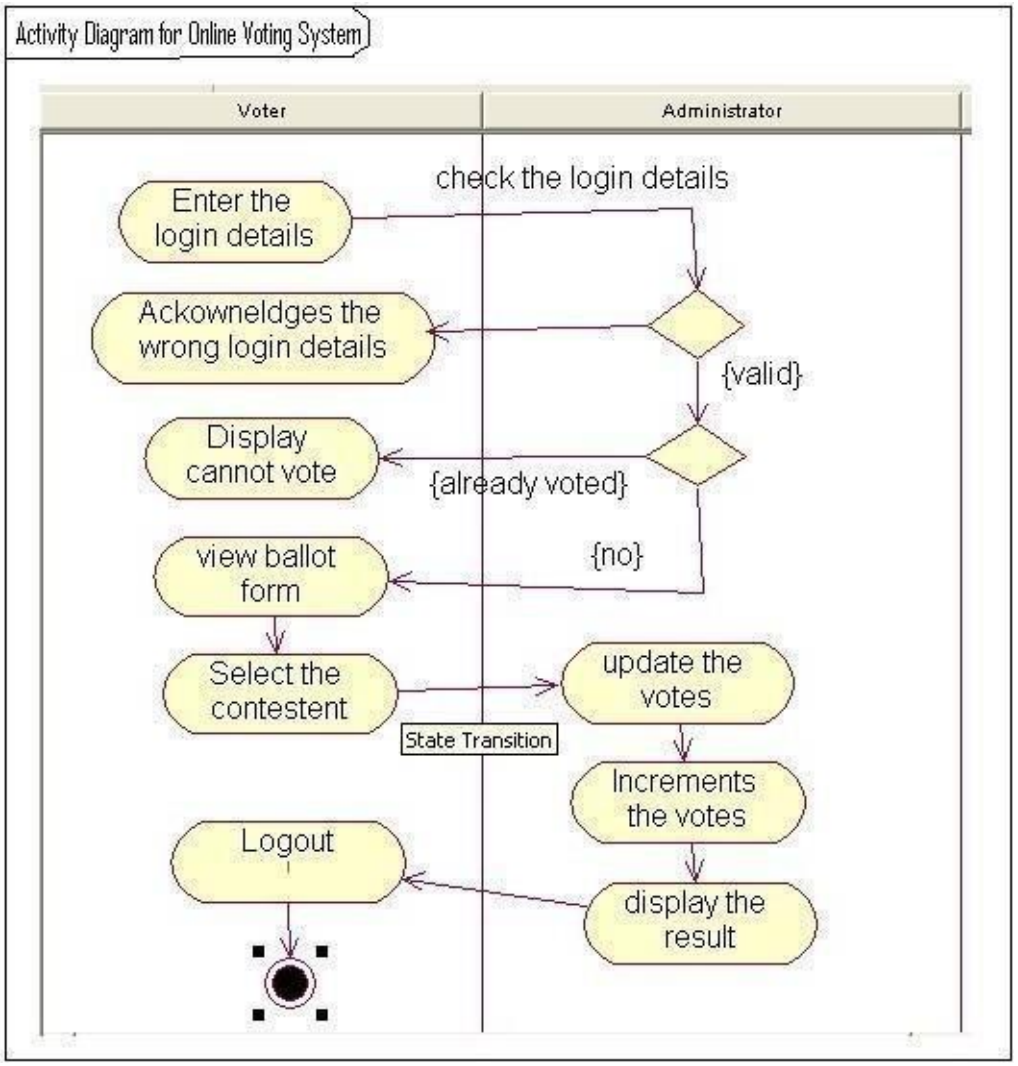
- Use cases;
- Classes;
- Components;
- Organizational units (in business modeling); Roles (in workflow modeling).

Signal

A signal is a way of representing a package of information that is communicated asynchronously between two objects. A signal event occurs when an object receives a signal.

A signal send is modeled on an activity diagram as a convex pentagon labeled with the name of the signal that is sent. The signal send has one input transition and one output transition. There is no action associated with this state; all it does is send the signal. You can draw a dashed arrow from the signal send to the object that receives the signal

ACTIVITY DIAGRAM FOR ONLINE VOTING SYSTEM:



8.IDENTIFICATION OF ANALYSIS CLASSES

The class diagram is fundamental to object-oriented analysis. Through successive iterations, it provides both a high level basis for systems architecture, and a low-level basis for the allocation of data and behavior to individual classes and object instances, and ultimately for the design of the program code that implements the system. So, it is important to identify classes correctly. However, given the iterative nature of the object-oriented approach, it is not essential to get this right on the first attempt itself.

APPROACHES FOR IDENTIFYING CLASSES:

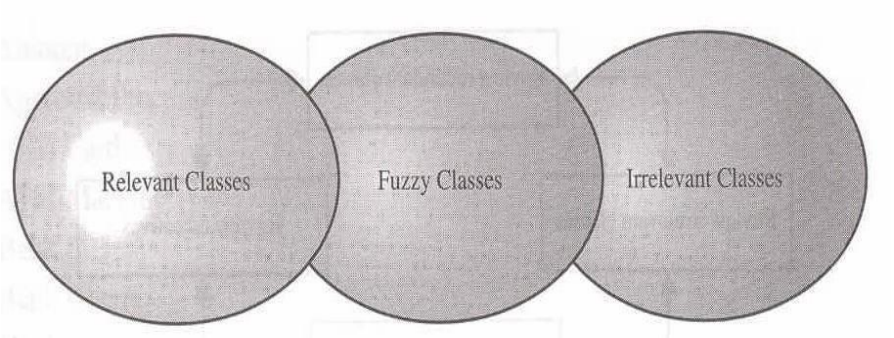
We have four alternative approaches for identifying classes:

1. The noun phrase approach;
2. The common class patterns approach;
3. The use- case driven TO sequence/collaboration modeling approach;
4. Class Responsibility collaboration cards (CRC) approach.

The first two approaches have been included to increase analysts understanding of the subject; the unified approach uses the use-case driven approach for identifying classes and understanding the behavior of objects. However, these can be combined to identify classes for a given problem.

1) NOUN PHRASE APPROACH.

In this method, analysts read through the requirements or use cases looking for noun phrases. Nouns in the textual description are considered to be classes and verbs to be methods of the classes All plurals are changed to singular, the nouns are listed, and the list divided into three categories relevant classes, fuzzy classes (the "fuzzy area," classes we are not sure about), and irrelevant classes as shown below.



Using the noun phrase strategy, candidate classes can be divided into three categories: Relevant classes, Fuzzy Area or Fuzzy classes (those classes that we are not sure about), and irrelevant classes.

It is safe to scrap the irrelevant classes, which either have no purpose or will be unnecessary. Candidate classes then are selected from the other two categories. Here identifying classes and developing a UML class diagramjust likeother activities is an iterative process. Depending on whether such object modeling is for the analysis or designphase of development, some classes may need to be added or removed from the model .Analyst must be able to formulate a statement of purpose for each candidate class; if not, simply eliminate it.

1. Identifying Tentative Classes:

The following are guidelines for selecting classes in an application:

- Look for nouns and noun phrases in the use cases.
- Some classes are implicit or taken from general knowledge.
- All classes must make sense in the application domain; avoid computer implementation classes refer them to the design stage.
- Carefully choose and define class names.

Identifying classes is an incremental and iterative process. This incremental and iterative nature is evident in the development of such diverse software technologies as graphical user interfaces, database standards, and even fourth-generation languages.

2. Selecting Classes from the Relevant and Fuzzy Categories:

The following guidelines help in selecting candidate classes from the relevant and fuzzy categories of classes in the problem domain.

a) Redundant classes. Do not keep two classes that express the same information. If more than one word is being used to describe the same idea, select the one that is the most meaningful in the context of the system. This is part of building a common vocabulary for the system as a whole. Choose your vocabulary carefully; use the word that is being used by the user of the system.

b) Adjectives classes. "Be wary of the use of adjectives. Adjectives can be used in many ways. An adjective can suggest a different kind of object, different use of the same object, or it could be utterly irrelevant. Does the object represented by the noun behave differently when the adjective is applied to it? If the use of the adjective signals that the behavior of the object is different, then make a newclass".

For example : Single account holders behave differently than Joint account holders, so the two should be classified as different classes.

c) Attribute classes: Tentative objects that are used only as values should be defined or restated as attributes and not as a class. For example, Client Status and Details of Client are not classes but attributes of the Client class.

d) Irrelevant classes: Each class must have a purpose and every class should be clearly defined and necessary. You must formulate a statement of purpose for each candidate class. If you cannot come up with a statement of purpose, simply eliminate the candidate class.

As this is an incremental process. Some classes will be missing; others will be eliminated or refined later. Unless you are starting with a lot of domain knowledge, you probably are missing more classes than you will eliminate. Although some classes ultimately may become super classes, at this stage simply identify them as individual, specific classes. Your design will go through many stages on its way to completion, and you will have adequate opportunity to revise it.

This refining cycles through the development process until you are satisfied with the results. Remember that this process (of eliminating redundant classes, classes containing adjectives, possible attributes, and irrelevant classes) is not sequential. You can move back and forth among these steps as often analysts likes.

2) COMMON CLASS PATTERNS APPROACH

The second method for identifying classes is using common class patterns, which is based on a knowledge base of the common classes. The following patterns are used for finding the candidate class and object:

a) Concept class: A concept is a particular idea or understanding that we have of our world. The concept class encompasses principles that are not tangible but used to organize or keep track of business activities or communications.

Example: Performance is an example of concept class object.

b) Events class: Events classes are points in time that must be recorded. Things happen, usually to something else at a given date and time or as a step in an ordered sequence. Associated with things remembered are attributes (after all, the things to remember are objects) such as who, what, when, where, how, or why.

Example: Landing, interrupt, request, and order are possible events.

c) Organization class: An organization class is a collection of people, resources, facilities, or groups to which the users belong; their capabilities have a defined mission, whose existence is largely independent of the individuals.

Example: An accounting department might be considered a potential class.

d) People class (also known as person, roles, and roles played class):

The people class represents the different roles users play in interacting with the application.

Example: Employee, client, teacher, and manager are examples of people.

e) Places class: Places are physical locations that the system must keep information about.

Example: Buildings, stores, sites, and offices are examples of places.

3) USE-CASE DRIVEN APPROACH:

IDENTIFYING CLASSES AND THEIR BEHAVIORS THROUGH SEQUENCE/COLLABORATION MODELING

One of the first steps in creating a class diagram is to derive from a use case, via a collaboration (or collaboration diagram), those classes that participate in realizing the use case. Through further analysis, a class diagram is developed for each use case and the various use case class diagrams are then usually assembled into a larger analysis class diagram. This can be drawn first for a single subsystem or increment, but class diagrams can be drawn at any scale that is appropriate, from a single use case instance to a large, complex system.

Identifying the objects involved in collaboration can be difficult at first, and takes some practice before the analyst can feel really comfortable with the process. Here collaboration (i.e. the set of classes that it comprises) can be identified directly for a use case, and that, once the classes are known, the next step is to consider the interaction among the classes and so build a collaboration diagram.

The next step in the development of a requirements model is usually to produce a class diagram that corresponds to each of the collaboration diagrams. The class diagram that corresponds to the use case E-Pay is shown below.

Collaboration diagrams are obtained by result of reasonably careful analysis, the transition is not usually too difficult. The similarities & differences B/W Collaboration and class diagrams are:

First, consider the similarities:

Both show class or object symbols joined by connecting lines. In general, a class diagram has more or less the same structure as the corresponding collaboration diagram.

Next, the differences are:

1. The difference is that an actor is almost always shown on a collaboration diagram, but not usually shown on a class diagram. This is because the collaboration diagram represents a particular interaction and the actor is an important part of this interaction. However, a class diagram shows the more enduring structure of associations among the classes,and frequently supports a number of different interactions that may represent several different use cases.
2. A collaboration diagram usually contains only object instances, while a class diagram usually contains only classes.
3. The connections between the object symbols on a collaboration diagram symbolize links between objects, while on a class diagram the corresponding connections stand for associations between classes.
4. A collaboration diagram shows the dynamic interaction of a group of objects and thus every link needed for message passing is shown. The labeled arrows alongside the links represent messages between objects. On a class diagram, the associations themselves are usually labeled, but messages are not shown.
5. Finally, any of the three stereotype symbols can be used on either diagram; there are also differences in this notation.

