SASTRA

ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION

DEEMED TO BE UNIVERSITY

(U/S 3 of the UGC Act, 1956)

THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

Software Design with UML

Course Code: CSE317

Semester: V

Lab Manual

2024

SHANMUGHA ARTS, SCIENCE, TECHNOLOGY AND RESEARCH ACADEMY
(SASTRA Deemed to be) University
Tirumalaisamudram, Thanjavur-613 401
School of Computing

Course Objective:

This course will help the learner to explore and model the software using UML that transforms the analysis data to design components which will be used for constructing the software.

Course Learning Outcomes:

- Prepare the software requirement specification by thorough analysis of requirements
- Model the interaction between the use-cases as sequence and collaboration diagrams
- Design state structure diagrams using classes and packages
- Develop a dynamic model comprises of activity diagrams and state transition diagrams
- Design component and deployment models that represents the system components' interaction and execution architecture respectively

List of Experiments:

1**.** Choose a problem statement, analyze the problem and identify the requirements. Generate SRS document with the requirements. Design the software model and present it as the following models: Case Modeling

2. Draw Use-case diagrams that depicts the relationship between the actors and the system Interaction diagram

3. Depict the interactions between the objects using sequence diagram.

4. Emphasize the structural aspects of the objects (i.e., how the objects of the system are connected to each other) using Collaboration diagram Static structure diagram

5. Identify the analysis classes and depict the relationship between these classes (Class diagram). The class diagram should project the relationships like Association, Generalization, Aggregation, Dependency, Interfacing, and Multiplicity

6. Simplify the complex class diagrams, group the classes as packages and draw package diagram.

7. Draw state diagram that represent the system as number of states and interactions between states.

8. Represent the workflow of the systems using activities and actions using activity diagram.

9. Draw component diagram to represent both the physical and logical aspects of the system (connections and dependencies)

10. Construct deployment diagram that shows the execution architecture of a system, including nodes such as hardware or software execution environments, and the middleware connecting them**.**

Exercise No. 1. Generate SRS document with the requirements

Objective:

- to define the project's functional and non-functional requirements clearly
- to serve as a reference for all stakeholders throughout the development process
- to ensure a shared understanding and alignment on project goals and deliverables.

Tools Required: Rational Requisite Pro

Theory or Concept:

An SRS is basically an organization's understanding (in writing) of a customer or potential client's system requirements and dependencies at a particular point in time (usually) prior to any actual design or development work. It's a two-way insurance policy that assures that both the client and the organization understand the other's requirements from that perspective at a given point in time.

The SRS document itself states in precise and explicit language those functions and capabilities a software system (i.e., a software application, an eCommerce Web site, and so on) must provide, as well as states any required constraints by which the system must abide. The SRS also functions as a blueprint for completing a project with as little cost growth as possible. The SRS is often referred to as the "parent" document because all subsequent project management documents, such as design specifications, statements of work, software architecture specifications, testing and validation plans, and documentation plans, are related to it.

It's important to note that an SRS contains functional and nonfunctional requirements only; it doesn't offer design suggestions, possible solutions to technology or business issues, or any other information other than what the development team understands the customer's system requirements to be. A well-designed, well-written SRS accomplishes four major goals: It provides feedback to the customer. An SRS is the customer's assurance that the development organization understands the issues or problems to be solved and the software behavior necessary to address those problems. Therefore, the SRS should be written in natural language (versus a formal language, explained later in this article), in an unambiguous manner that may also include charts, tables, data flow diagrams, decision tables, and so on. It decomposes the problem into component parts. The simple act of writing down software requirements in a well-designed format organizes information, places borders around the problem, solidifies ideas, and helps break down the problem into its component parts in an orderly fashion. It serves as an input to the design specification. As mentioned previously, the SRS serves as the parent document to subsequent documents, such as the software design specification and statement of work. Therefore, the SRS must contain sufficient detail in the functional system requirements so that a design solution can be devised. It serves as a product validation check. The SRS also serves as the parent document for testing and validation strategies that will be applied to the requirements for verification.

SRSs are typically developed during the first stages of "Requirements Development," which is the initial product development phase in which information is gathered about what requirements are needed--and not. This information-gathering stage can include onsite visits, questionnaires, surveys, interviews, and perhaps a return-on-investment (ROI) analysis or needs analysis of the customer or client's current business environment. The actual specification, then, is written after the requirements have been gathered and analyzed. SRS should address the following The basic issues that the SRS shall address are the following:

a) Functionality. What is the software supposed to do?

b) External interfaces. How does the software interact with people, the system‘s hardware, other hardware, and other software?

c) Performance. What is the speed, availability, response time, recovery time of various software functions, etc.?

d) Attributes. What are the portability, correctness, maintainability, security, etc. considerations?
e) Design constraints imposed on an implementation. Are there any required standards in effect, implementation language, policies for database integrity, resource limits, operating environment(s) etc.?
Characteristics of a good SRS
An SRS should be
a) Correct
b) Unambiguous
c) Complete
d) Consistent
e) Ranked for importance and/or stability
f) Verifiable
g) Modifiable
h) Traceable
Correct - This is like motherhood and apple pie. Of course you want the specification to be correct. No one writes a specification that they know is incorrect. We like to say - "Correct and Ever Correcting." The discipline is keeping the specification up to date when you find things that are not correct.
Unambiguous - An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation. Again, easier said than done. Spending time on this area prior to releasing the SRS can be a waste of time. But as you find ambiguities - fix them. Complete - A simple judge of this is that is should be all that is needed by the software designers to create the software.
Consistent - The SRS should be consistent within itself and consistent to its reference documents. If you call an input "Start and Stop" in one place, don't call it "Start/Stop" in another.
Ranked for Importance - Very often a new system has requirements that are really marketing wish lists. Some may not be achievable. It is useful to provide this information in the SRS.
Verifiable - Don't put in requirements like - "It should provide the user a fast response." Another One of my favorites is - "The system should never crash." Instead, provide a quantitative requirement like: "Every keystroke should provide a user response within 100 milliseconds."
Modifiable - Having the same requirement in more than one place may not be wrong - but tends to make the document not maintainable.
Traceable - Often, this is not important in a non-politicized environment. However, in most organizations, it is sometimes useful to connect the requirements in the SRS to a higher level document. Why do we need this requirement?

Procedure:

1. Open a text editor
2. Write the introduction, purpose, intended audience and reading suggestions, scope, overall description, product perspective and references of the project
3. Identify the functional requirements
4. Identify the non-functional requirements
5. Specify the hardware requirements
6. Specify the software requirements
7. Review the document
8. Manage the document

Output:

1.INTRODUCTION
Movie ticket booking system that was moderately complicated in the implementation, required recommended slandered supportive provisions to cover all the minor and major requirements .The functional and non- functional requirements for the ticket booking stimulation system will be discussed in the following recommended framed template.
1.1.PURPOSE

This document describes the software requirements for a movie ticket booking system. It is intended for a developer designer stakeholder and maintainer of the movie ticket booking system.

1.2.DOCUMENT CONVENTIONS

This SRS is written in Arial font with size 11 in italics, and Times font with size 14 for bullet points. The priorities for higher level requirements are inherited in detailed requirements.

1.3 INTENDED AUDIENCE AND READING SUGGESTIONS

This System Requirement Specification is meant for the Project makers, Project overseers, The Website Administrators, Customers, and the Suppliers. The SRS is written in the standard IEEE [2] format. The readers should read the whole SRS sequentially to gain a proper understanding of the project.

1.4 PRODUCT SCOPE

The functional and non- functional specification of the booking system is to support booking in theaters for audiences. Software to communicate between theater owners and audience is not a part of the requirement. Sometimes payments may fail during certain reasons.

2.OVERALL DESCRIPTION

This section does not state specific requirements instead, it provides a background of requirements. This section contains subsections as follows.

2.1.PRODUCT PERSPECTIVE

The software described in this SRS is the software for a complete movie ticket booking system. The system merges various hardware and software elements and further interfaces with external systems. It relies on a number of external interfaces for persistence and un - handled tasks, as well as physically interfacing with humans.

2.2 PRODUCT FUNCTIONS

The movie ticket booking system will provide a number of functions,each is listed below. • Maintain data associated with theaters,movie names ,vacancies,cost of ticket . Maintain records for many customers. • A customer can be either a member or non-member. • A customer has a username (unique across all users), password (no restrictions), email address (no restrictions), and postal address (unverified).

• Anyone may sign up for a customer account. Allow any customer to become a member. Allow customers and managers to log in and out of the system. Payment option should be there to pay money online. Show if any discounts are available. After completion of payment check the history and email to view the ticket If any help and support is needed go to the help and support section.

2.3.USER CLASSES AND CHARACTERISTICS

There are three separate user interfaces used by the RFOS software, each related to an interfaced physical hardware device. These three user interfaces are the surface computer user interface, tablet UI, display UI.

2.4 OPERATING ENVIRONMENT

The system will work on any hardware device having a internet connectivity and browsers to browse the website. The browsers should be up to date to run various features of the website. The system will not run on an OS below Windows XP. The system will also not run on browsers not supporting JAVA plugins.

2.5 DESIGN AND IMPLEMENTATION CONSTRAINTS

Security is not a concern for this system. The database may store passwords in plain text and there doesn't need to be a password recovery feature nor lockout after numerous invalid login attempts. As such, the system may not work correctly in cases when security is a concern. These cases include those listed above in addition to lack of an encrypted connection when sending credit card information and forcing users to use “strong” passwords. A strong password is a password that meets a number of conditions that are set in place so that user's passwords cannot be easily guessed by an attacker. Generally, these rules include ensuring that the password contains a sufficient number of characters and contains not only lowercase letters but also capitals, numbers, and in some cases, symbols.

2.6 ASSUMPTIONS AND DEPENDENCIES

Since the Online Movie ticket booking system is only accessible through the internet, it is assumed that the end user has a connection to the internet. It is also assumed that the user has a web browser able to display the website.
Client:
We have assumed that all of the computer systems are in proper working condition and that the user is capable of operating these system's basic functions including but not limited to being able to power on the system, login and open browsers, and navigate the browser to the address of the website.
Provider:
We have assumed that the Online movie ticket booking system will be running on a properly working web server and database system with an Internet connection that allows this system to perform all communications with clients.
Assumptions:
• The manager account's username and password may be hard coded.
• The manager cannot be a customer.
• Any user cannot edit their account information.
2.7 APPORTIONING OF REQUIREMENTS
As stated by the customer, security is not a concern of this project. As such, it is beyond the scope of this system to encrypt personal user data, encrypt credit card information, prevent unauthorized login attempts, or any other concern of this nature. Additionally, the system is not responsible for the following Ensure that the credit card or debit card information is valid Verifying email address provided by the user Allowing users to edit their account details (username, password, mailing address, etc).
Allowing customers to book more than one ticket Providing individual theatre in each page along with images and reviews Allowing the manager to update login credentials or other information about the Manager.
3.EXTERNAL INTERFACE REQUIREMENTS
3.1 SYSTEM INTERFACES
The system will interface with the following two systems:
1. A credit/debit card processing system: The system will access the credit/debit card processing system via its web services API.
2. The theater Inventory database: The system will interact with the inventory database via an ODBC connection.
3.2 .USER INTERFACES
The interface of the ticket booking system must represent the ergonomic requirements.
3.3 .HARDWARE INTERFACES
Not applicable
3.4 .SOFTWARE INTERFACES
Not applicable
3.5.COMMUNICATION INTERFACES
Not applicable
4.OTHER NON FUNCTIONAL REQUIREMENTS
4.1 PERFORMANCE REQUIREMENTS
The performance requirements are as follows:-
System login/logout shall take less than 5 seconds.
The system should run properly on all browsers.
The system should be light and run smoothly.
4.2 SAFETY REQUIREMENTS
The safety requirements are as follows:-
The password should not be visible while typing.
The system should use a secure connection to connect with the credit/debit card processing system.
4.3 SECURITY REQUIREMENTS
The security requirements are as follows:-
The user (require a username and password to login into the system.