When the rectangular box variant of the notation is used in a collaboration diagram it represents object instances rather than classes, is normally undivided and contains only the class name .On a class diagram, the symbol is usually divided into three compartments that contain in turn the class name, its attributes and its operations.

9.IDENTIFICATION OF RESPONSIBILITIES OF CLASSES

Class Responsibility collaboration Cards (CRC Cards)

At the starting, for the identification of classes we need to concentrate completely on uses cases. A further examination of the use cases also helps in identifying operations and the messages that classes need to exchange. However, it is easy to think first in terms of the overall responsibilities of a class rather than its individual operations.

A *responsibility* is a high level description of something a class can do. It reflects the knowledge or information that is available to that class, either stored within its own attributes or requested via collaboration with other classes, and also the services that it can offer to other objects. A responsibility may correspond to one or more operations. It is difficult to determine the appropriate responsibilities for each class as there may be many alternatives that all appear to be equally justified.

Class Responsibility Collaboration (CRC) cards provide an effective technique for exploring the possible ways of allocating responsibilities to classes and the collaborations that are necessary to fulfill the responsibilities.

CRC cards can be used at several different stages of a project for different purposes.

1. They can be used early in a project to help the production of an initial class diagram 2. To develop a shared understanding of user requirements among the members of the team.

3. CRCs are helpful in modeling object interaction.

The format of a typical CRC card is shown below

Class Name:	
Responsibilities	Collaborations
Responsibilities of class are listed in this section	*Collaborations with other classes are listed here, together with a brief description of the purpose of the collaboration*

CRC cards are an aid to a group role-playing activity. Index cards are used in preference to pieces of paper due to their robustness and to the limitations that their size (approx. 15cm x 8cm) imposes on the number of responsibilities and collaborations that can be effectively allocated to each class. A class name is entered at the top of each card and responsibilities and collaborations are listed underneath as they become apparent. For the sake of clarity, each collaboration is normally listed next to the corresponding responsibility.

From a UML perspective, use of CRC cards is in analyzing the object interaction that is triggered by a particular use case scenario. The process of using CRC cards is usually structured as follows.

1. Conduct a session to identify which objects are involved in the use case.
2. Allocate each object to a team member who will play the role of that object.
3. Act out the use case: This involves a series of negotiations among the objects to explore how responsibility can be allocated and to identify how the objects can collaborate with each other.
4. Identify and record any missing or redundant objects.

Before beginning a CRC session it is important that all team members are briefed on the organization of the session and a CRC session should be preceded by a separate exercise that identifies all the classes for that part of the application to be analyzed.

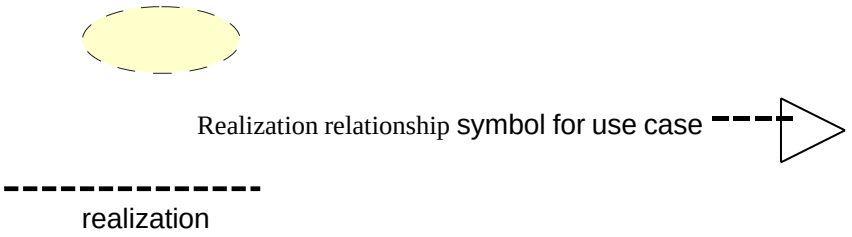
The team members to whom these classes are allocated can then prepare for the role playing exercise by considering in advance a first-cut allocation of responsibilities and identification of collaborations. Here, it is important to ensure that the environment in which the sessions take place is free from interruptions and free for the flow of ideas among team members.

During a CRC card session, there must be an explicit strategy that helps to achieve an appropriate distribution of responsibilities among the classes. One simple but effective approach is to apply the rule that each object should be as lazy as possible, refusing to take on any additional responsibility unless instructed to do so by its fellow objects.

During a session conducted according to this rule, each role player identifies the object that they feel is the most appropriate to take on each responsibility, and attempts to persuade that object to accept the responsibility. For each responsibility that must be allocated, one object is eventually persuaded by the weight of rational argument to accept it. This process can help to highlight missing objects that are not explicitly referred to by the use case description. When responsibilities can be allocated in several different ways it is useful to role-play each allocation separately to determine which is the most appropriate. The aim normally is to minimize the number of messages that must be passed and their complexity, while also producing class definitions that are cohesive and well focused.

10. USE-CASE REALIZATIONS

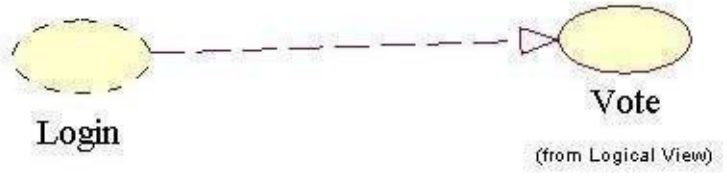
Use case realization is nothing but an instance of a use case which involves the identification of a possible set of classes, together with an understanding of how those classes might interact to deliver the functionality of the use case. The set of classes is known as collaboration.
In the UML, use case realizations are drawn as dashed ovals and relation ship symbols are as shown below.



A use case realization is a graphic sequence of events, also referred as a scenario or an instance of a use case. These realizations or scenarios are represented using either a sequence or collaboration diagrams. Use case Realization can be provided using Sequence diagrams, Collaboration diagrams and Class diagrams. These diagrams can also be given by considering any scenario from the system or sub-system

The following are the use case realizations in our project:

Voting:



11. SEQUENCE DIAGRAM

A sequence diagram is a graphical view of a scenario that shows object interaction in a time based sequence. Sequence diagrams establish the roles of objects and help provide essential information to determine class responsibilities and interfaces.

In Sequence diagram the vertical dimension represents time and all objects involved in the interaction are spread horizontally across the diagram.

Time normally proceeds down the pages. However, a sequence diagram may be drawn with a horizontal time axis if required, and in this case, time proceeds from left o right across he page. Each object is represented by a vertical dashed line, called a *lifeline*, with an object symbol at the top. A message is shown by a solid horizontal arrow from one lifeline to another and is labeled with the message name. Each message name may optionally be preceded by a sequence number that message number that represents the sequence in which the messages are sent, but this is not usually necessary on a sequence diagram since the message sequence is already conveyed by theirrelative positions along time axis.

Steps:

1. An object is shown as a box at the top of a dashed vertical line. Object names can be specific (e.g., Algebra 101, Section 1) or they can be general (e.g., a course offering). Often, an anonymous object (class name may be used to represent any object in the class.)
2. Each message is represented by an Arrow between the lifelines of two objects. The order in which these messages occur is shown top to bottom on the page. Each message is labeled with the message name.

Purpose of sequence diagrams:

- A sequence diagram is an interaction diagram that emphasizes the time ordering of messages.
- The sequence diagram is used primarily to show the interactions between objects in the sequential order that those interactions occur.
- One of the primary uses of sequence diagrams is in the transition from requirements expressed as use cases to the next and more formal level of refinement. Use cases are often refined into one or more sequence diagrams.

- In addition to their use in designing new systems, sequence diagrams can be used to document how objects in an existing (call it "legacy") system currently interact. This documentation is very useful when transitioning a system to another person or organization.
- The main purpose of a sequence diagram is to define event sequences that result in some desired outcome. The focus is less on messages themselves and more on the order in which messages occur; nevertheless, most sequence diagrams will communicate what messages are sent between a system's objects as well as the order in which they occur.
- The diagram conveys this information along the horizontal and vertical dimensions: the vertical dimension shows, top down, the time sequence of messages/calls as they occur, and the horizontal dimension shows, left to right, the object instances that the messages are sent.

ELEMENTS OF SEQUENCE DIAGRAMS:

There are mainly five elements in sequence diagrams. Three of them are common to the two interaction diagrams and two are for sequential diagrams. They are-

- Objects
- Links
- Messages
- Focus of control
- Object life line

An **object** is a concrete manifestation of a class to which a set of operations can be applied and which has a state that stores the effects of the operations. Objects are instances of classes.

A **link** is a semantic connection among objects. In general, a link is an instance of an association. Whenever a class has an association to another class, there may be a link between the instances of the two classes; whenever there is a link between two objects, one object can send a message to the other object.

A **message** is the specification of a communication among objects that conveys information with expectation that activity will ensure. The receipt of message instance may be considered an instance of an event. When a message is passed, the action that results is an executable statement. An action may result in a change in state.

The **focus of control** is a tall, think rectangle that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure.

An **object lifeline** is a vertical dashed line that represents the existence of an object over a period of time. Most objects that appear in an interaction diagram will be in existence of an object over a period of time. Most objects that appear in an interaction diagram will be in existence for the duration of the interaction.

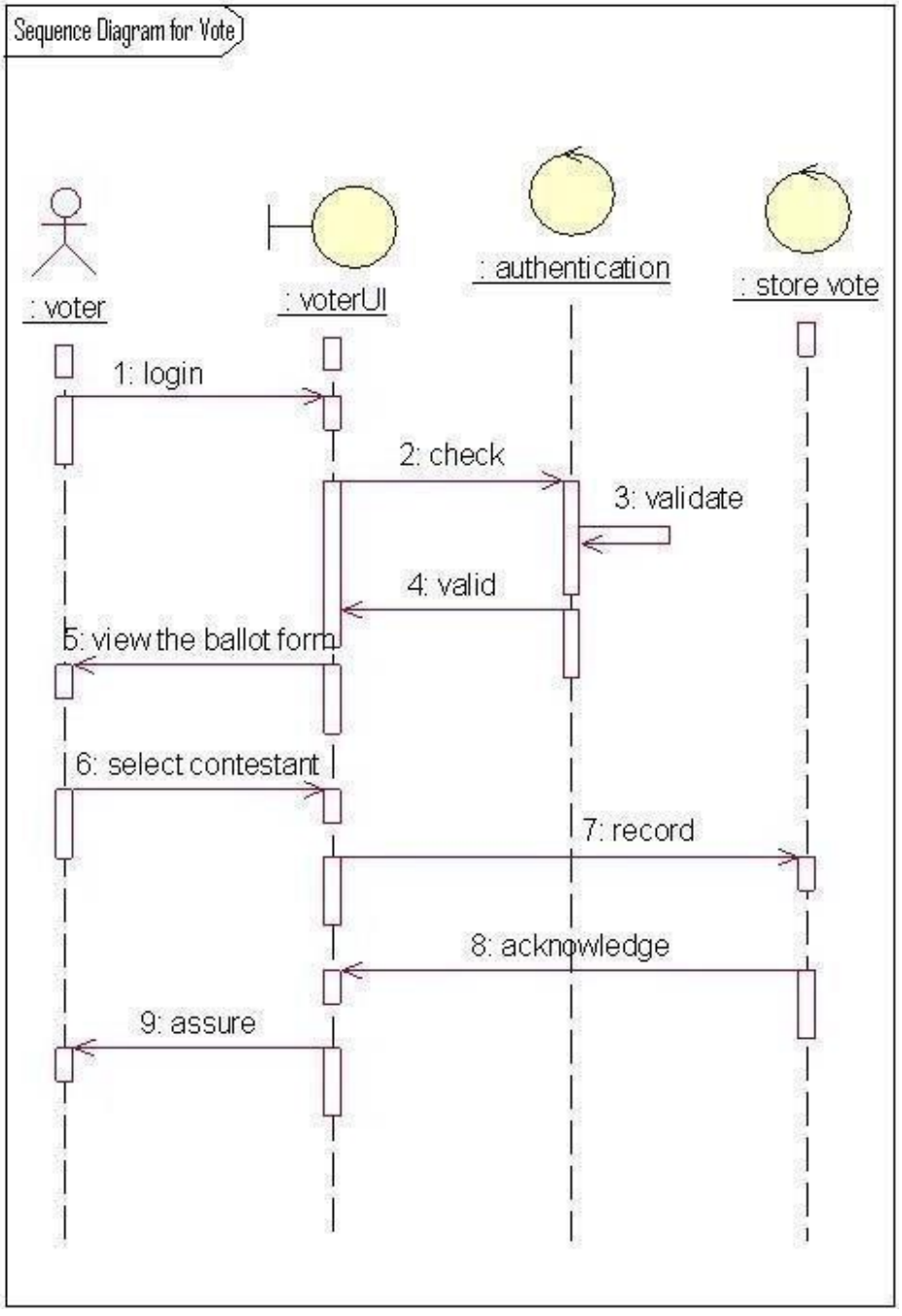
Message to self: A message to self is a tool that sends a message from one object back to the same object. It does not involve other objects because the message returns to the same object.
The sender of a message is the same as the receiver.