The customers cannot book without logging in.
The booking will not be confirmed until the payment process is completed.

4.4 SOFTWARE QUALITY ATTRIBUTES

Reliability

The average time of failure for the system is 30 days. In the event that the server crashes, The system will take a week to be running again.

Availability

The Online Booking will be available 24x7, with the exception of being down for maintenance no more than 3 hours a week. If the system crashes, it should be back up within a week.

Security

Users will be able to access only their own personal information and not that of other users. Purchases will be handled through a secure server to ensure the protection of user's credit card and personal information.

Maintainability

Any updates or defect fixes shall be able to be made on server-side computers only, without any patches required by the user.

4.5 BUSINESS RULES

The business rules are as follows:
The administrator cannot be a customer.
The administrator cannot access the customers' personal information.
The administrator will receive notifications from the credit/debit card processing system to process the booking and confirm payment of booking is received.

Exercise No. 2 Draw Use-case diagrams that depicts the relationship between the actors and the system Interaction diagram

Objective

- To draw the Use Case Diagram using Rational Rose.

Tools Required:
Rational Rose, Windows XP

Theory or Concept:
According to the UML specification a use case diagram is —a diagram that shows the relationships among actors and use cases within a system.‖ Use case diagrams are often used to:
Provide an overview of all or part of the usage requirements for a system or organization in the form of an essential model or a business model Communicate the scope of a development project
Model your analysis of your usage requirements in the form of a system use case model Use case models should be developed from the point of view of your project stakeholders and not from the (often technical) point of view of developers. There are guidelines for:
Use Cases
Actors
Relationships
System Boundary Boxes

1. Use Cases

A use case describes a sequence of actions that provide a measurable value to an actor. A use
the case is drawn as a horizontal ellipse on a UML use case diagram.
1. Use Case Names Begin With a Strong Verb
2. Name Use Cases Using Domain Terminology
3. Place Your Primary Use Cases In The Top-Left Corner Of The Diagram
4. Imply Timing Considerations By Stacking Use Cases.

2. Actors

An actor is a person, organization, or external system that plays a role in one or more interactions
with your system (actors are typically drawn as stick figures on UML Use Case diagrams).
1. Place Your Primary Actor(S) In The Top-Left Corner Of The Diagram
2. Draw Actors To The Outside Of A Use Case Diagram
3. Name Actors With Singular, Business-Relevant Nouns
4. Associate Each Actor with One Or More Use Cases
5. Actors Model Roles, Not Positions
6. Use <<system>> to Indicate System Actors
7. Actors Don ‘t Interact With One Another
8. Introduce an Actor Called —Time‖ to Initiate Scheduled Events

3. Relationships

There are several types of relationships that may appear on a use case diagram:
1. An association between an actor and a use case
2. An association between two use cases
3. A generalization between two actors
4. A generalization between two use cases

Associations are depicted as lines connecting two modeling elements with an optional open-headed arrowhead on one end of the line indicating the direction of the initial invocation of the relationship. Generalizations are depicted as a close-headed arrow with the arrow pointing towards the more general modeling element.
1. Indicate An Association Between An Actor And A Use Case If The Actor Appears Within The Use Case Logic

2. Avoid Arrowheads On Actor-Use Case Relationships
3. Apply <<include>> When You Know Exactly When To Invoke The Use Case
4. Apply <<extend>> When A Use Case May Be Invoked Across Several Use Case Steps
5. Introduce <<extend>> associations sparingly
6. Generalize Use Cases When a Single Condition Results In Significantly New Business Logic
7. Do Not Apply <<uses>>, <<includes>>, or <<extends>>
8. Avoid More Than Two Levels Of Use Case Associations
9. Place An Included Use Case To The Right Of The Invoking Use Case
10. Place An Extending Use Case Below The Parent Use Case
11. Apply the —Is Likell Rule to Use Case Generalization
12. Place an Inheriting Use Case Below The Base Use Case
13. Apply the —Is Likell Rule to Actor Inheritance
14. Place an Inheriting Actor Below the Parent Actor

4. System Boundary Boxes

The rectangle around the use cases is called the system boundary box and as the name suggests it indicates the scope of your system – the use cases inside the rectangle represent the functionality that you intend to implement.

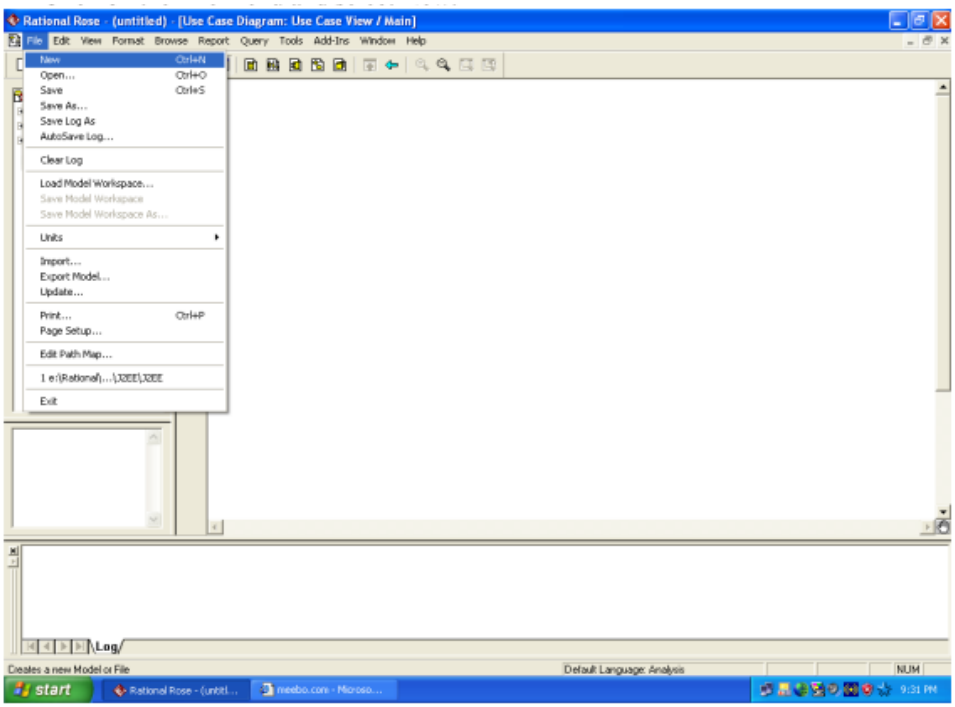
1. Indicate Release Scope with a System Boundary Box.
2. Avoid Meaningless System Boundary Boxes.

Creating Use Case Diagrams

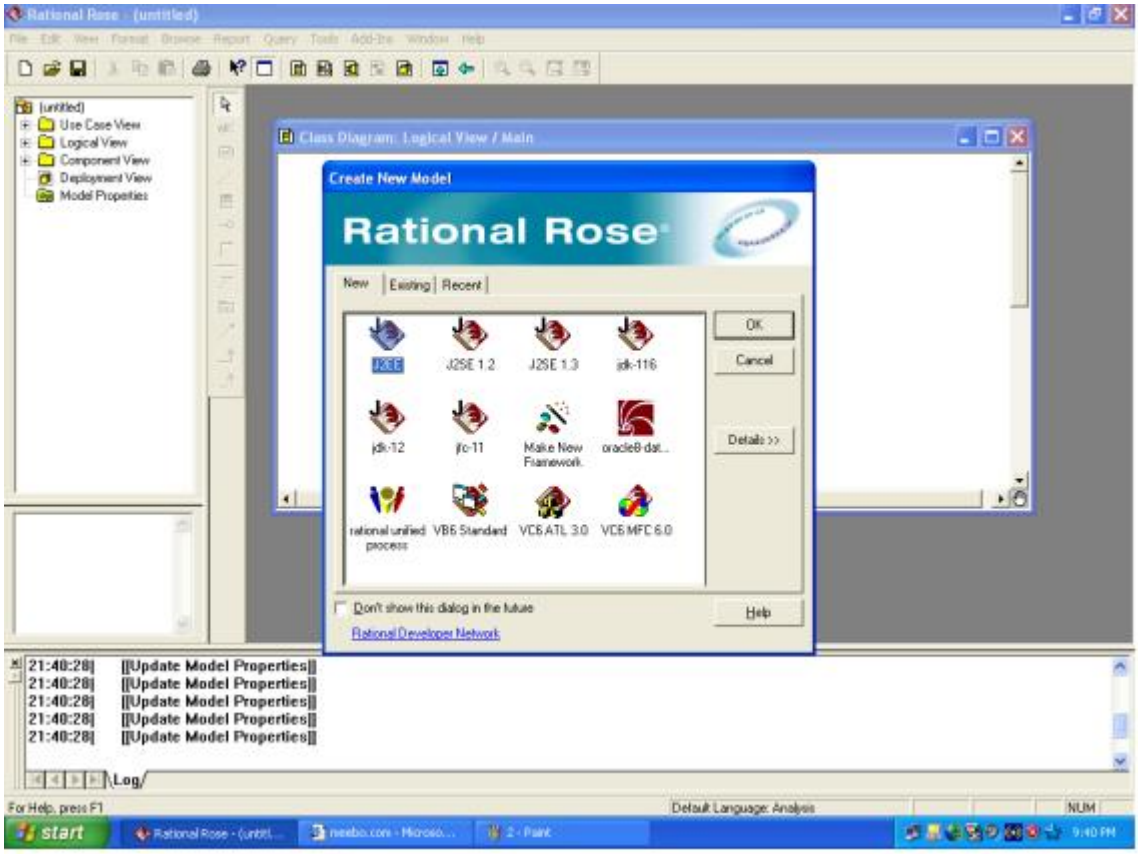
We start by identifying as many actors as possible. You should ask how the actors interact with the system to identify an initial set of use cases. Then, on the diagram, you connect the actors with the use cases with which they are involved. If actor supplies information, initiates the use case, or receives any information as a result of the use case, then there should be an association between them.

Procedure:

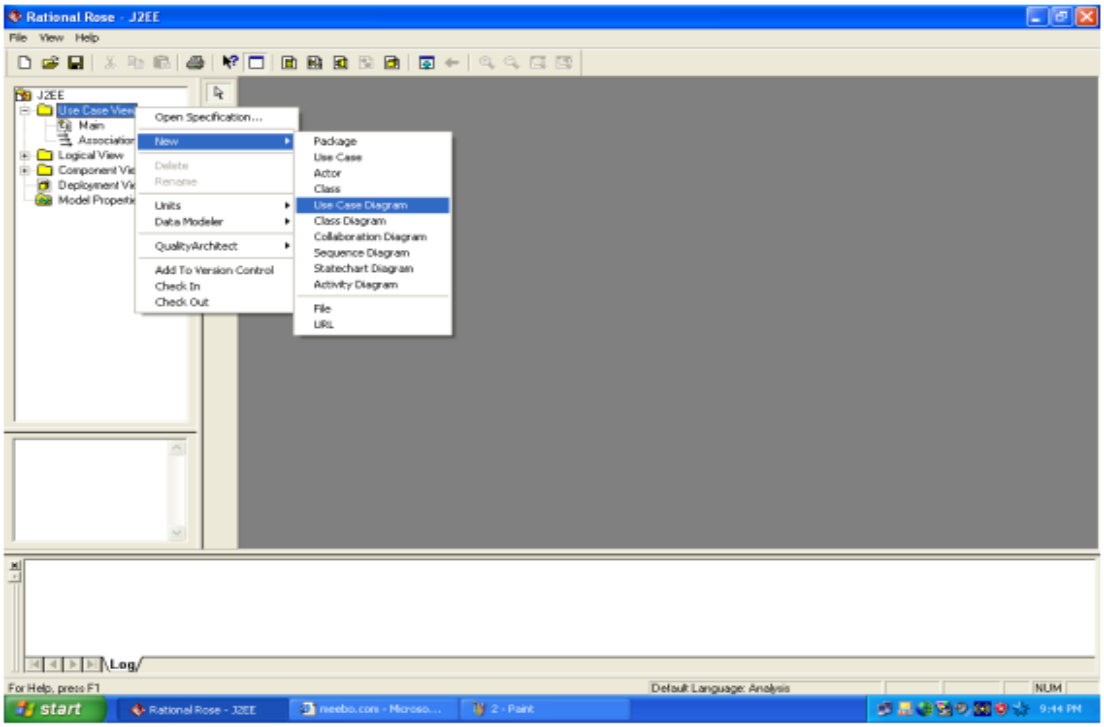
Step 1: Click on the File menu and select New.



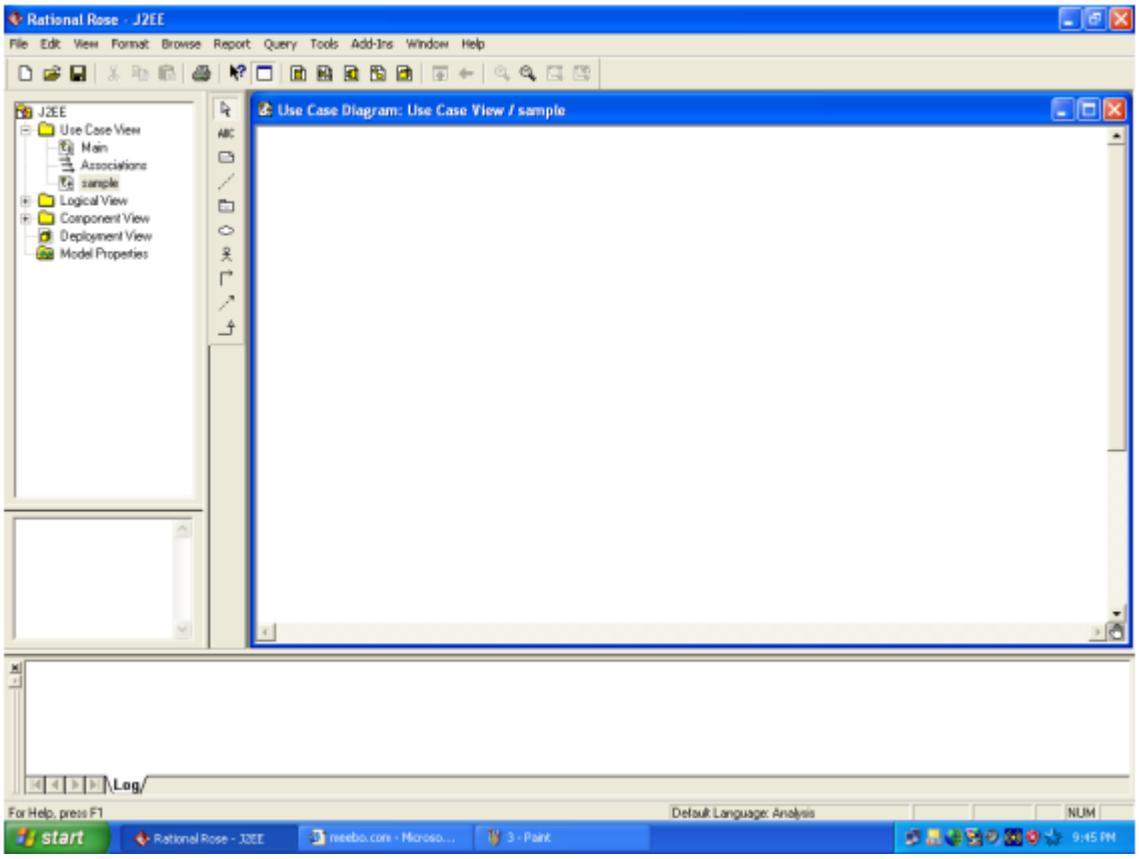
Step2: Now from the Dialogue Box that appears ,select the language which you want to use for creating your model.



Step 3: In the left hand side window of Rational Rose right click on —Use Case viewll and select New>Use Case Diagram.

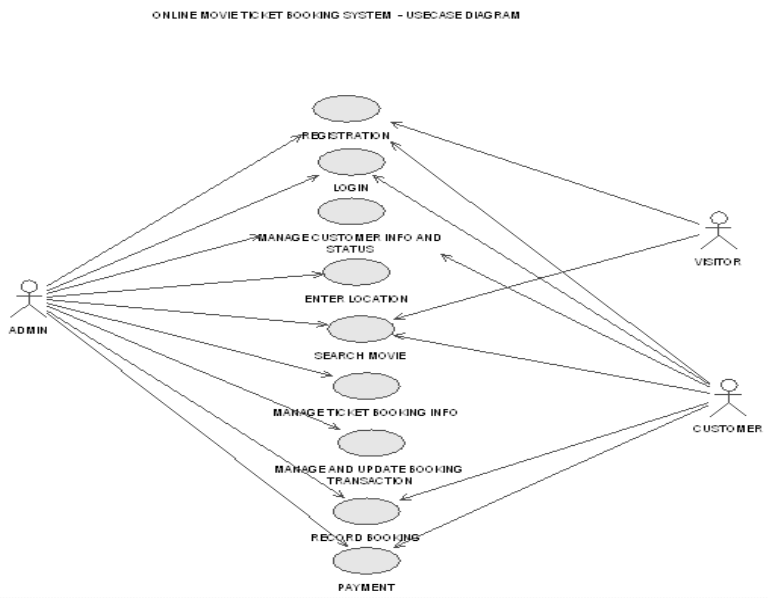


Step 4: Enter the name of new Use Case file in the space provided, and then click on that file name.



You can now use the window that appears on right hand side to draw your Use Case diagram using the buttons provided on the vertical toolbar.

Output:



Exercise No. 3 Depict the interactions between the objects using sequence diagram.

Objective:

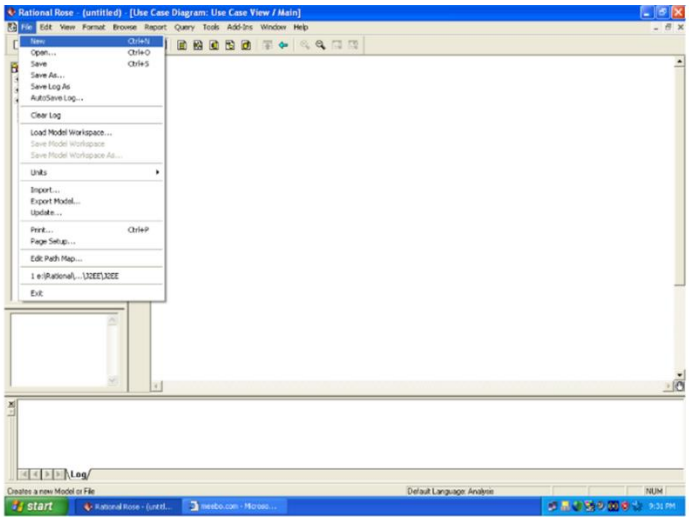
- To draw the Sequence Diagram using Rational Rose.

Tools Required:
Rational Rose, Windows XP

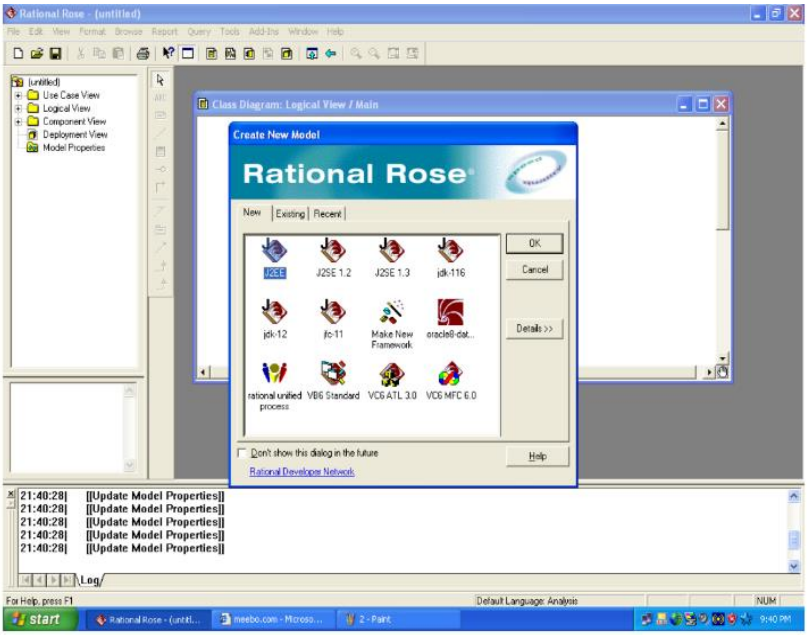
Theory or Concept:

1. **Objects (Participants)**:
   - **Lifeline**: Each object participating in the sequence diagram is represented by a vertical dashed line known as a lifeline. The lifeline indicates the existence and time span of the object during the execution of the scenario.
2. **Messages**:
   - **Message Types**:
     - **Synchronous Message**: A message where the sender waits for a response from the receiver before continuing.
     - **Asynchronous Message**: A message where the sender continues without waiting for a response from the receiver.
     - **Return Message**: A message used to return control and possibly a value from a method invocation.
   - **Message Arrows**: These arrows indicate the direction of the message flow between objects. They are labeled with the message name and, optionally, parameters and return values.
3. **Activation (Execution) Bar**:
   - This bar, typically shown as a thin rectangle on the lifeline, represents the period during which an object is performing an operation or method call. It shows the duration of the method execution.
4. **Focus of Control**:
   - **Self-Message**: A message that an object sends to itself, representing internal processing or method invocation.
   - **Creation Message**: Indicates the creation of a new object.
5. **Parallel Execution**:
   - **Parallel Combined Fragments**: Represented by a box with a dashed line, indicating concurrent execution paths where messages may be sent and received concurrently.