Sequence diagram for Admin:

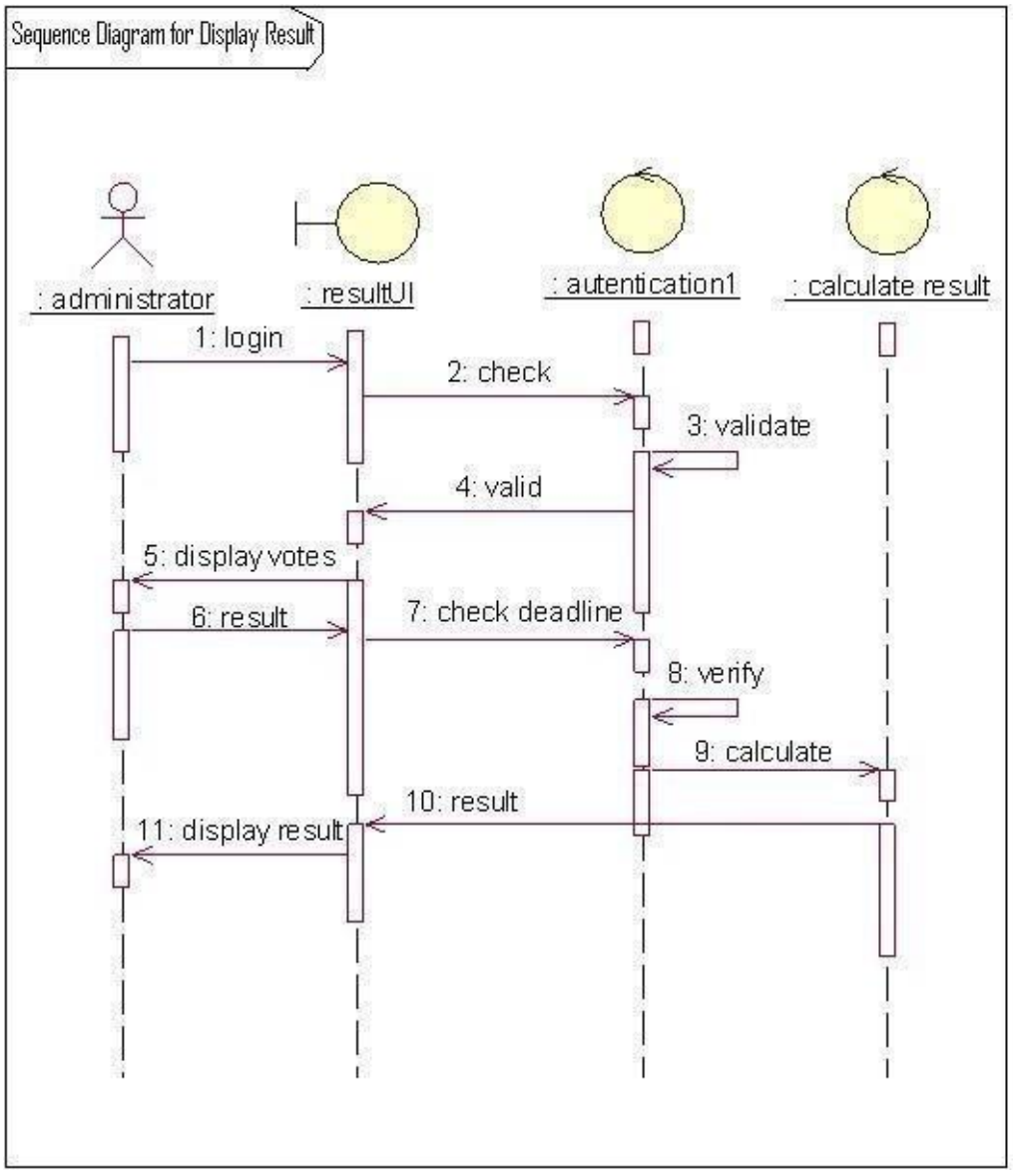
1. The admin enter the user name and password 2.
He will start the user interface
3 .the user interface displays a set of options
4. Admin can add/delete a student
5. Admin can also update the student information
6. The user displays that the student info is added/deleted or updated.
7. Admin again start interface to add attendance record to database.
8. The records will be added by specifying regd.no, sec, year of the student.
9. The user interface displays the records uploaded.
10. The Admin can view the evaluation records submitted by students.

Sequence Diagram for Online Voting System

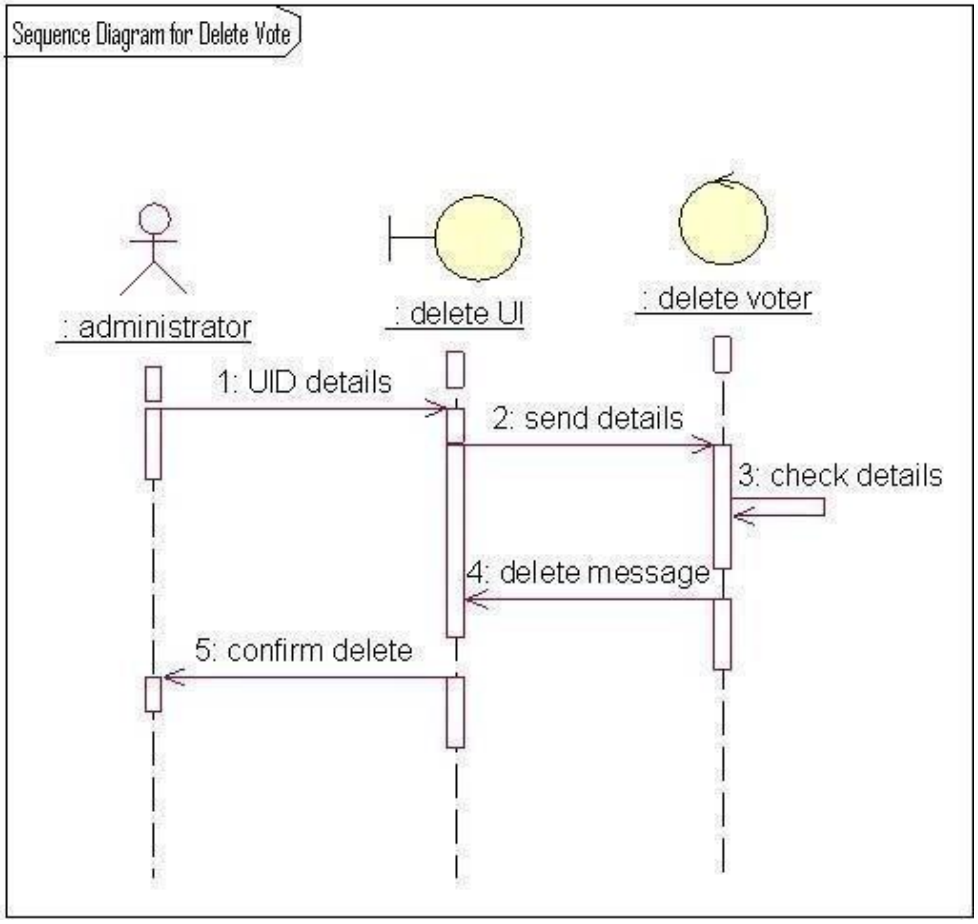
SEQUENCE DIAGRAM FOR VOTE:



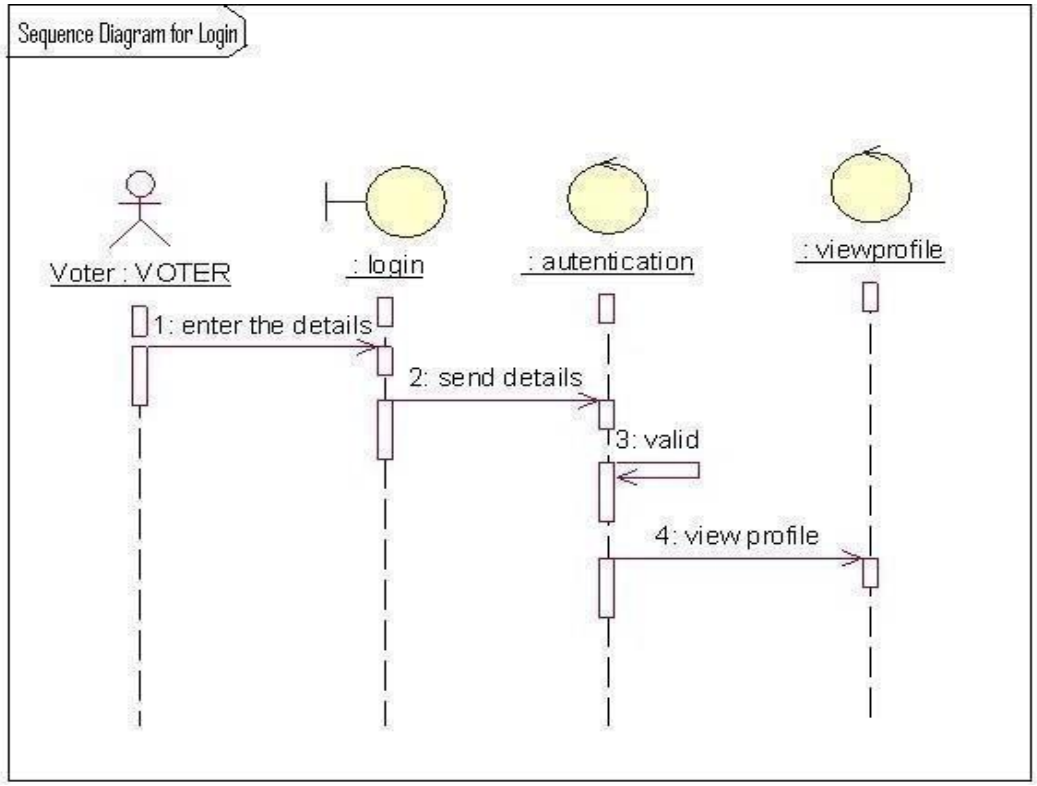
SEQUENCE DIAGRAM FOR DISPLAY RESULT:



SEQUENCE DIAGRAM FOR DELETE VOTE:



SEQUENCE DIAGRAM FOR LOGIN:



12. COLLABORATION DIAGRAM

A collaboration diagram is an alternate way to show a scenario. This type of diagram shows object interactions organized around the objects and their links to each other. A collaboration diagram contains:

- Objects drawn as rectangles
- Links between objects shown as lines connecting the linked objects
- Messages shown as text and an arrow that points from the client to the supplier Message labels in collaboration diagrams:

Messages on a collaboration diagram are represented by a set of symbols that are the same as those used in a sequence diagram, but with some additional elements to show sequencing and recurrence as these cannot be inferred from the structure of the diagram. Each message label includes the message signature and also a sequence number that reflects call nesting, iteration, branching, concurrency and synchronization within the interaction.

The formal message label syntax is as follows: [Predecessor] [guard-condition] sequenceexpression [return-value ':='] message-name' (' [argument-list] ') '

A predecessor is a list of sequence numbers of the messages that must occur before the current message can be enabled. This permits the detailed specification of branching pathways. The message with the immediately preceding sequence number is assumed to be the predecessor by default, so if an interaction has no alternative pathways the predecessor list may be omitted without any ambiguity. The syntax for a predecessor is as follows:

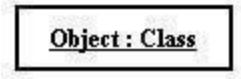
Sequence-number { ',' sequence-number} '*1*'

The 'l' at the end of this expression indicates the end of the list and is only included when an explicit predecessor is shown. *Guard conditions* are written in Object Constraint Language (OCL), and are only shown where the enabling of a message is subject to the defined condition.
A guard condition may be used to represent the synchronization of different threads of control.