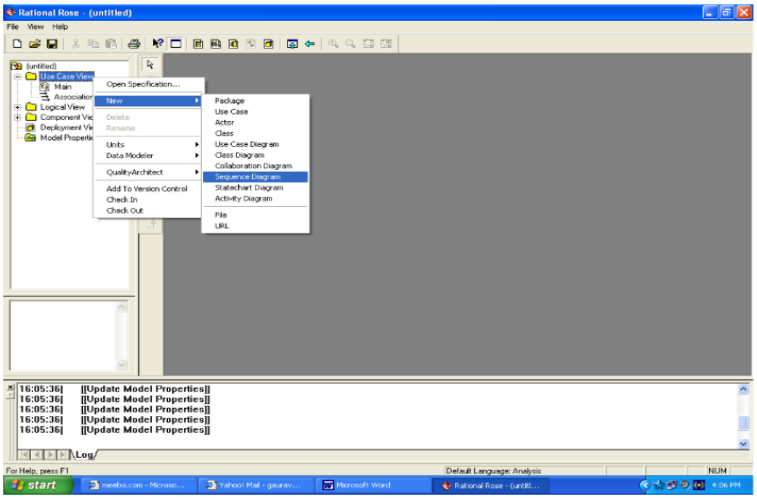
Procedure:
Step 1: Click on the File menu in Rational Rose Tool



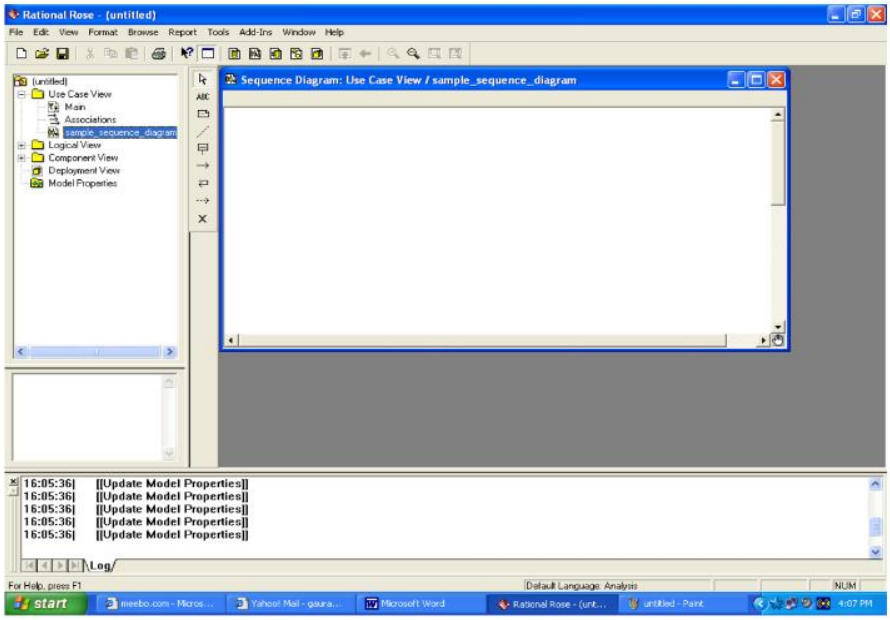
Step 2: Now from the Dialogue Box that appears, select the language which you want to use for creating your model



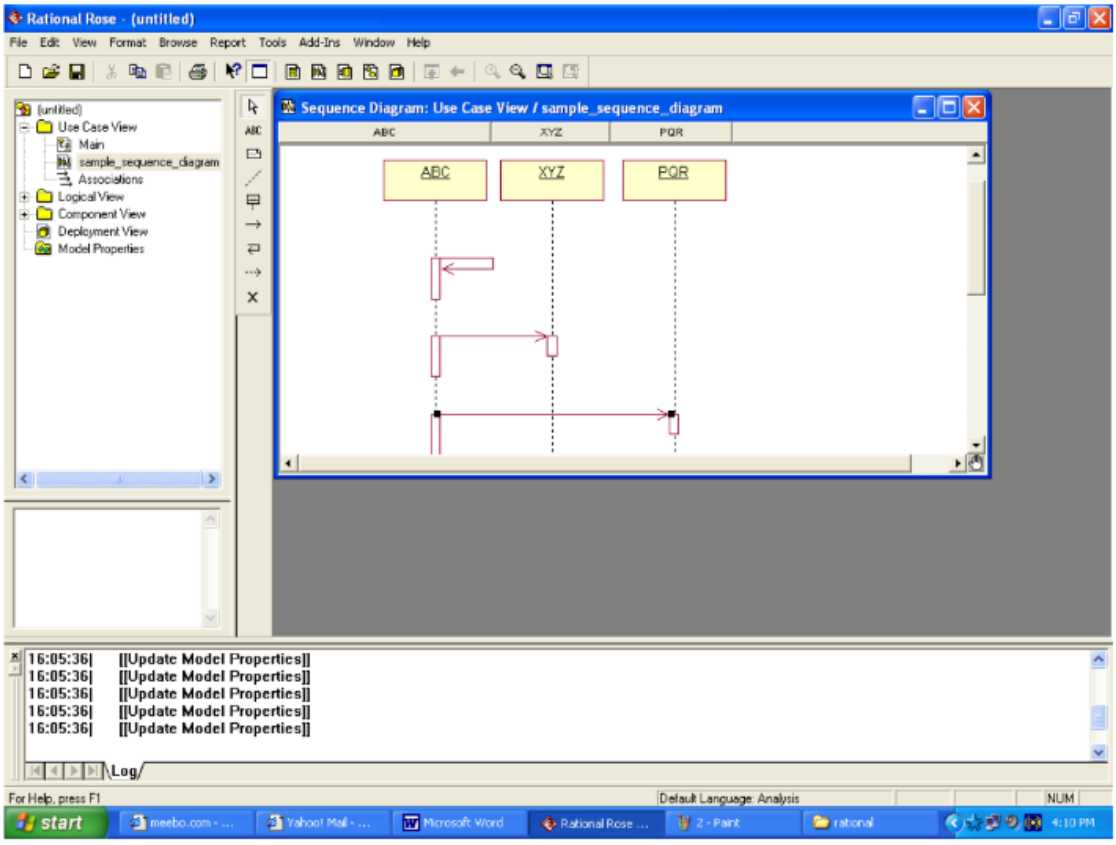
Step 3: In the left-hand side window of Rational Rose right-click on ‘use case view’ and select New > Sequence Diagram



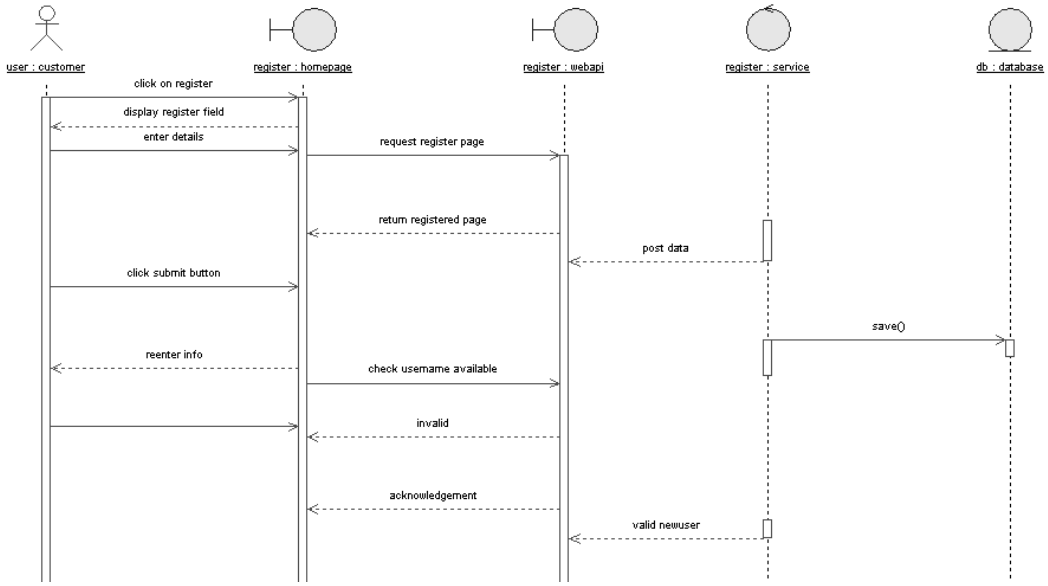
Step 4: Enter the name of new Sequence file in the space provided, and then click on that file name



Step 5: You can now use the window that appears on the right-hand side to draw sequence diagram using the buttons provided on the vertical toolbar



Output

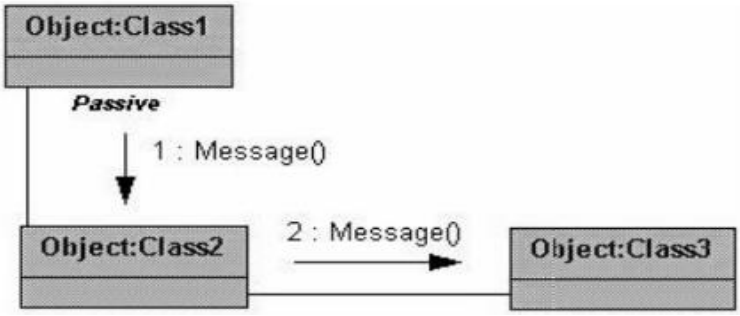

user : customer
register : homepage
register : webapi
register : service
db : database
click on register
display register field
enter details
request register page
return registered page
post data
click submit button
save()
reenter info
check username available
invalid
acknowledgement
valid newuser

Exercise No. 4 Emphasize the structural aspects of the objects using Collaboration diagram Static structure diagram

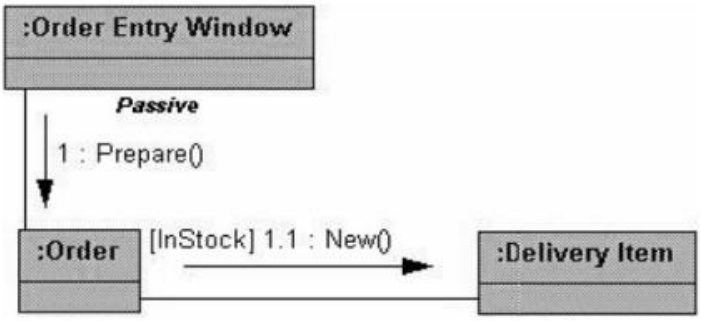
Objective:
to illustrate how objects interact to accomplish specific tasks, highlighting the flow of messages and object responsibilities. It provides a clear view of the system's dynamic behavior, helping identify optimization opportunities. Additionally, it enhances communication among stakeholders and serves as a blueprint for design, implementation, and maintenance.

Tools Required:
Rational Rose, Windows XP

Theory or Concept:
Collaboration diagrams are also relatively easy to draw. They show the relationship between objects and the order of messages passed between them. The objects are listed as icons and arrows indicate the messages being passed between them. The numbers next to the messages are called sequence numbers. As the name suggests, they show the sequence of the messages as they are passed between the objects. There are many acceptable sequence numbering schemes in UML. A simple 1, 2, 3... format can be used, as the example below shows, or for more detailed and complex diagrams a 1, 1.1 ,1.2, 1.2.1... scheme can be used.



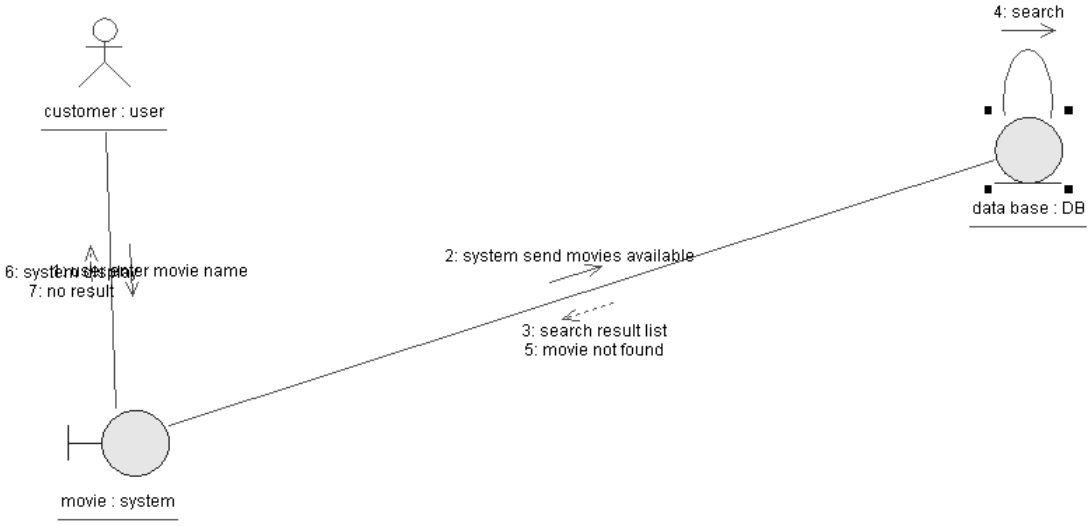
The example below shows a simple collaboration diagram for the placing an order use case. This time the names of the objects appear after the colon, such as: Order Entry Window following the objectName: className naming convention. This time the class name is shown to demonstrate that all of objects of that class will behave the same way.



Procedure:
Step 1: Create a new collaboration diagram
Step 2: Place objects on the canvas
Step 3: Draw links between objects
Step 4: Add messages
Step 5: Sequence Numbering
Step 6: Include conditions and iterations
Step 7: Verify roles and responsibilities

Step 8: Review and validate diagram

Output:



Exercise No. 5 Identify the analysis classes and depict the relationship between these classes.

Objective:
to provide a static view of the system by modeling its classes, attributes, methods, and relationships. It helps in understanding and designing the system's structure and serves as a blueprint for implementation. Additionally, it facilitates communication among stakeholders by clearly defining the system's components and their interactions.

Tools Required:
Rational Rose, Windows XP

Theory or Concept:
A class diagram is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes. Class diagrams show the classes of the system, their inter-relationships, and the operations and attributes of the classes. Class diagrams are typically used, although not all at once, to:
Explore domain concepts in the form of a domain model Analyze requirements in the form of a conceptual/analysis model
Depict the detailed design of object-oriented or object-based software A class model is comprised of one or more class diagrams and the supporting specifications that describe model elements including classes, relationships between classes, and interfaces. There are guidelines
1. General issues
2. Classes
3. Interfaces
4. Relationships
5. Inheritance
6. Aggregation and Composition

Because class diagrams are used for a variety of purposes – from understanding requirements to describing your detailed design – it is needed to apply a different style in each circumstance. This section describes style guidelines pertaining to different types of class diagrams.

CLASSES
A class in the software system is represented by a box with the name of the class written inside it. A compartment below the class name can show the class's attributes (i.e. its properties). Each attribute is shown with at least its name, and optionally with its type, initial value, and other properties.
A class is effectively a template from which objects are created (instantiated). Classes define attributes, information that is pertinent to their instances, and operations, functionality that the objects support. Classes will also realize interfaces (more on this later). Class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system, conceptual, specification, and implementation. These perspectives become evident as the diagram is created and help solidify the design.

INTERFACES
An interface is a collection of operation signature and/or attribute definitions that ideally defines a cohesive set of behaviors. Interface a class or component must implement the operations and attributes defined by the interface. Any given class or component may implement zero or more interfaces and one or more classes or components can implement the same interface.

RELATIONSHIPS
A relationship is a general term covering the specific types of logical connections found on a class and object diagram. Class diagrams also display relationships such as containment, inheritance, associations and others.

The association relationship is the most common relationship in a class diagram. The association shows the relationship between instances of classes.
Another common relationship in class diagrams is a generalization. A generalization is used when two classes are similar, but have some differences.

AGGREGATION

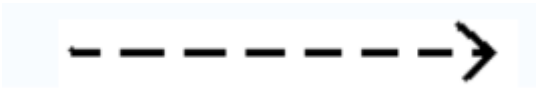
Aggregation is a variant of the "has a" or association relationship; composition is more specific than aggregation. Aggregation occurs when a class is a collection or container of other classes, but where the contained classes do not have a strong life cycle dependency on the container--essentially, if the container is destroyed, its contents are not.

ASSOCIATION

Associations are semantic connections between classes. When an association connects two classes Each class can send messages to others in a sequence or a collaboration diagram. Associations can be bidirectional or unidirectional.

DEPENDENCIES

Dependencies connect two classes. Dependencies are always unidirectional and show that one class, depends on the definitions in another class.



GENERALIZATION

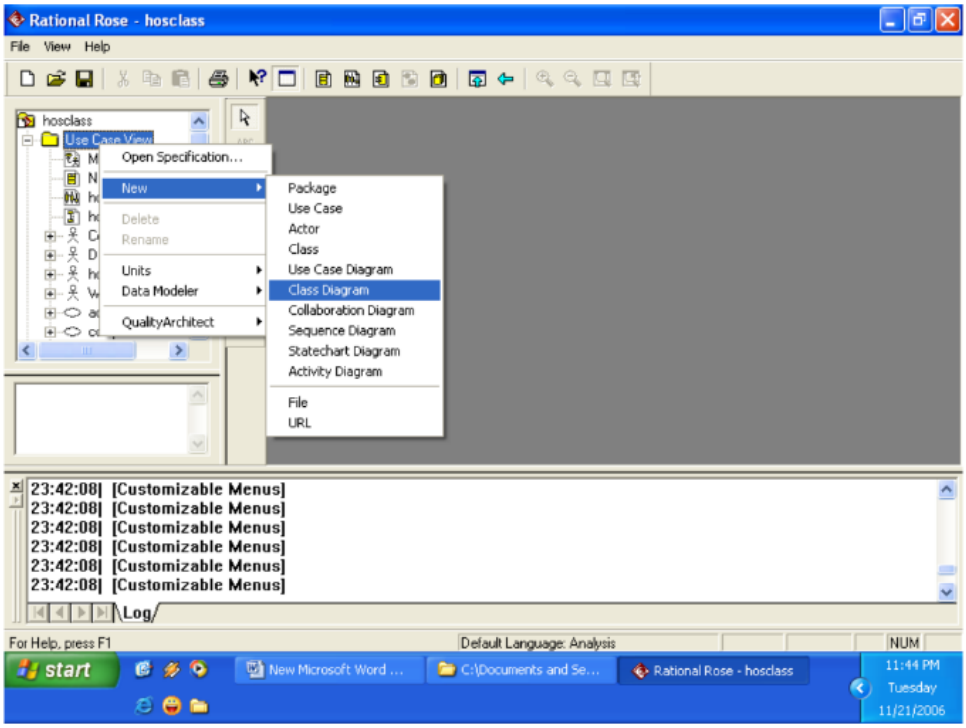
The generalization relationship indicates that one of the two related classes (the supertype) is considered to be a more general form of the other (the subtype). In practice, this means that any instance of the subtype is also an instance of the supertype. The generalization relationship is also known as the inheritance or "is a" relationship. The supertype in the generalization relationship is also known as the "parent", superclass, base class, or base type. The subtype in the generalization relationship is also known as the "child", subclass, derived class, derived type, inheriting class, or inheriting type.

MULTIPLICITY

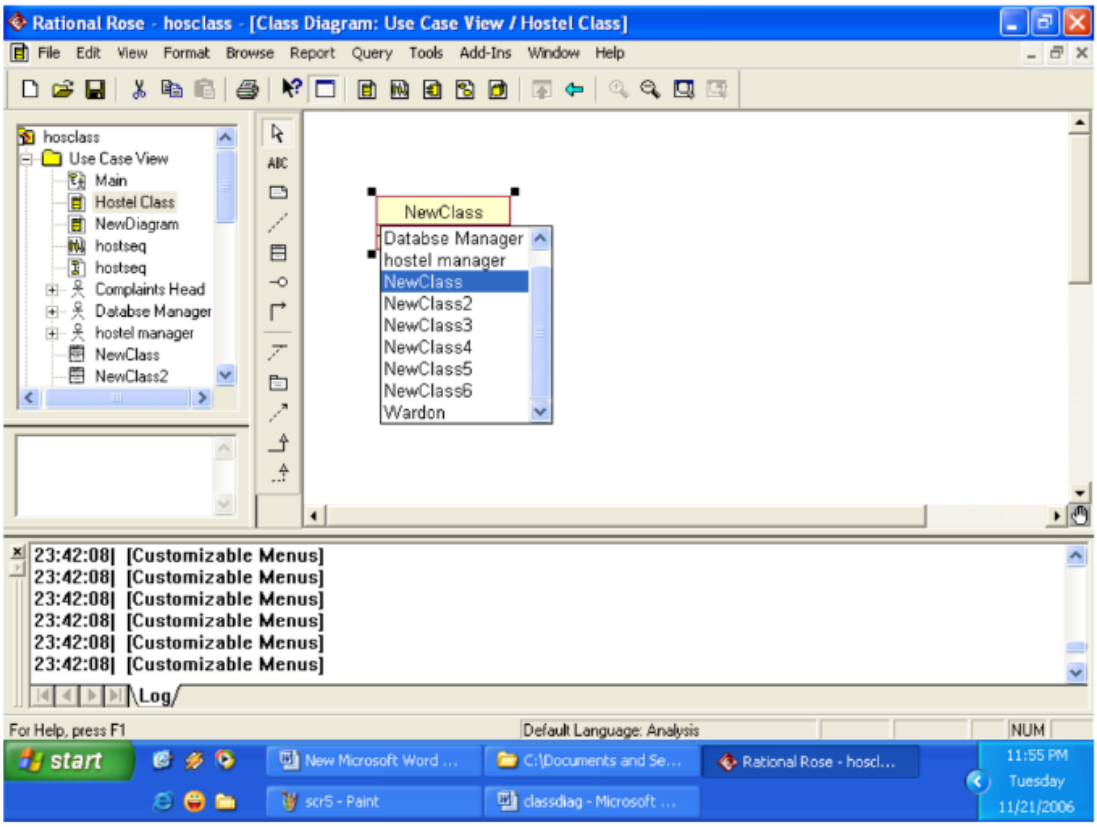
The association relationship indicates that (at least) one of the two related classes makes reference to the other.