Notations used:

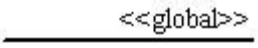
Class roles

Class roles describe how objects behave. Use the UML object symbol to illustrate class roles, but don't list object attributes.



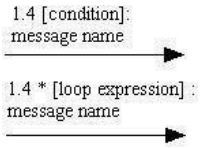
Association roles

Association roles describe how an association will behave given a particular situation. You can draw association roles using simple lines labeled with stereotypes.

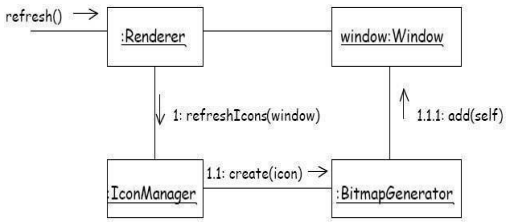


Messages

Unlike sequence diagrams, collaboration diagrams do not have an explicit way to denote time and instead number messages in order of execution. Sequence numbering can become nested using the Dewey decimal system. For example, nested messages under the first message are labeled 1.1, 1.2, 1.3, and so on. The condition for a message is usually placed in square bracketsimmediately following the sequence number. Use a * after the sequence number to indicate a loop.



Example



Types of Message Flows

Simple: Indicates flow of control - sender waits until receiver processes message, shown by an 'empty' arrow

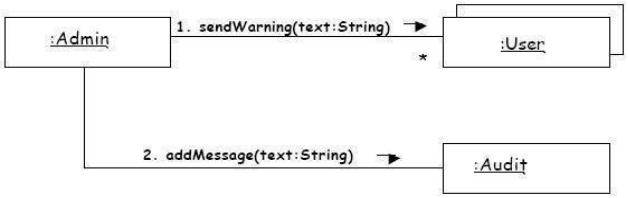
Synchronous: Indicates nested flow of control, used to ensure that state cannot be compromised by external factors shown by a 'filled' arrow.

Ex: not interrupted by the operating system.

Asynchronous: A signal from one object to another, shown as half an empty arrow

Multiplicity in Collaboration Diagrams

A server object needs to send a message to each user logged in, and another message to the audit object (which keeps a track of messages sent).



Instance form collaboration diagrams

The instance form collaboration diagram shows the collaboration between classifier instances. These instances are joined by links. Figure 1 shows the instance form of Figure 2

Figure 1

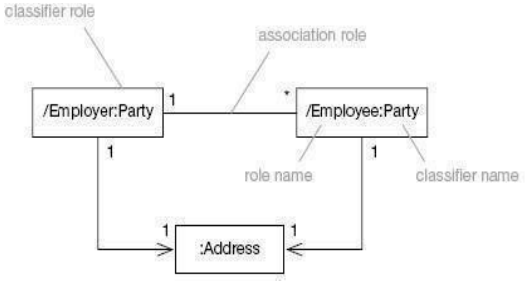
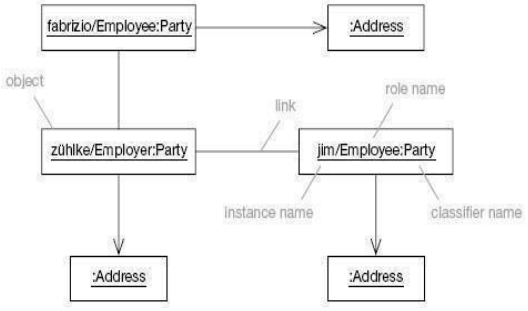
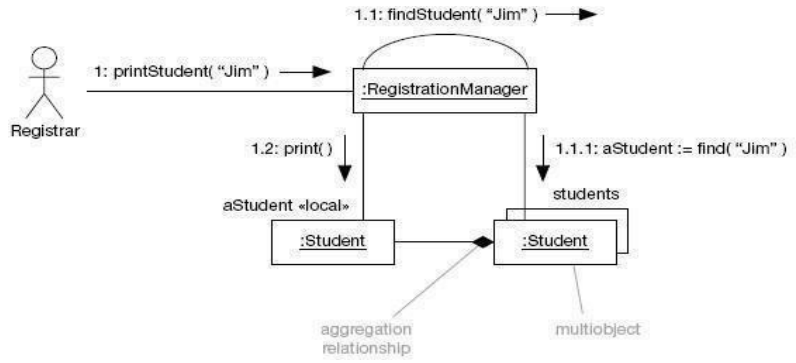


Figure2



Multiobjects

A multiobject represents a set of objects – it provides a way to represent a collection of objects in collaboration diagrams. Messages sent to the multiobject go to the set and *not* to any individual object.



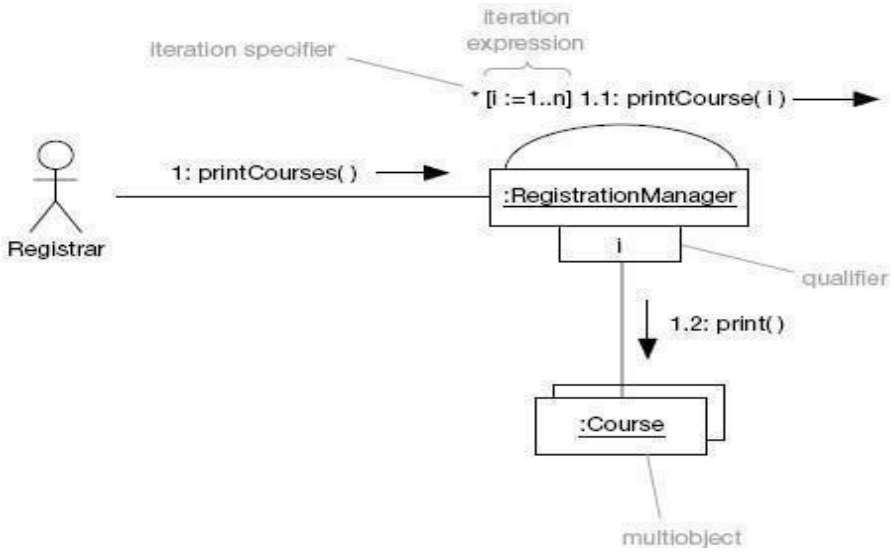
Iteration

You show iteration by prefixing the sequence number with the iteration specified (*) and an optional iteration expression. There is no formal UML syntax for the iteration expression, and any expression that is readable and makes sense will do.

Some common iteration expressions are given in Table given below

Iteration expression	Semantics
[i := 1..n]	Iterate n times
[i := 1..7]	Iterate 7 times
[while (some Boolean expression)]	Iterate while the Boolean expression is true
[until (some Boolean expression)]	Iterate until the Boolean expression is true
[for each (expression which evaluates to a collection of objects)]	Iterate over the contents of a collection of objects

Example:

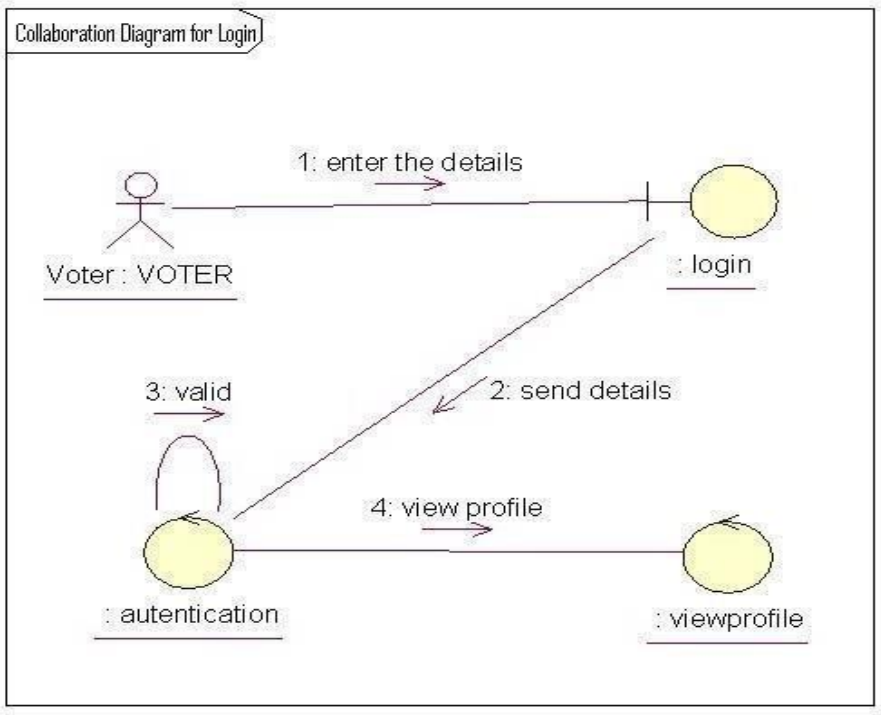


Difference between sequence and collaboration diagrams

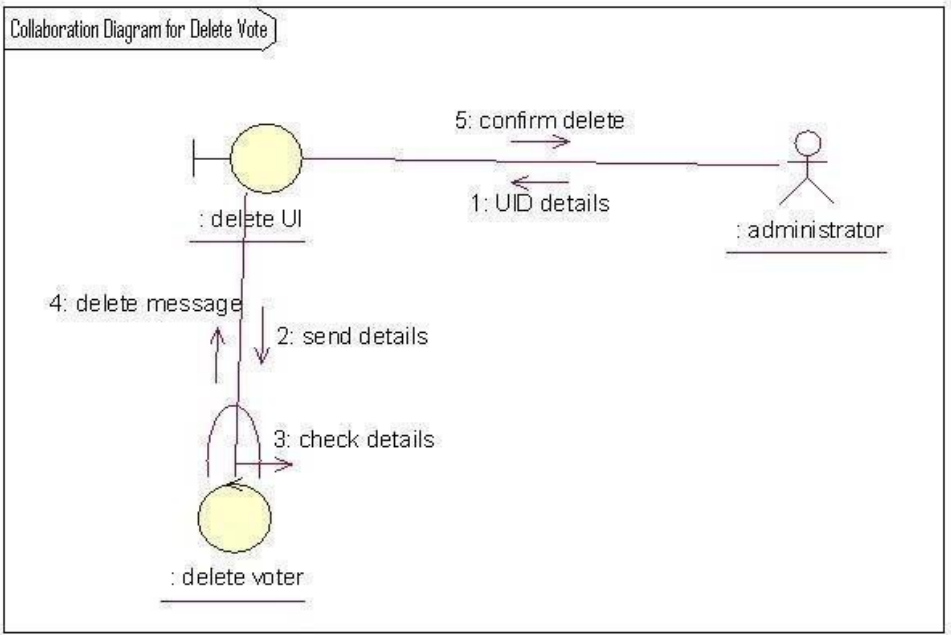
- Sequence diagrams are closely related to collaboration diagrams and both are alternate representations of an interaction.
- Sequence diagrams show time-based object interaction while collaboration diagrams show how objects associate with each other.
- A sequence diagram is a graphical view of a scenario that shows object interaction in a time based sequence
- A collaboration diagram shows object interactions organized around the objects and their links to each other.

Collaboration Diagrams for Online Voting System

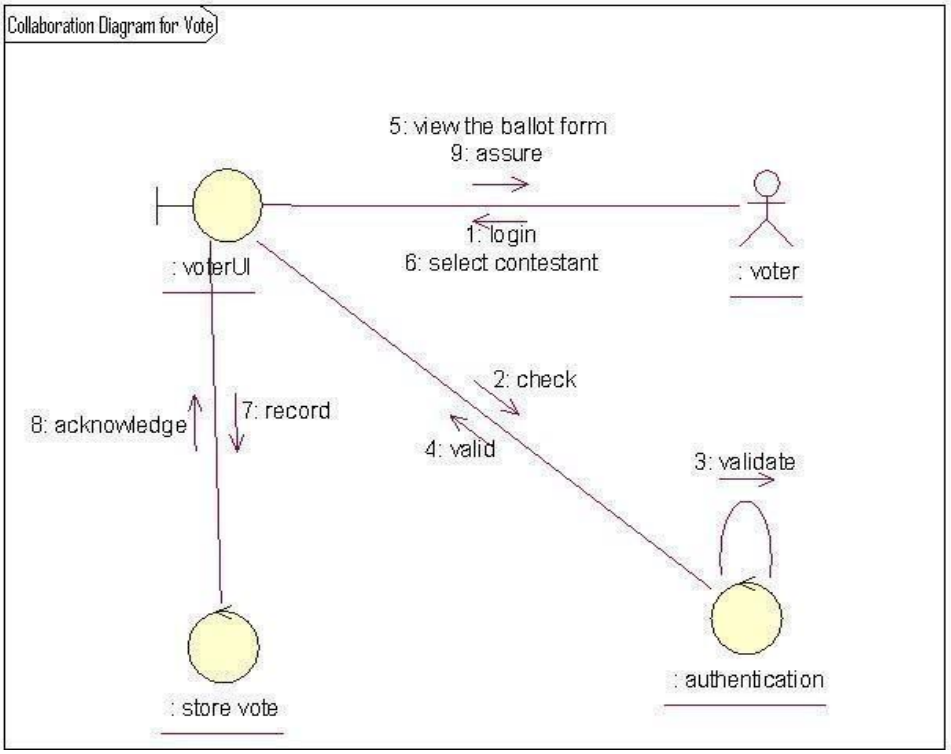
COLLABORATION DIAGRAM FOR LOGIN:



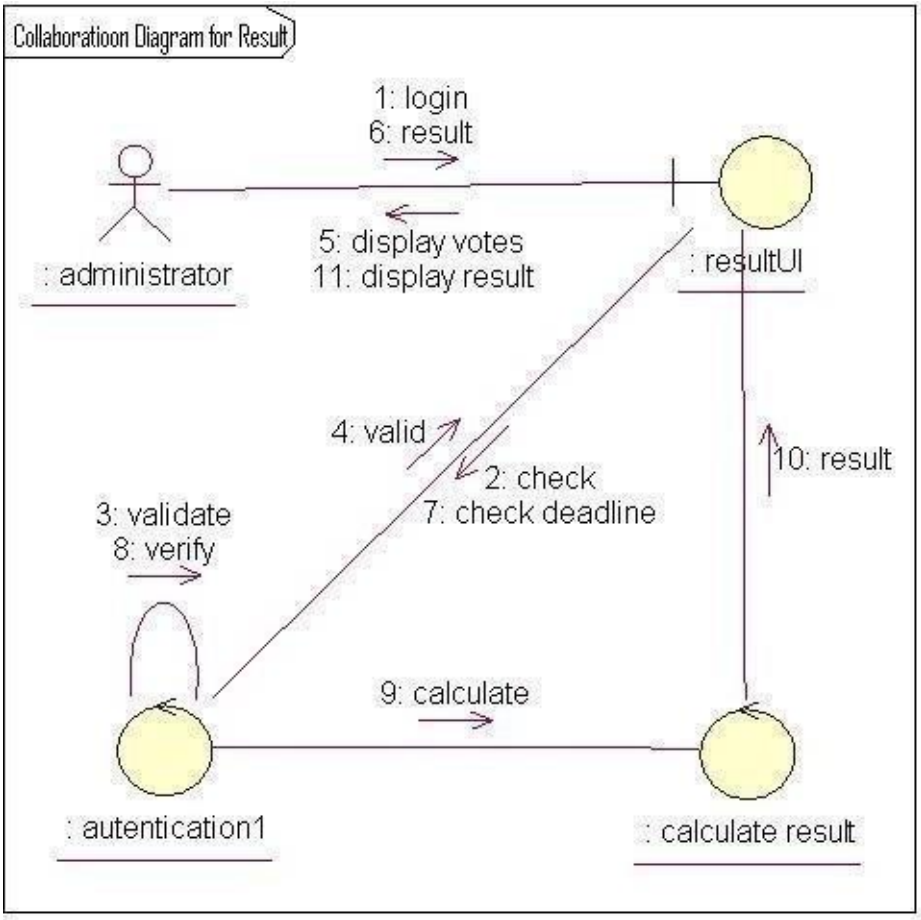
COLLABORATION DIAGRAM FOR DELETE VOTE:



COLLABORATION DIAGRM FOR VOTE



COLLABORATION DIAGRAM FOR RESULT:



13. IDENTIFICATION OF METHODS AND ATTRIBUTES OF CLASSES

Attributes

Attributes are part of the essential description of a class. They belong to the class, unlike objects, which instantiate the class. Attributes are the common structure of what a member of the class can 'know'. Each object will have its own, possibly unique, value for each attribute.

Operations

Operations are the elements of common behavior shared by all instances of a class. They are actions that can be carried out by or carried on, an object. The classes modeled during requirementsanalysis represent real-world things and concepts, so their operations can be said to represent aspects of the behavior of the same things and concepts. Another way of defining operations, operations are services that objects may be asked to perform by other objects.

An operation is a specification for some aspect of the behavior of a class. Here Operations are eventually implemented by *methods*, and what a method actually does on any given occasion may be constrained by the value of object attributes and links when the method is invoked.

List of attributes that are identified in our application are:

- Voter id
- Voter name
- authenticate
- store voter details
- add voter
- delete voter
- Calculate Result
- Announce result

Responsibilities that are identified for each class in our application:

Voter: This is an actor class. The voter search the participants in the election. Vote the desired contestant.

Administrator: Administrator is responsible for verifying the voter details. **Seller**:

It is an entity class. He submits the products to

14. IDENTIFICATION OF RELATIONSHIPS AMONG CLASSES

Need for Relationships among Classes

All systems are made up of many classes and objects. System behavior is achieved through the collaborations of the objects in the system.

For example, a passenger can perform reservation operation by submitting form to reservation clerk. This is often referred to as an object sending a message to another object. Relationships provide the medium or tool for object interaction. Two types of relationships in CLASS diagram are:

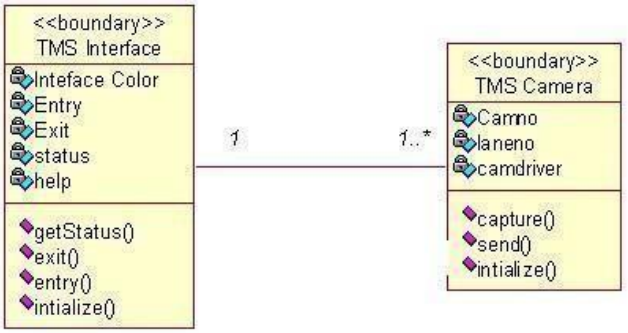
1. Associations Relationship
2. Aggregations Relationship

1. Association Relationship:

An association is a bidirectional semantic connection between classes. It is not a data flow as defined in structured analysis and design data may flow in either direction across the association. An association between classes means that there is a link between objects in the associated classes.

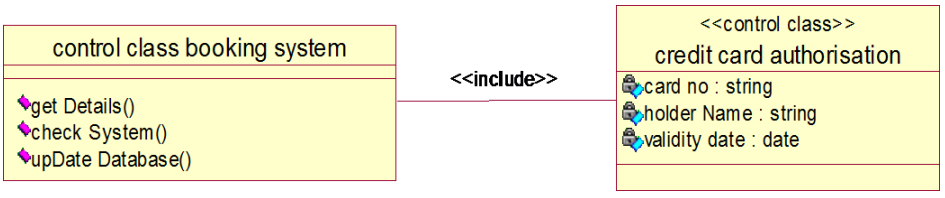
For example, an association between the Searching system class and the Airline database means that objects in the class searching system are connected to objects in the Airline database.

Association Relationship with Multiplicity



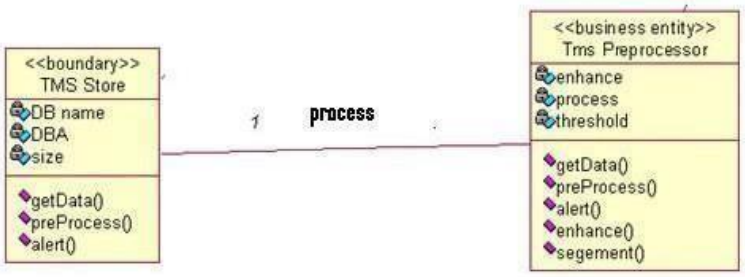
2. Aggregation Relationship:

An aggregation relationship is a specialized form of association in which a whole is related to its part(s). Aggregation is known as a "part-of" or containment relationship. The UML notation for an aggregation relationship is an association with a diamond next to the class denoting the aggregate (whole), as shown below:



3. Naming Relationship: _

An association may be named. Usually the name is an active verb or verb phrase that communicates the meaning of the relationship. Since the verb phrase typically implies a reading direction, it is desirable to name the association so it reads correctly from left to right or top to bottom. The words may have to be changed to read the association in the other direction. It is important to note that the name of the association is optional.



Role Names:

The end of an association where it connects to a class is called an association role. Role names can be used instead of association names.

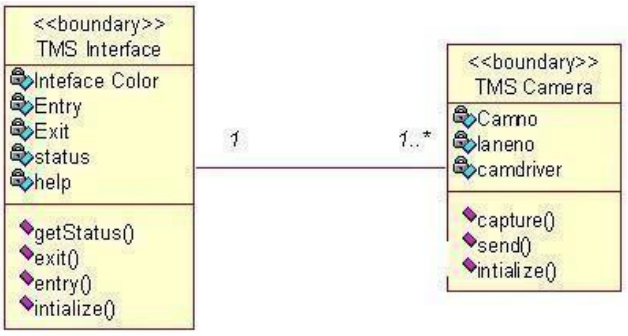
A role name is a noun that denotes how one class associates with another. The role name is placed on the association near the class that it modifies, and may be placed on one or both ends of an association line.

- It is not necessary to have both a role name and an association name. >•
- Associations are named or role names are used only when the names are needed for clarity.

Multiplicity Indicators:

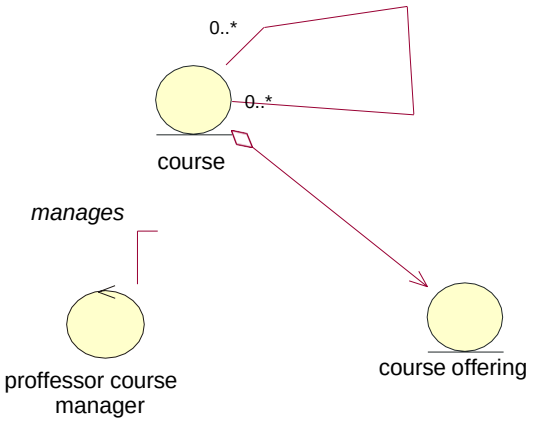
Although multiplicity is specified for classes, it defines the number of objects that participate in a relationship. Multiplicity defines the number of objects that are linked to one another. There are two multiplicity indicators for each association or aggregation one at each end of the line. Some common multiplicity indicators are

1	Exactly one
0.. *	Zero or more
1... *	One or more
0 .. 1	Zero or one
5 .. 8	Specific range (5, 6, 7, or 8)
4 .. 7,9	Combination (4, 5, 6, 7, or 9)



Reflexive Relationships:

Multiple objects belonging to the same class may have to communicate with one another. This is shown on the class diagram as a reflexive association or aggregation. Role names rather than association names typically are used for reflexive relationships.



15. UML CLASS DIAGRAM

- Class diagrams are created to provide a picture or view of some or all of the classes in the model.
- The main class diagram in the logical view of the model is typically a picture of the packages in the' system. Each package also has its own main class diagram, which typicallydisplays the "public" classes of the package.

A class diagram is a picture for describing generic descriptions of possible systems. Class diagrams and collaboration diagrams are alternate representations of object models. Class diagrams contain icons representing classes, packages, interfaces, and their relationships. You can create one or more class diagrams to depict the classes at the top level of the current model; such class diagrams are themselves contained by the top level of the current model.

Class:

A class is a description of a group of objects with common properties (attributes), common behavior (operations), common relationships to other objects, and common semantics. Thus, a class is a template to create objects. Each object is an instance of some class and objects cannot be instances of more than one class.

Classes should be named using the vocabulary of the domain. For example, the Bus class may be defined with the following characteristics: Attributes - location, time offered Operations - retrieve location, retrieve time of day, add a student to the offering. Each object would have a value for the attributes and access to the operations specified by the Airline database class.

UML Representation:

- In the UML, classes are represented as compartmentalized rectangles.
- The top compartment contains the name of the class.
- The middle compartment contains the structure of the class (attributes).
- The bottom compartment contains the behavior of the class as shown below.

Analysis Class Stereotypes:

Analysis class stereotypes represent three particular kinds of class that will be encountered again and again when carrying out requirements modeling. UML DEFINITION:

Stereotype:

- A new type of modeling element that extends the semantics of the met model.
- Stereotypes must be based on certain existing types or classes in the met model.
- Stereotypes may extend the semantics but not the structure of preexisting classes.
- Certain stereotypes are defined in the UML, others may be user defined.

UML is designed to be capable of extension; developers can add new stereotypes depend on need. But this is only done when it is absolutely necessary. Three analysis class stereotypes to the UML are:

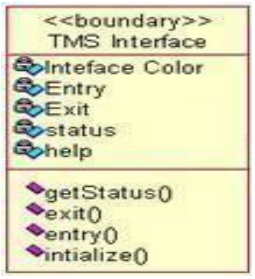
- Boundary classes,
- Control classes ➢ Entity classes.

1. Boundary classes:

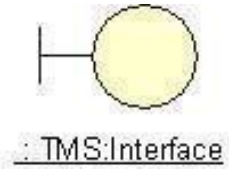
Boundary classes, it is a 'model interaction between the system and its actors'. Since they are part of the requirements model, boundary classes are relatively abstract. They do not directly represent all the different sorts of interface that will be used in the implementation language. The design model may well do this later, but from an analysis perspective we are interested only in identifying the main logical interfaces with users and other systems.

This may include interfaces with other software and also with physical devices such as printers, motors and sensors. Stereotyping these as boundary classes emphasizes that their main task is to manage the transfer of information across system boundaries. It also helps to partition the system, so that any changes to the interface or communication aspects of the system can be isolated from those parts of the system that provide the information storage.

The class TMS Interface is a typical boundary class. This style of writing the name shows that the class is TMS Interface and it belongs to the User Interface package when we write the package name in this way before the class name, it means that this class is imported from a different package from the one with which we are currently working. In this case, the current package is the Agate application package, which contains the application requirements model, and thus consists only of domain objects and classes. Alternative notations for Boundary class stereotype can be represented as shown below a) With stereotype



b) Symbol

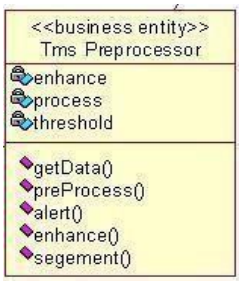


2. Entity classes

Entity classes represent something within the application domain, but external to the software system, about which the system must store some information. Instances of an entity class will often require persistent storage of information about the things that they represent. This can sometimes help to decide whether an entity class is the appropriate modeling construct.

For example, an actor is often not represented as an entity class. This is in spite of the fact that all actors are within the application domain, external to the software system and important to its operation. But most systems have no need to store information about their users or to model their behavior. While there are some obvious exceptions to this (consider a system that monitors user access for security purposes), these are typically separate, specialist applications in their own right. In such a context, an actor would be modeled appropriately as an entity class, since the essential requirements for such a system would include storing information about users, monitoring their access to computer systems and tracking their actions while logged on to a network. But it is more commonly the case that the software we develop does not need to know anything about the people that use it, and so actors are not normally modeled as classes. The following are representations for Entity classes.

a) With stereotype

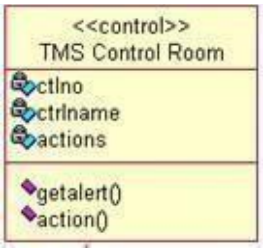


b) Symbol

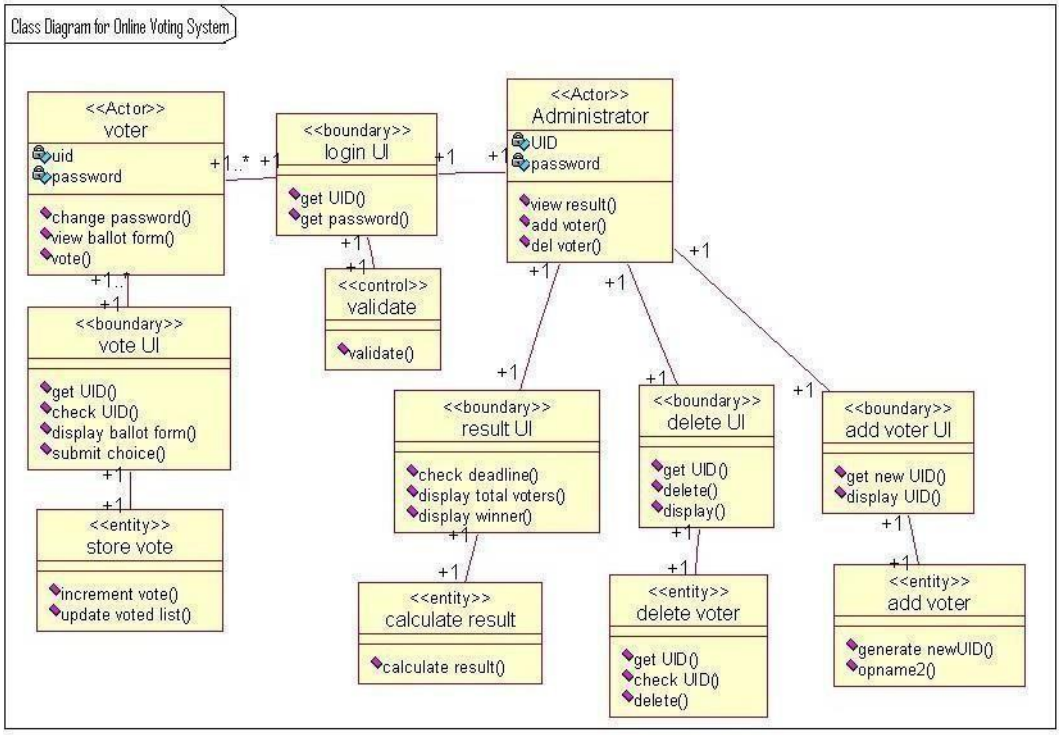
3. Control classes

The third of the analysis class stereotypes is the control class, given by y the class Searching system in Search flight. Control classes 'represent coordination, sequencing, transactions and control of other objects' .In the USDP, as in the earlier methodology Objector. It is generally recommended that there should be a control class for each use case.

In a sense, then, the control class represents the calculation and scheduling aspects of the logic of the use case at any rate, those parts that are not specific to the behavior of a particular entity class, and that *are* specific to the use case. Meanwhile the boundary class represents interaction with the user and the entity classes represent the behavior of things in the application domain and storage of information that is directly associated with those things. The following are the notations can be used to represent Control class a) With stereotype



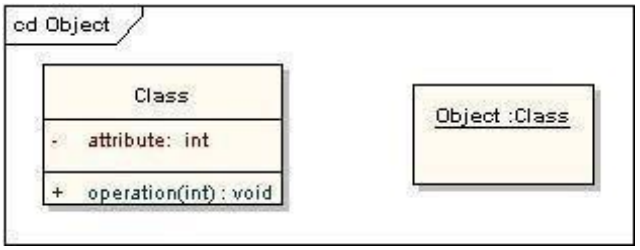
Class Diagram for ONLINE VOTING SYSTEM



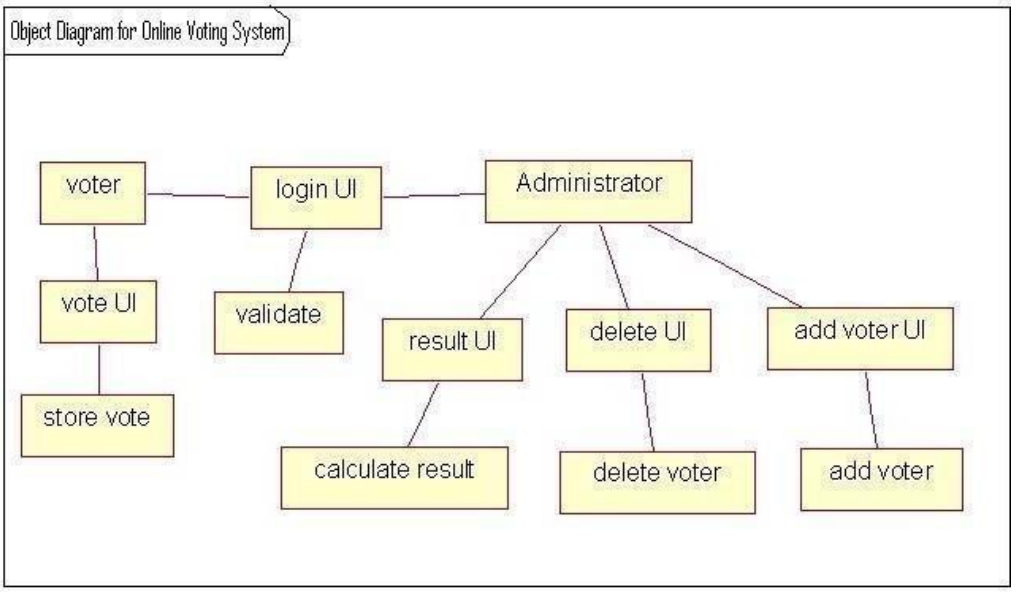
16. OBJECT DIAGRAMS

An object diagram may be considered a special case of a class diagram. Object diagrams use a subset of the elements of a class diagram in order to emphasize the relationship between instances of classes at some point in time. They are useful in understanding class diagrams. They don't show anything architecturally different to class diagrams, but reflect multiplicity and roles. Class and Object Elements

The following diagram shows the differences in appearance between a class element and an object element. Note that the class element consists of three parts, being divided into name, attribute and operation compartments; by default, object elements don't have compartments. The display of names is also different: object names are underlined and may show the name of the classifier from which the object is instantiated.



Object Diagram for Online Voting System



17. UML STATE CHART DIAGRAM

STATE MACHINE DIAGRAM:

Use case and scenarios provide a way to describe system behavior in the form of interaction between objects in the system. Some times it is necessary to consider inside behavior of an object.

A state chart diagram shows the state of a single object, the events or messages that causes a transition from one state to another and the actions that result from a state change. As in activity diagram ,state chart diagram also contains special symbols for start state and stop state. State chart diagram cannot be created for every class in the system, it is only for those class objects with significant behavior STATE.

UML notation for STATE is

To identify the states for an object its better to concentrate on sequence diagram.

In our application the object for Account may have in the following states, initialization, open and closed state. These states are obtained from the attribute and links defined for the object.

Each state also contains a compartment for actions.

ACTIONS:

Actions on states can occur at one of four times:

- on entry
- on exit
- do
- On event.

On entry: What type of action that object has to perform after entering into the state?

On exit: What type of action that object has to perform after exiting from the state?

Do: The task to be performed when object is in this state, and must to continue until it leaves the state.

On event: An on event action is similar to a state transition label with the following syntax:

event(args)[condition]: the Action

STATE TRANSITION:

A state transition indicates that an object in the source state will perform certain specified actions and enter the destination state when a specified event occurs or when certain conditions are satisfied. A state transition is a relationship between two states, two activities, or between an activity and a state.

We can show one or more state transitions from a state as long as each transition is unique. Transitions originating from a state cannot have the same event, unless there are conditions on the event.

Provide a label for each state transition with the name of at least one event that causes the state transition. You do not have to use unique labels for state transitions because the same event can cause a transition to many different states or activities.

Transitions are labeled with the following syntax:

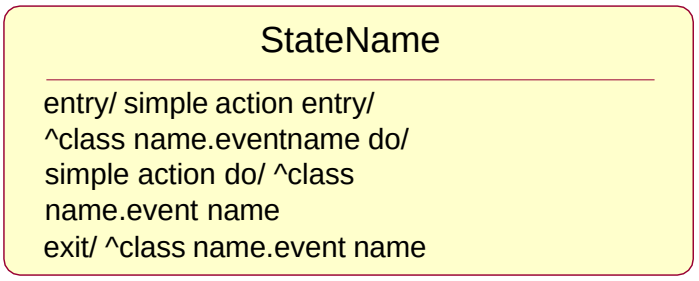
event (arguments) [condition] / action ^ target.sendEvent (arguments) Only one event is allowed per transition, and one action per event.

STATE DETAILS:

Actions that accompany all state transitions into a state may be placed as an entry action within the state. Like wise that accompany all state transitions out of a state may be placed as exit actions within the state. Behavior that occurs within the state is called an activity.

An activity starts when the state is entered and either completes or is interrupted by an outgoing state transition. The behavior may be a simple action or it may be an event sent to another object.

UML notation for State Details



Purpose of state chart diagrams:

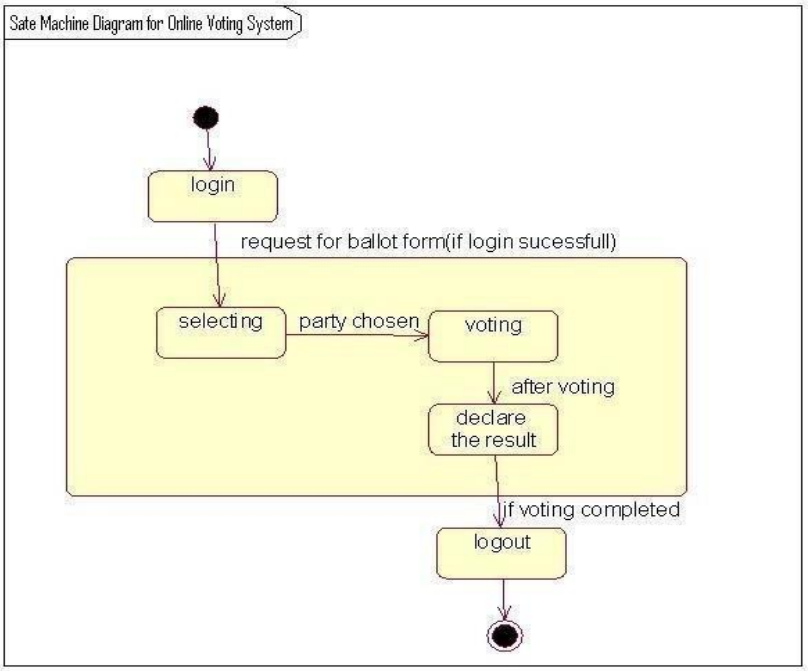
- We use state chart diagram when working on real-time process control applications or systems that involve concurrent processing. It will also be used when showing the behavior of a class over several use cases.
- State chart diagrams are used to model dynamic view of a system.
- State chart diagrams are used to emphasizing the potential states of the objects and the transitions among those states.
- State chart diagrams are used to modeling the lifetime of an object.
- State chart diagrams are used to model the behavior of an interface. Although an interface may not have any direct instances, a class that realizes such an interface may. Those classes conform to behavior specified by the state machine of this interface.
- State chart diagrams are used to focus on the changing state of a system driven by events.
- This diagram is also for constructing executable systems through forward and reverse engineering.
- To model reactive objects, especially instances of a class, use cases, and the system as a whole.

ELEMENTS OF STATE CHART DIAGRAMS:

1. **State:** It is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event.
2. **Event:** It is the specification of significant occurrence that has a location in time and space.
3. **Transition:** It is a relation between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and conditions are satisfied.
4. **Action state:** An action state is shorthand for a state with an entry action and at least one outgoing transition involving the implicit event of completing the entry action.
5. **Sequential sub state:** A submachine state represents the invocation of a state machine defined elsewhere. The submachine state is depicted as a normal state with the appropriate "include" declaration within its internal transitions compartment. As an option, the submachine state may contain one or more sub states, which represent already created states.
6. **Concurrent sub state:** A concurrent state is divided into two or more sub states. It is a state that contains other state vertices. Naturally, any sub state of a concurrent state may also be a composite state of either type. Any state enclosed within a composite state is called a sub state of that concurrent state.
7. **Initial state:** A pseudo state to establish the start of the event into an actual state.
8. **Final state:** The final state symbol represents the completion of the activity.

9. **History state:** History state is a state machine describes the dynamic aspects of an object whose current behavior depends on its past.
10. **Vertical Synchronization:** This merge branch bar symbol is also known as a "Synchronization Bar". It merges concurrent transitions to a single target. It splits a single transition into parallel transitions.
11. **Horizontal Synchronization:** This merge branch bar symbol is also known as a "Synchronization Bar". It merges concurrent transitions to a single target. It splits a single transition into parallel transitions.
12. **Guard conditions:** Activity and state diagrams express a decision when conditions are used to indicate different possible transitions that depend on Boolean conditions of container object. UML calls those conditions as guard conditions.
13. **Forks and joins:** A fork construct is used to model a single flow of control that divides into two or more separate, but simultaneous flows

State Chart Diagram for Online Voting System



18. COMPONENT DIAGRAM

Two type's implementation diagrams in UML terminology are

1. Component diagrams
2. Deployment diagrams

In a large project there will be many files that make up the system. These files will have dependencies on one another. The nature of these dependencies will depend on the language or languages used for the development and may exist at compile-time, at link-time or at run-time. There are also dependencies between source code files and the executable files or byte code files that are derived from them by compilation. Component diagrams are one of the two types of implementation diagram in UML. Component diagrams show these dependencies b/n software components in the system. Stereotypes can be used to show dependencies that are specific to particular languages also.

A component diagram shows the allocation of classes and objects to components in the physical design of a system. A components diagram may represent all or part of the component architecture of a system along with dependency relationships.

The dependency relationship indicates that one entity in a components diagram uses the services or facilities of another.

- Dependencies in the component diagram represent compilation dependencies.
- The dependency relationship may also be used to show calling dependencies among components, using dependency arrows from components to interfaces on othercomponents.

Different authors use component diagrams in different ways. Here we have the following distinction b/n them

- Components in a component diagram should be the physical components of a system.
- During analysis and the early stages of design, package can be used to show the logical grouping of class diagrams or of models that use other kinds of diagrams into packages relating to sub-systems.
- During implementation, package diagrams can be used to show the grouping of physical components into sub-systems.

If component diagrams are used, it is better to keep separate sets of diagrams to show compiletime and run-time dependencies, however, this is likely to result in a large number of diagrams.

Component diagrams show the components as types. If you wish to show instances of components of components you can use a deployment diagram.

Notations used in Component Diagram

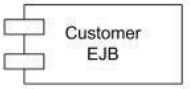
A component is shown as a classifier rectangle with the keyword **«component»**

Component WeatherService

Optionally, a component icon similar to the **UML 1.4** icons can be displayed in the right hand corner.

Component UserService

For backward compatibility reasons, the **UML 1.4** notations with protruding rectangles can still be used.



Component CustomerEJB in UML 1.x notation

A component may be manifested by one or more **artifacts**, and in turn, that artifact may be deployed to its execution environment. A **deployment specification** may define values that parameterize the component's execution.

Standard Component Stereotypes: There are several standard UML stereotypes that apply to components:

- «subsystem»
- «process»
- «service»
- «specification»
- «realization»
- «implement»

«subsystem»

Subsystem is a component representing unit of hierarchical decomposition for large systems, and is used to model large scale components. Definitions of subsystems may vary among different domains and software methods. It is expected that domain and method profiles will specialize this

element. A subsystem is usually indirectly instantiated. A subsystem may have specification and realization elements.

«process»:UML Standard Profile defines **process** as a transaction based component.

«service»:Service is a stateless, functional component.

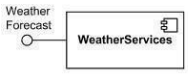
«specification»: Specification is a classifier that specifies a domain of objects without defining the physical implementation of those objects. For example, a component stereotyped by «specification» will only have provided and required interfaces, and is not intended to have any realizing classifiers as part of its definition. This differs from «type» because a «type» can have features such as attributes and methods that are useful to analysts modeling systems. «Specification» and «realization» are used to model components with distinct specification and realization definitions, where one specification may have multiple realizations.

«realization»:Realization is a classifier that specifies a domain of objects and that also defines the physical implementation of those objects. For example, a component stereotyped by «realization» will only have realizing classifiers that implement behavior specified by a separate «specification» component. This differs from «implementation class» because an «implementation class» is a realization of a class that can have features such as attributes and methods that are useful to system designers.

«implement»:Implement is a component definition that is not intended to have a specification itself. Rather, it is an implementation for a separate «specification» to which it has a dependency. It seems to duplicate «realization».

Provided Interface: A **provided interface** is the one that is either

- realized directly by the component itself, or
- realized by one of the classifiers realizing component, or Is provided by a public port of the component.

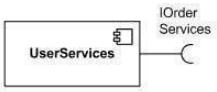


Weather Services component provides (implements) Weather Forecast interface

Required Interface

A **required interface** is either

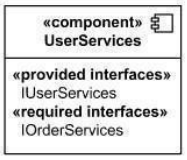
- designated by usage dependency from the component itself, or
- designated by usage dependency from one of the classifiers realizing component, or Is required by a public port of the component.



User Services component requires IOrderServices interface

External View of Component

A component has an **external view** or **"black-box"** view by means of interface symbols sticking out of the component box exposing its publicly visible properties and operations.
Optionally, a behavior such as a protocol state machine may be attached to an interface, port, and to the component itself, to define the external view more precisely by making dynamic constraints in the sequence of operation calls explicit. Other behaviors may also be associated with interfaces or connectors to define the "contract" between participants in collaboration (e.g., in terms of use case, activity, or interaction specifications).
Alternatively, the interfaces and/or individual operations and attributes can be listed in the compartments of a component box (for scalability, tools may offer a way of listing and abbreviating component properties and behavior).



External view of User Services component - it provides IUserServices interface and requires IOrderServices interface

For displaying the full signature of an interface of a component, the interfaces can also be displayed as typical classifier rectangles that can be expanded to show details of operations and events.

Connector

Connector in components extends connector from internal structures. It specifies a link that enables communication between two or more instances. Connector was extended in the components to include **contracts** and specific notation.

Connector linking components could be either:

- delegation connector, or
- Assembly connector.

Kind of connector attribute is derived: a connector with one or more ends connected to a port that is not on a part and that is not a behavior port is a delegation; otherwise it is an assembly.

Connector's contract is set of behaviors that specify the valid interaction patterns across the connector.

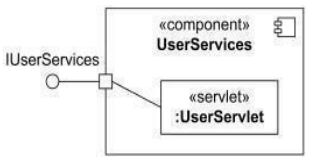
Delegation Connector

A delegation connector is a connector that links the external contract of a component (as specified by its ports) to the realization of that behavior. It represents the forwarding of events (operation requests and events): a signal that arrives at a port that has a delegation connector to one or more parts or ports on parts will be passed on to those targets for handling.

A delegation connector is a declaration that behavior that is available on a component instance is not actually realized by that component itself, but by one or more instances that have "compatible" capabilities. These situations are modeled through a delegation connector from a Port to compatible Ports or Parts.

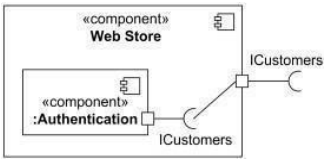
Delegation connectors can be used to model the hierarchical decomposition of behavior, where services provided by a component may ultimately be realized by one that is nested multiple levels deep within it. The word delegation suggests that concrete message and signal flow will occur between the connected ports, possibly over multiple levels. It should be noted that such signal flow is not always realized in all system environments or implementations (i.e., it may be design time only).

A port may delegate to a set of ports on subordinate components. In that case, these subordinate ports must collectively offer the delegated functionality of the delegating port. At execution time, signals will be delivered to the appropriate port. In cases where multiple target ports support the handling of the same signal, the signal will be delivered to all these subordinate ports. A delegation connector is notated as a connector from the delegating port to the handling port or part.



Delegation connector from the delegating port to the UserServlet part .If the delegation is handled by a simple port, then the connector may optionally be shown connected to the single lollipop or socket.

Delegation connector from the delegating port to the simple port of SearchEngine



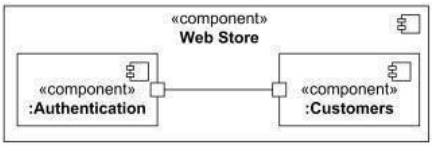
Delegation connector from the simple port of Authentication component to the delegating port

Assembly Connector

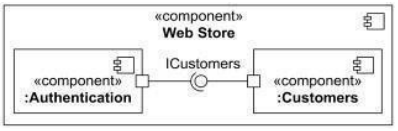
An assembly connector is a connector between two or more parts or ports on parts that defines that one or more parts provide the services that other parts use.

The execution time semantics for an assembly connector are that signals travel along an instance of a connector. Multiple connectors directed to and from different parts, or n-ary connectors where n > 2, indicates that the instance that will originate or handle the signal will be determined at execution time.

The interface compatibility between ports that are connected enables an existing component in a system to be replaced by one that (minimally) offers the same set of services. Also, in contexts where components are used to extend a system by offering existing services, but also adding new functionality, connectors can be used to link in the new component definition. Assembly connector is notated as a connector between two or more parts or ports on parts.

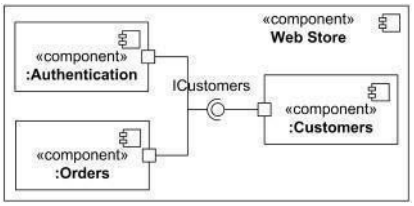


Assembly connector between ports of Authentication and Customers components When an assembly connector connects simple ports (ports that provide or require a single interface), it may be notated by a "ball-and-socket" connection between a provided interface and a required interface.



Assembly connector between simple ports of Authentication and Customers components Balland-socket notation may not be used to connect "complex" ports or parts without ports.

Where multiple components have simple ports that provide or require the same interface, a single symbol representing the interface can be shown, and lines from the components can be drawn to that symbol. This presentation option is applicable whether the interface is shown using "balland-socket" notation, or just using a required or provided interface symbol.

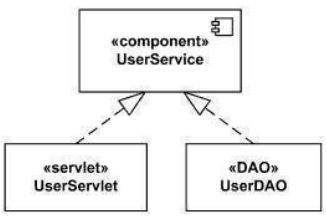


Component Realization

Component Realization is specialized realization dependency used to (optionally) define classifiers that realize the contract offered by a component in terms of its provided interfaces andrequired interfaces.

A component’s behavior may typically be realized (or implemented) by a number of Classifiers. In effect, it forms an abstraction for a collection of model elements. In that case, a component owns a set of Component Realization Dependencies to these Classifiers. In effect, it forms an abstraction for a collection of model elements. In that case, a component owns a set of Realization Dependencies to these Classifiers.

A component realization is notated in the same way as the realization dependency, i.e., as a general dashed line from implementing classifiers to realized component with hollow triangle as an arrowhead.

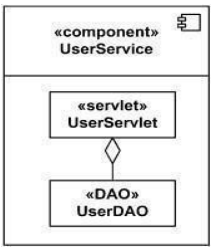


Component UserService realized by UserServlet and UserDAO.

For the purpose of applications that require multiple different sets of realizations for a single component specification, a set of standard stereotypes are defined in the UML Standard Profile. In particular, «specification» and «realization» are defined there for this purpose.

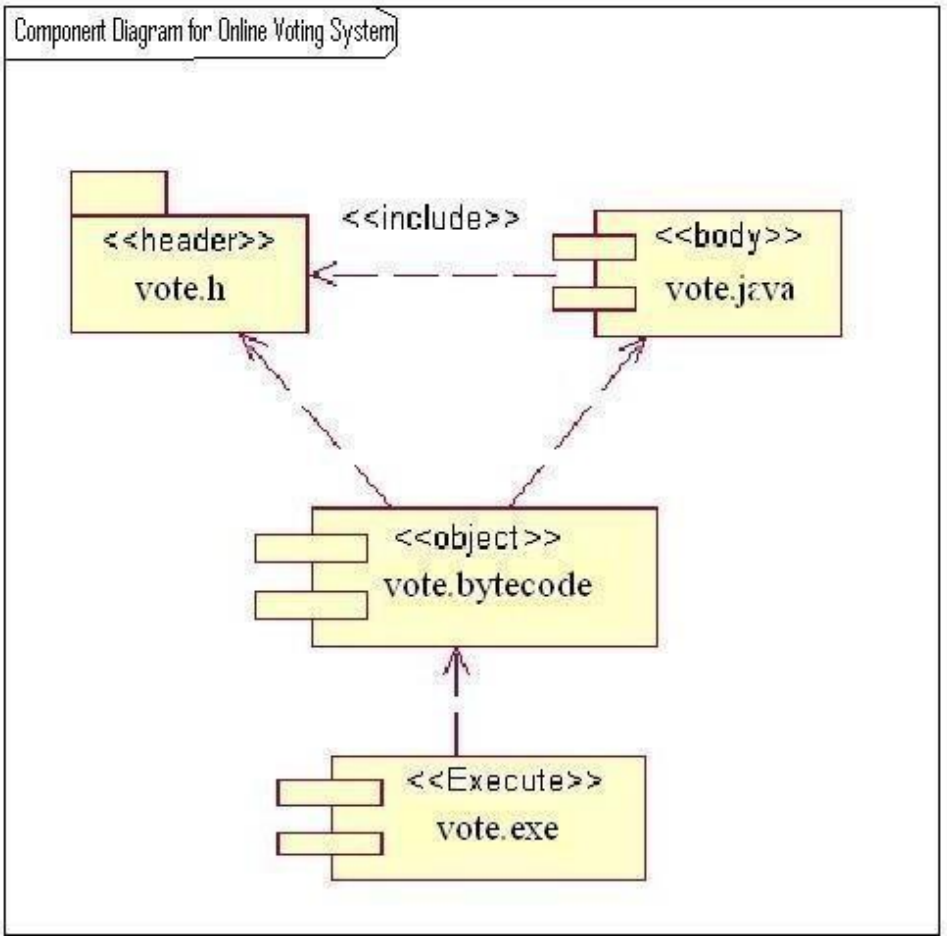
Component could be shown using internal view or "white-box" view exposing its private properties and realizing classifiers. This view shows how the external behavior is realizedinternally. The realizing classifiers could be listed in an additional «realizations» compartment. Compartments may also be used to display a listing of any parts and connectors, or any implementing artifacts.

White box view of User Services component - it is realized by UserServlet and UserDAO and manifested by UserService.jar artifact alternatively, the internal classifiers that realize the behavior of a component may be displayed nested within the component shape.



White box view of User Services component - it is realized by UserServlet and UserDAO

Component diagram for ONLINE VOTING SYSTEM



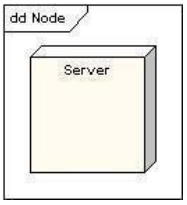
19. Deployment Diagram

The second type of implementation diagram provided by UML is the deployment diagram . they are used to show the configuration of runtime processing elements and the software components and processes that are located on them .They are made up of nodes and communication associations**.**

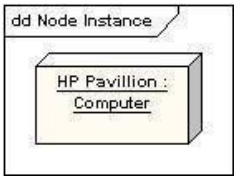
Notations used in Deployment Diagram

Node

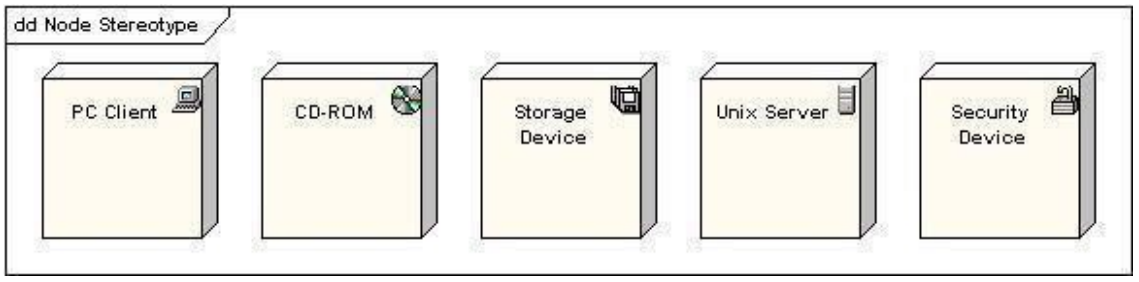
A Node is either a hardware or software element. It is shown as a three-dimensional box shape, as shown below.



Node Instance: A node instance can be shown on a diagram. An instance can be distinguished from a node by the fact that its name is underlined and has a colon before its base node type. An instance may or may not have a name before the colon. The following diagram shows a named instance of a computer.

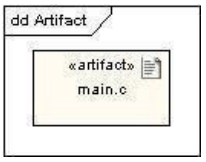


Node Stereotypes: A number of standard stereotypes are provided for nodes, namely «cdrom», «cd-rom», «computer», «disk array», «pc», «pc client», «pc server», «secure», «server», «storage», «Unix server», «user pc». These will display an appropriate icon in the top right corner of the node symbol

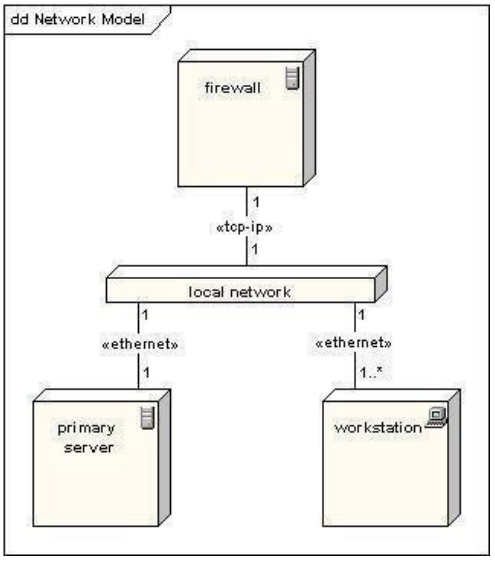


Artifact

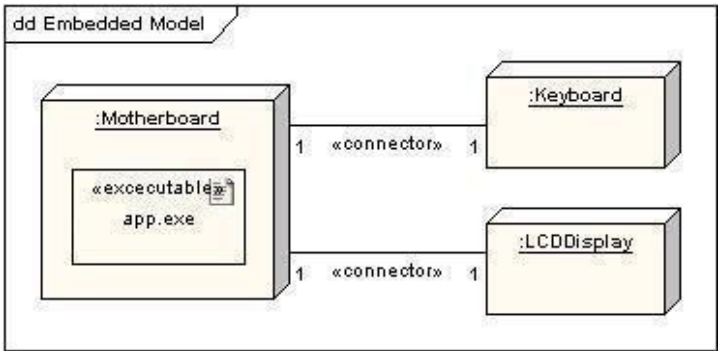
An artifact is a product of the software development process. That may include process models (e.g. use case models, design models etc), source files, executables, design documents, test reports, prototypes, user manuals, etc. An artifact is denoted by a rectangle showing the artifact name, the «artifact» keyword and a document icon, as shown below.



Association: In the context of a deployment diagram, an association represents a communication path between nodes. The following diagram shows a deployment diagram for a network, depicting network protocols as stereotypes, and multiplicities at the association ends.



Node as Container: A node can contain other elements, such as components or artifacts. The following diagram shows a deployment diagram for part of an embedded system, depicting an executable artifact as being contained by the motherboard node.



Deployment Diagram for ONLINE VOTING SYSTEM