Procedure:

Step 1: In Rational Rose, click on the click on the ‘use case view’ and selection new class diagram

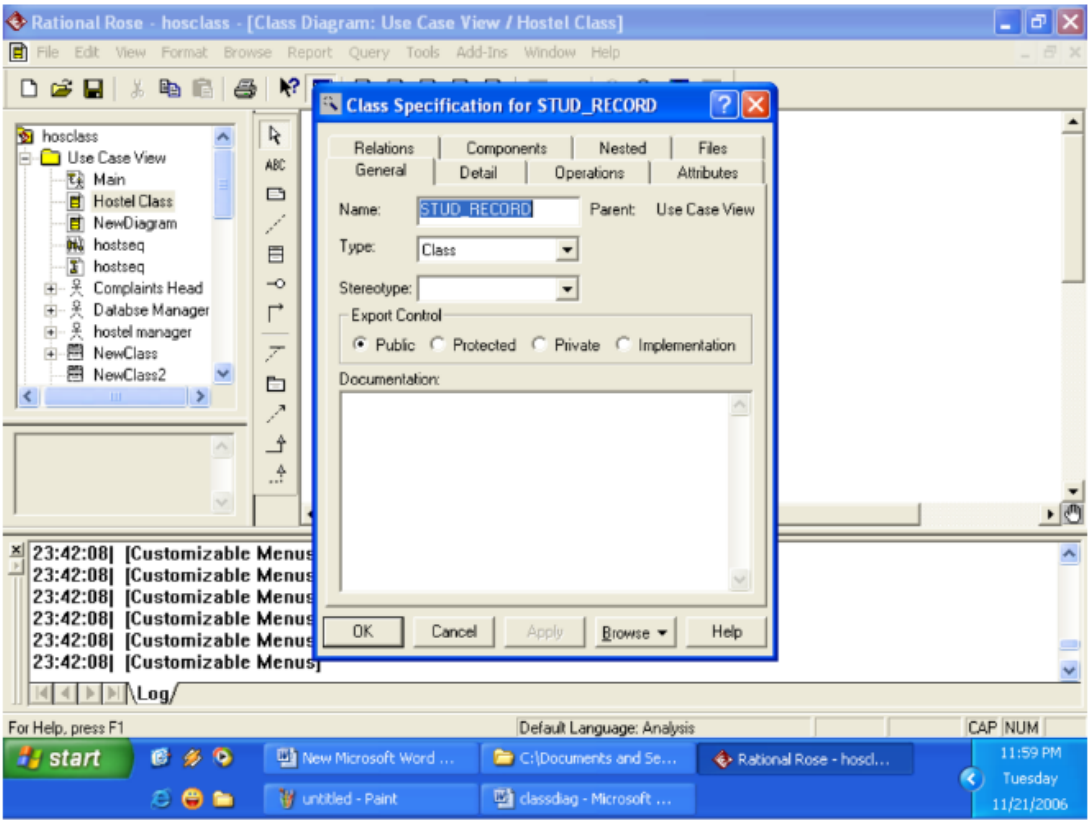


Step 2: Enter the class name in the dialog box

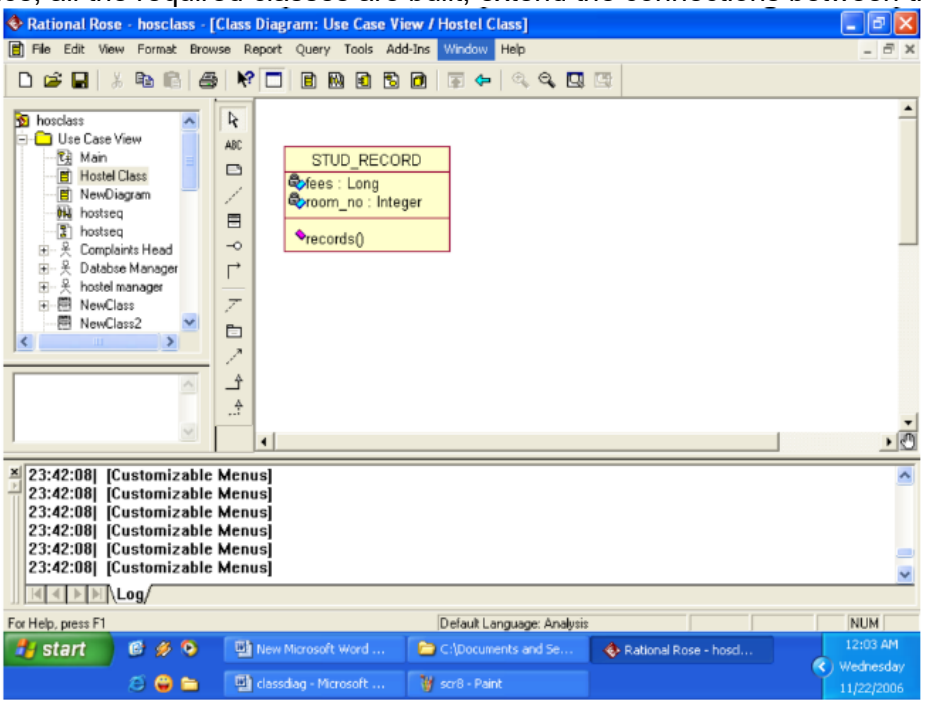


Step 3: Click the cursor on the block representing class from the table of predefined symbols into the screen

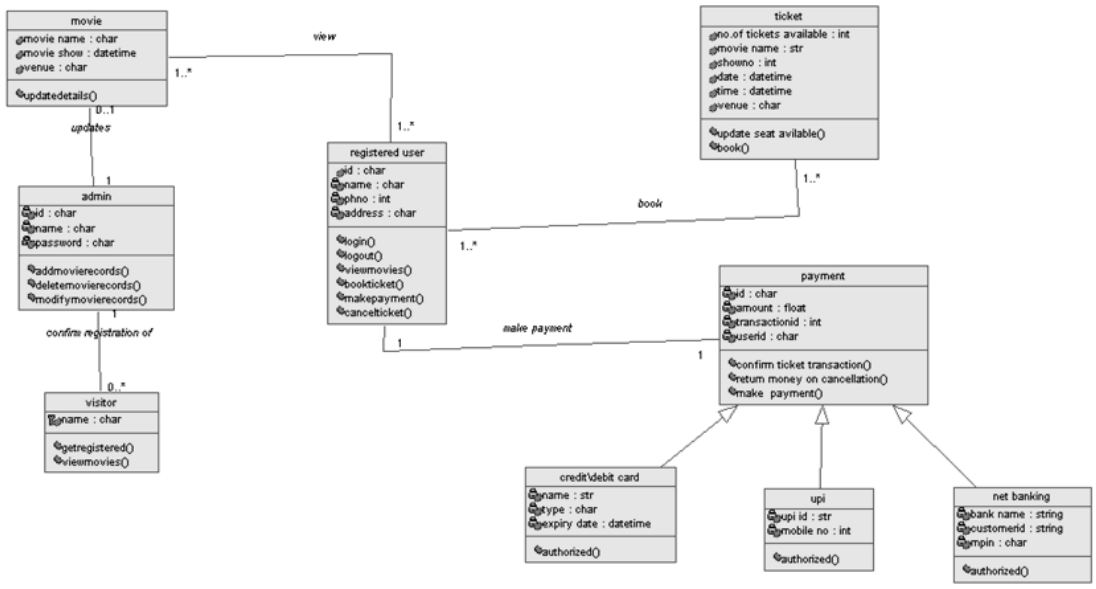
Step 4: Double click on the class and enter the class name, attributes and operations of the class



Step 5: Once, all the required classes are built, extend the connections between the classes



Output:


movie
movie name : char
movie show : datetime
venue : char
updatedetails()
view
1..*
1..*
0..1
updates
1
admin
aid : char
aname : char
apassword : char
addmovierecords()
deletemovierecords()
modifymovierecords()
1
confirm registration of
0..*
visitor
name : char
getregistered()
viewmovies()
registered user
uid : char
uname : char
uphno : int
uaddress : char
login()
logout()
viewmovies()
bookticket()
makepayment()
cancelticket()
1..*
book
1..*
ticket
no.of tickets available : int
movie name : str
showno : int
date : datetime
time : datetime
venue : char
update seat avilable()
book()
1..*
make payment
1
1
payment
pid : char
amount : float
transactionid : int
userid : char
confirm ticket transaction()
return money on cancellation()
make payment()
credit\debit card
cname : str
ctype : char
expiry date : datetime
authorized()
upi
upi id : str
mobile no : int
authorized()
net banking
bank name : string
customerid : string
mpin : char
authorized()

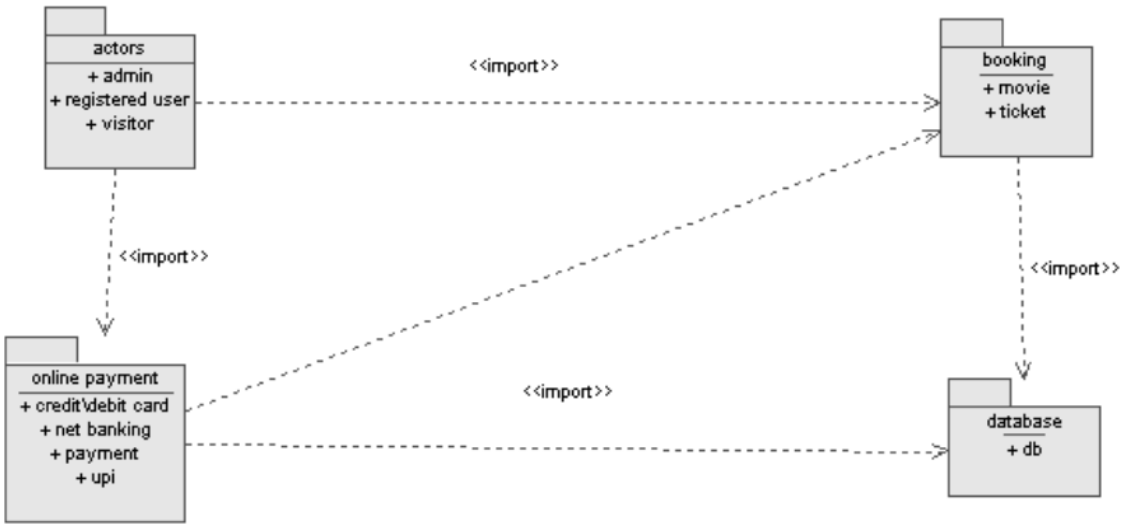
Exercise No. 6 Draw package diagram

Objective:
to organize and group related classes into packages, making the system's structure more modular and manageable. It helps in understanding and visualizing the dependencies and relationships between different packages. Additionally, it enhances the organization and maintainability of the system by clearly defining the package hierarchy and interactions.

Tools Required:
Rational Rose, Windows XP

Procedure:
Step 1: start->programs
Step 2: click on rational suite enterprise
Step 3: now click on the rational rose enterprise edition.
Step 4: select rational unified process from the displayed icons.
Step 5: right click on logical view and click on new and create package folders for each package.
Step 6: drag and drop the classes into packages.
Step 7: use icons and complete package diagram

Output:

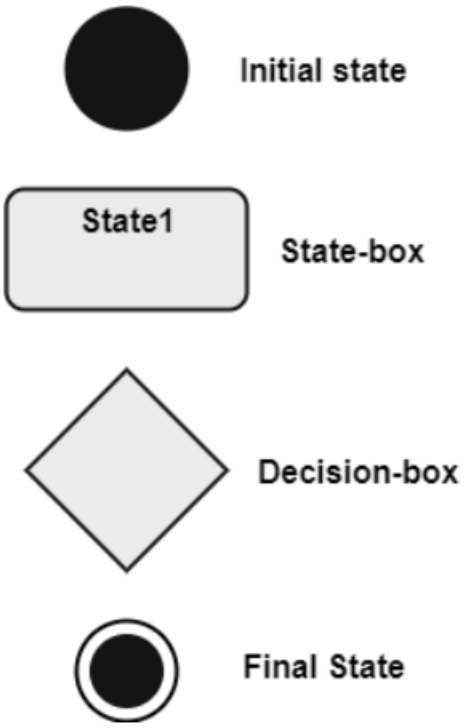


Exercise No. 7 Draw state diagram that represent the system as number of states and interactions between states.

Objective:
To draw a state diagram, identify the various states and transitions of the object or system along with the events that trigger these transitions. Define the actions that occur when entering or exiting states and the events that cause state changes. Finally, create a visual representation using standard symbols and notations to clearly illustrate the states, transitions, actions, and events

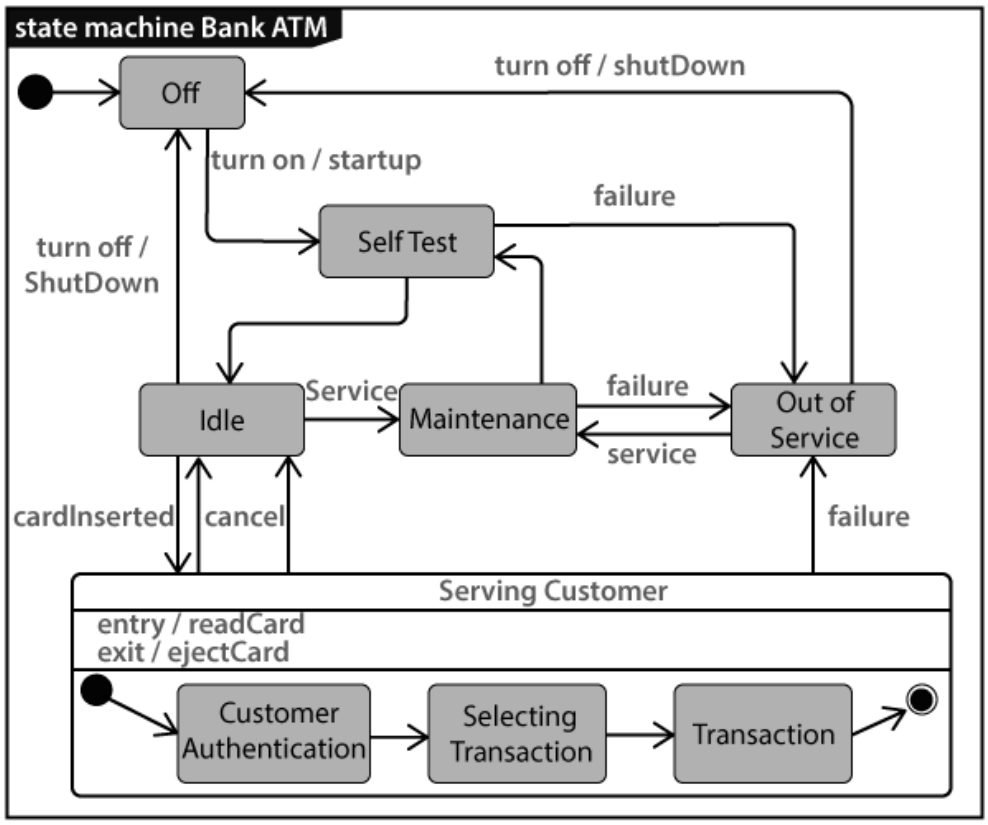
Tools Required:
Rational Rose, Windows XP

Theory or Concept:
A state diagram is a type of UML (Unified Modeling Language) diagram that represents the dynamic behavior of a system by showing its various states and the transitions between those states. It captures the possible states of an object or an entity throughout its lifecycle and the events or conditions that cause transitions from one state to another. State diagrams are useful for modeling the behavior of objects in response to external stimuli, making them valuable for understanding and designing complex systems, especially in areas like software engineering, process control, and protocol design.



Procedure:
Step 1: Launch Rational Rose and Create a New Project
Step 2: Create a State Diagram
Step 3: Identify and Define States and Transitions
Step 4: Add Actions and Events
Step 5: Refine and Review the diagram

Output:

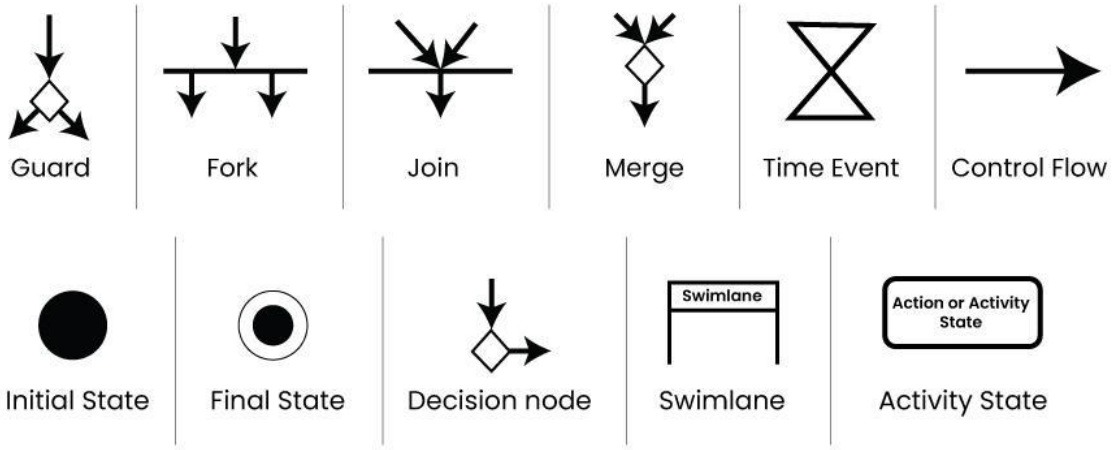

state machine Bank ATM
Off
turn off / shutDown
turn on / startup
failure
Self Test
turn off /
ShutDown
Idle
Service
Maintenance
failure
service
Out of
Service
cardInserted
cancel
failure
Serving Customer
entry / readCard
exit / ejectCard
Customer
Authentication
Selecting
Transaction
Transaction

Exercise No. 8 Represent the workflow of the systems using activities and actions using activity diagram

Objective:
to model the workflow of a system or process by identifying key activities and their sequences. Define the actions, decisions, and parallel processes involved, along with the transitions between them. Create a clear visual representation to understand and communicate the flow of activities and their dependencies

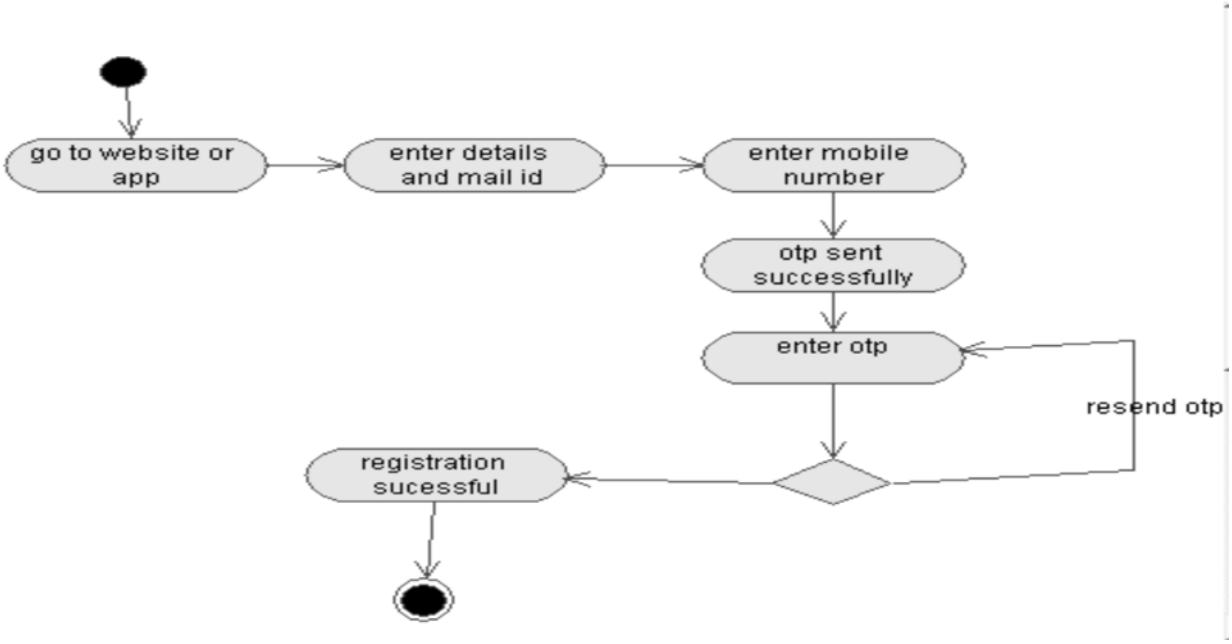
Tools Required:
Rational Rose, Windows XP

Theory or Concept:
An activity diagram is a type of UML (Unified Modeling Language) diagram that represents the dynamic aspects of a system by depicting the flow of control or data from one activity to another. It models the sequence of activities, including actions, decisions, parallel processes, and their dependencies, providing a visual representation of the workflow or business process. Activity diagrams are useful for analyzing and designing the detailed logic of complex processes, highlighting how tasks are carried out, and identifying potential bottlenecks or inefficiencies.



Procedure:
Step 1: Launch Rational Rose and Create a Project
Step 2: Create an Activity Diagram
Step 3: Identify Activities and their Sequence
Step 4: Add Initial and Final nodes
Step 5: Add Activities
Step 6: Connect Activities with Transitions
Step 7: Include Decision and Merge Nodes
Step 8: Add Fork and Join Nodes
Step 9: Refine and Review the diagram

Output:

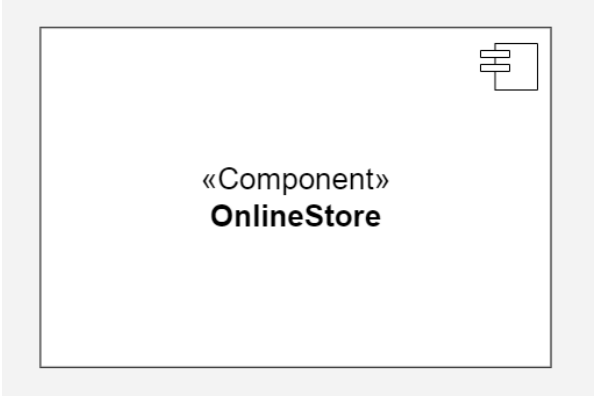

go to website or app
enter details and mail id
enter mobile number
otp sent successfully
enter otp
resend otp
registration sucessful

Exercise No. 9 Draw component diagram to represent both the physical and logical aspects of the system

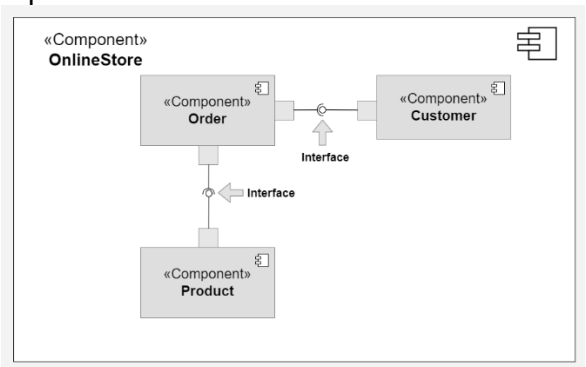
Objective:
to visualize the physical components of a system and their interdependencies. It helps in understanding and documenting the architecture by showing how components, such as software modules, libraries, and subsystems, interact and are organized. This aids in ensuring proper component functionality, facilitating communication among stakeholders, and guiding system development and maintenance.

Tools Required:
Rational Rose, Windows XP

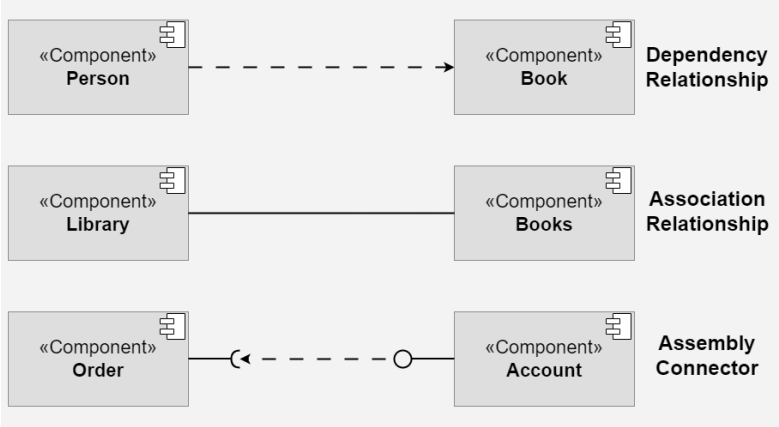
Theory or Concept:
A component diagram is a type of UML (Unified Modeling Language) diagram that depicts the organization and dependencies among the physical components of a system. It shows how software components, such as modules, libraries, executables, and their interfaces, are connected and interact with each other. Component diagrams provide a high-level view of the system architecture, highlighting the structure and relationships between components, making them essential for understanding system organization, ensuring proper functionality, and guiding development and maintenance efforts. They are particularly useful for visualizing the implementation details and managing complex systems by illustrating the components' interdependencies and communication paths.
Component: Represent modular parts of the system that encapsulate functionalities. Components can be software classes, collections of classes, or subsystems.



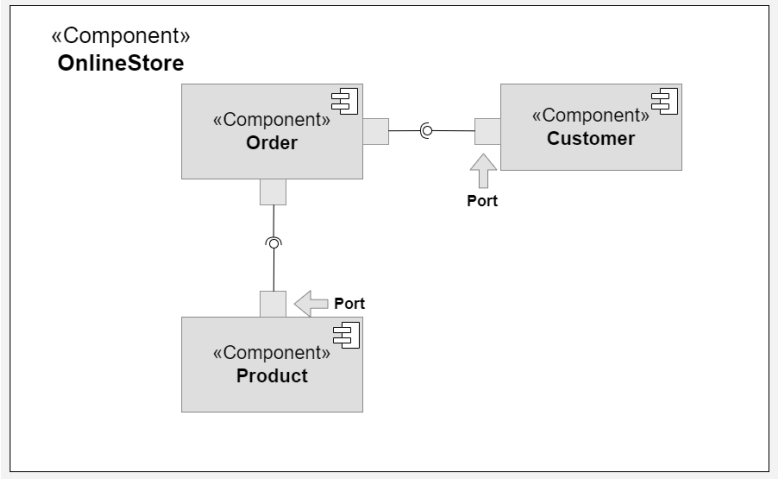
Interfaces: Specify a set of operations that a component offers or requires, serving as a contract between the component and its environment



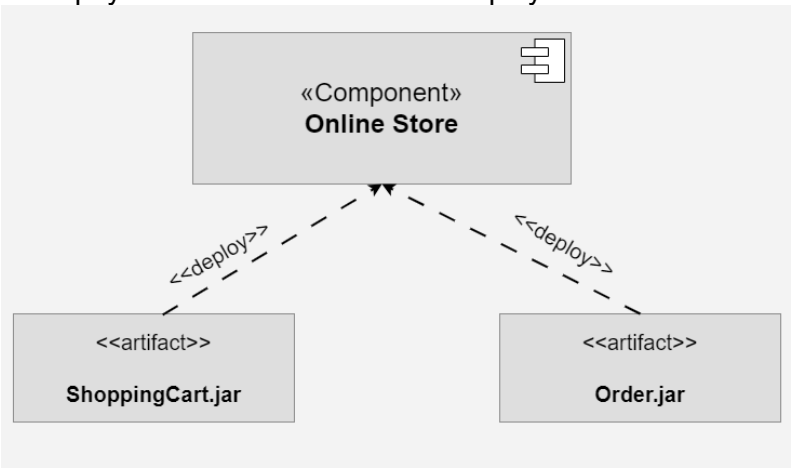
Relationships: Depict the connections and dependencies between components and interfaces.



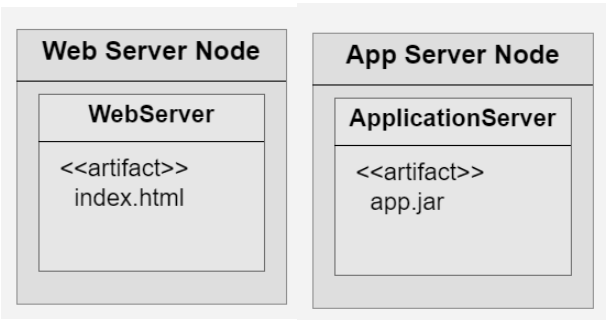
Ports: Represent specific interaction points on the boundary of a component where interfaces are provided or required.



Artifacts: Represent physical files or data that are deployed on nodes.

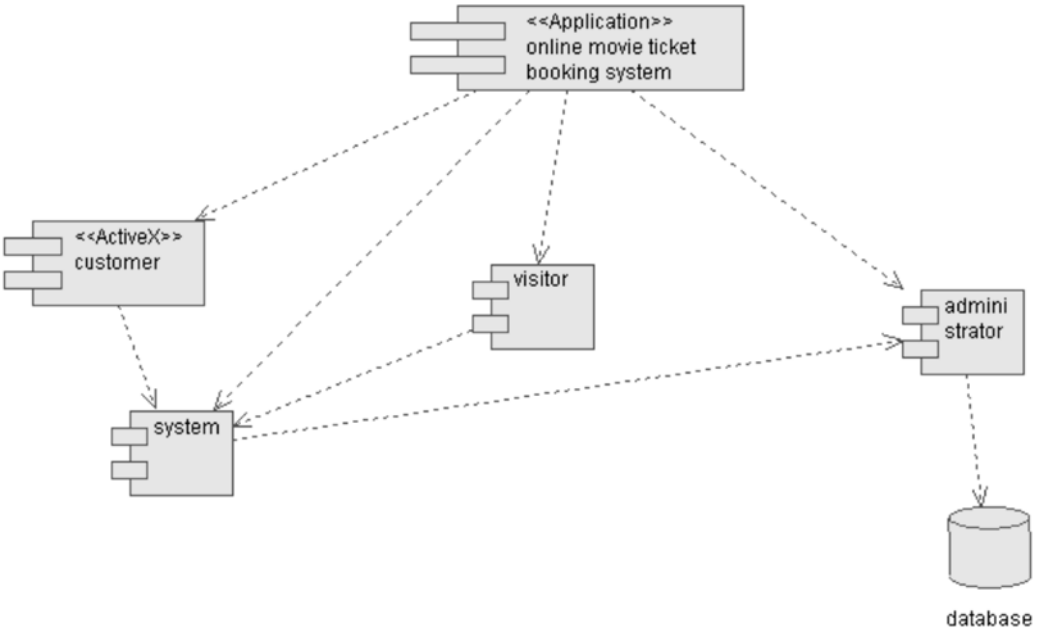


Nodes: Represent physical or virtual execution environments where components are deployed.



Procedure:
Step 1: Launch Rational Rose and Create a Project
Step 2: Create a Component Diagram
Step 3: Identify Components
Step 4: Add Components
Step 5: Define Interfaces
Step 6: Specify Relationships
Step 7: Add Connectors and Ports
Step 8: Refine and Review the diagram

Output:



Exercise No. 10 Construct deployment that shows the execution architecture of a system

Objective:
to illustrate the physical deployment of software components across hardware nodes within a system or network. It aims to depict how software artifacts, such as executables, databases, and services, are distributed across physical nodes like servers, workstations, and devices.

Tools Required:
Rational Rose, Windows XP

Theory or Concept:
Deployment refers to the process of making a software application operational on a specific computing environment, such as servers, virtual machines, or mobile devices. It involves installing, configuring, testing, and launching the application to ensure it functions correctly in its intended environment. Deployment encompasses tasks like setting up hardware infrastructure, deploying software components, managing dependencies, and configuring network settings. It is crucial for ensuring the reliability, availability, and scalability of the application while meeting performance requirements. Effective deployment practices streamline operations, facilitate updates and maintenance, and support the seamless integration of new features or improvements into the operational environment.
Component: represents a modular and reusable part of a system, typically implemented as a software module, class, or package. It encapsulates its behavior and data and can be deployed independently.

COMPONENT

Artifact: represents a physical piece of information or data that is used or produced in the software development process. It can include source code files, executables, documents, libraries, configuration files, or any other tangible item.

ARTIFACT

Interface: defines a contract specifying the methods or operations that a component must implement. It represents a point of interaction between different components or subsystems.

INTERFACE

Interface: defines a contract specifying the methods or operations that a component must implement. It represents a point of interaction between different components or subsystems.

Procedure:
Step 1: Launch Rational Rose and Create a Project
Step 2: Create Deployment Diagram
Step 3: Identify Nodes
Step 4: Add Nodes
Step 5: Identify Software Components
Step 6: Add Components to Nodes
Step 7: Specify Relationships and Connections
Step 8: Add Communication Paths
Step 9: Refine and Review the Diagram

Output: