SRI ESHWAR COLLEGE OF ENGINEERING

KINATHUKADAVU, COIMBATORE-641202

(An Autonomous Institution Affiliated to Anna University, Chennai)

SOFTWARE ENGINEERING LABORATORY RECORD

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SRI ESHWAR COLLEGE OF ENGINEERING

KINATHUKADAVU COIMBATORE - 641202

SRI ESHWAR COLLEGE OF ENGINEERING

KINATHUKADAVU, COIMBATORE-641202

(An Autonomous Institution Affiliated to Anna University, Chennai)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

Certified that this is the bonafide record of work done by

Name: Mr. /Ms. ...……

Register No: ..... of 2nd Year B.E – Computer Science and Engineering in the **U23IT481 - SOFTWARE ENGINEERING LABORATORY** during the **3rd Semester** of the academic year **2024 – 2025 (Odd Semester).**

Signature of Staff In-charge

Head of the department

Submitted for the practical examinations of Anna University, held on …….

Internal Examiner

External Examiner

Contents

S.No	Date	Name of the Experiment	Page Number	Marks (50)	Signature of the Faculty Member
1		Define the problem statement for the given project.			
2		Identifying the requirements from problem statements.			
3		E-R Modelling from the problem statements.			
4		Modelling Data Flow Diagrams			
5		Modelling UML Use Case Diagrams and capturing Use Case Scenarios			
6		State chart and Activity Modelling			
7		Identifying Domain Classes from the problem statements			
8		Modelling UML Class Diagrams and Sequence Diagrams			
9		Designing Test Suites			
10		Implementation and testing of the modified system using Sonar Cloud			

Average:

Average (in words) :

Signature of the Faculty

INTRODUCTION TO THE LAB

Requirement of the lab

Hardware Requirements:

Pentium 4 processor (2.4 GHz),

128 Mb RAM,

Standard keyboard n mouse,

colored monitor.

Software Requirements:

Rational Rose/Argo UML/Star UML/Visual Paradigm

Windows XP/2000,

MS-office.

This lab deals with the analysis and design of a software problem. The tool used in a lab is Star UML this tool is used for an object-oriented design of a problem. We draw a UML diagram in a Star UML which deals with the objects and classes in a system. The **Unified Modeling Language** or **UML** is a mostly graphical modelling language that is used to express designs. It is a standardized language in which to specify the artefacts and components of a software system. It is important to understand that the UML describes a notation and not a process. It does not put forth a single methodor process of design, but rather is a standardized tool that can be used in a design process.

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.[1] The UML is a very important part of developing object oriented software and the software development process.

The UML uses mostly graphical notations to express the design of software projects. Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

Goals of UML

The primary goals in the design of the UML were:

1. Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts such as collaborations, frameworks, patterns and components.
7. Integrate best practices.

Why Use UML?

As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software and to improve quality and reduce cost and time-to-market. These techniques include component technology, visual programming, patterns and frameworks.

Businesses also seek techniques to manage the complexity of systems as they increase in scope and scale. In particular, they recognize the need to solve recurring architectural problems, such as physical distribution, concurrency, replication, security, load balancing and fault tolerance.

SOFTWARE ENGINEERING LAB

Paper Code: U23IT481

Paper: Software Engineering Lab

Tools Required: ArgoUML/StarUML/Visual Paradigm/Rational Rose Enterprise Edition

List of Experiments:

1. Define the problem statement for the given project
2. Identifying the requirements from problem statements
3. Modelling UML Use Case Diagrams and capturing Use Case Scenarios
4. E-R Modelling from the problem statements
5. Identifying Domain Classes from the problem statements
6. State chart and Activity Modelling
7. Modelling UML Class Diagrams and Sequence Diagrams
8. Modelling Data Flow Diagrams
9. Designing Test Suites
10. Implementation and testing of the modified system using Sonar Cloud

Text Books:

1. K.K. Aggarwal & Yogesh Singh, ―Software Engineering‖, New Age International, 2005

2. Pankaj Jalote, ―An Integrated Approach to Software Engineering‖, Second Edition, Springer.

List of Projects:

a) Digitalized Secure Banking.

b) Ecotourism management system.

c) Natural Resources utilization management system for Agricultural Development.

d) Fisheries Resource Management System.

e) Autonomous Robot Aided Agriculture.

f) E-Waste Recycling System.

g) Railway Train Ticket Generation.

h) Coffee Vending system.

i) Robotic Vacuum Cleaning system.

j) Insurance Management system.

k) Primary Health Centre (PHC) Monitoring and Management System.

l) Automated Healthcare monitoring system.

m) Asian Tourism Management system.

n) RFID based security system.

o) Inventory Management System for Car accessories.

p) Automated Food Ordering System.

q) Loan Automation System.

r) Investment scheme Guidelines System.

s) Sports Event Management System.

t) Automated Farming Assistance system.

EX. NO: 1 Date:	PREPARE PROBLEM STATEMENT FOR ANY PROJECT

AIM: To prepare problem statement for any project.

REQUIREMENTS:

Hardware Interfaces

- Pentium(R) 4 CPU 2.26 GHz, 128 MB RAM
- Screen resolution of at least 800 x 600 required for proper and complete viewing ofscreens. Higher resolution would not be a problem.
- CD ROM Driver

Software Interfaces

- Any window-based operating system (Windows 95/98/2000/XP/NT)
- WordPad or Microsoft Word

THEORY:

The problem statement is the initial starting point for a project. It is basically a one to three-page statement that everyone on the project agrees with that describes what will be done at a high level. The problem statement is intended for a broad audience and should be written in non- technical terms. It helps the non-technical and technical personnel communicate by providing a description of a problem. It doesn't describe the solution to the problem.

The input to requirement engineering is the problem statement prepared by customer.

It may give an overview of the existing system along with broad expectations from the new system.The first phase of requirements engineering begins with requirements elicitation i.e. gathering of information about requirements. Here, requirements are identified with the help of customer and

existing system processes. So from here begins the preparation of problem statement.

So, basically a problem statement describes **what** needs to be done without describing **how**.

Conclusion:

The problem statement was written successfully by following the steps described above.

EX. NO: 2 Date:	SOFTWARE REQUIREMENT SPECIFICATION(SRS)

Aim: Understanding an SRS.

Requirements:

Hardware Requirements:

- PC with 300 megahertz or higher processor clock speed recommended; 233 MHz minimumrequired.
- 128 megabytes (MB) of RAM or higher recommended (64 MB minimum supported)
- 1.5 gigabytes (GB) of available hard disk space
- CD ROM or DVD Drive
- Keyboard and Mouse (compatible pointing device).

Software Requirements:

- Star UML, Windows XP,

Theory:

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 clientand 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 capabilitiesa 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 andvalidation 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 requirementsto 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 theparent 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 requirementsare 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 afterthe requirements have been gathered and analyzed.

SRS should address the following

The basic issues that the SRS shall address are the following:

Functionality. What is the software supposed to do?

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

Performance. What is the speed, availability, response time, recovery time of various softwarefunctions, etc.?

Attributes. What are the portability, correctness, maintainability, security, etc. considerations?

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:

A SRS should be

- Correct
- Unambiguous
- Complete
- Consistent
- Ranked for importance and/or stability
- Verifiable
- Modifiable
- 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 notcorrect.

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 wishlists. Some may not be achievable. It is useful provide this information in the SRS.

Verifiable - Don't put in requirements like - "It should provide the user a fast response." Another ofmy favorites is - "The system should never crash." Instead, provide a quantitative requirement like:"Every key stroke should provide a user response within 100 milliseconds."

Modifiable - Having the same requirement in more than one place may not be wrong - but tends tomake 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?

An SRS Outline

1. Introduction

1.1 Product Scope

1.2 Product Value

1.3 Intended Audience

1.4 Intended Use

1.5 General Description

2. Functional Requirements

2.1 User authentication and profile management

2.2 Menu display and customization

2.3 Order processing

2.4 Payment integration

2.5 Order tracking

2.6 Admin panel

3. External Interface Requirements

3.1 User interface requirements

3.2 Hardware interface requirements

3.3 Technology Specification

3.4 Communication interface requirements

4. Non-functional Requirements

4.1 Security

4.2 Performance

4.3 Compatibility

4.4 Reliability

4.5 Scalability

4.6 Maintainability

4.7 Usability

4.8 Other

5. Other Requirements

Definitions and Acronyms.

Conclusion:

The SRS was made successfully by following the steps described above.

SOFTWARE REQUIREMENTS SPECIFICATION

AUTOMATED FOOD ORDERING SYSTEM

1. Introduction

The software AFOS is to be developed for the Automated Food Ordering System (AFOS). An Automated Food Ordering System is a computerized application designed to streamline the process of ordering food in restaurants. It provides customers with a secure and efficient method to browse menus, customize orders, and make payments without the need for direct interaction with restaurant staff. Through AFOS, users interact with a user-friendly interface that enables them to place orders, track their status, and complete payments seamlessly, improving the overall dining experience.

1.1 Product scope

The Automated Food Ordering System **(AFOS)** is designed to streamline the process of ordering food in restaurants by enabling customers to browse menus, place orders, and make payments through an intuitive digital platform. This system aims to enhance the dining experience, minimize order errors, and optimize the efficiency of restaurant operations.

1.2 Product value

The AFOS offers significant value to restaurants and customers by:

- Reducing the time required for placing orders.
- Eliminating human error in the order-taking process.
- Enhancing customer satisfaction through a seamless, self-service interface.
- Increasing restaurant productivity and order processing efficiency.

1.3 Indented Audience

The intended audience for this SRS includes:

- Restaurant owners and managers who are considering adopting the AFOS.
- Software developers and engineers involved in the design and implementation of the system.
- Quality assurance teams responsible for testing the AFOS.
- End users (customers) who will interact with the system for ordering.

1.4 Indented use

The AFOS will be used for:

- Browsing restaurant menus and customizing food orders.
- Placing orders directly from a digital device (e.g., tablets or smartphones).
- Making payments securely through various payment options.
- Managing and tracking orders by restaurant staff.

1.5 General description

The AFOS is a user-friendly application that integrates with restaurant POS (Point of Sale) systems to facilitate the food ordering process. It features a responsive design that supports multiple device types and provides real-time updates on order status.

2. Functional requirements

2.1 User authentication and profile management:

Users can create and manage their profiles for personalized experiences.

2.2 Menu display and customization:

Dynamic menus that allow users to browse items, view descriptions, and customize orders.

2.3 Order processing:

Real-time order submission to the restaurant's kitchen system.

2.4 Payment integration:

Support for multiple payment methods, including credit/debit cards and digital wallets.

2.5 Order tracking:

Notifications and updates on the order status for customers.

2.6 Admin panel:

A control interface for restaurant staff to manage orders, update menus, and monitor operations.

3. External interface requirements

3.1 User interface requirements

- The system must have a responsive design for optimal use on desktops, tablets, and mobile devices.
- The interface should provide clear, user-friendly navigation for browsing menus, placing orders, and making payments.
- Visual feedback (e.g., notifications, confirmation dialogs) should be provided to users at each step of the ordering process.

3.2 Hardware interface requirements

- The system should be compatible with devices such as tablets and smartphones.
- Integration with POS terminals for seamless order processing and payment handling.

3.3 Technology Specification

- **Frontend**: React, Angular, or other modern JavaScript frameworks.
- **Backend**: Node.js, Python (Django/Flask), or Java (Spring Boot).
- **Database**: MySQL, PostgreSQL, or MongoDB for managing user and order data.
- **Payment Gateway**: Integration with Stripe, PayPal, or local banking APIs.
- **Cloud Infrastructure**: AWS, Azure, or Google Cloud for scalable deployment.

3.4 Communication interface requirements

- The system should support RESTful APIs for communication between frontend and backend services.
- Secure sockets (HTTPS) for encrypted communication and data protection.
- WebSocket for real-time order status updates and notifications.

4. Non-functional requirements

4.1 Security

- The system must implement user authentication (e.g., OAuth 2.0, JWT).
- Data encryption must be applied for sensitive user data.
- Compliance with PCI-DSS standards for payment processing.

4.2 Performance

- The system should handle at least 1,000 simultaneous users without significant performance degradation.
- Response time for order submission should be under 2 seconds.

4.3 Compatibility

- The application should be compatible with modern web browsers (e.g., Chrome, Safari, Firefox).
- It must support both Android and iOS operating systems for mobile devices.

4.4 Reliability

- The system should ensure a 99.9% uptime to maintain availability.
- Failover mechanisms should be implemented to prevent data loss.

4.5 Scalability

- The system should be designed to scale horizontally to accommodate increased traffic and user growth.
- Microservices architecture is recommended for independent scalability of different modules.

4.6 Maintainability

- The codebase should follow industry standards and best practices for maintainability.
- Clear documentation should be provided for future enhancements and debugging.

4.7 Usability

- The interface should be intuitive and require minimal training for users.
- A feedback mechanism should be available for users to report issues and suggest improvements.

4.8 Other

- Localization support for multiple languages.
- Accessibility features for users with disabilities (e.g., screen reader compatibility).

5. Other requiremetns

None.

Definitions and acronyms

AFOS	Automated Food Ordering System
POS	Point of Sale
JWT	JSON Web Token

EX. NO: 3 Date:	ENTITY RELATIONSHIP DIAGRAM

AIM :

To draw a sample entity relationship diagram for real project or system.

Hardware Requirements:

Pentium 4 processor (2.4 GHz), 128 Mb RAM, Standard keyboard n mouse, colored monitor.

Software Requirements:

Star UML, Windows XP,

THEORY

Entity Relationship Diagrams are a major data modelling tool and will help organize the data in your project into entities and define the relationships between the entities. This process has proved to enable the analyst to produce a good database structure so that the data can be stored and retrieved in a most efficient manner.

Entity

A data entity is anything real or abstract about which we want to store data. Entity types fall into five classes: roles, events, locations, tangible things or concepts. E.g. employee, payment, campus, book. Specific examples of an entity are called **instances.** E.g. the employee John Jones, Mary Smith's payment, etc.

Relationship

A data relationship is a natural association that exists between one or more entities. E.g. Employees process payments. **Cardinality** defines the number of occurrences of one entity for a single occurrence of the related entity. E.g. an employee may process many payments but might not process any payments depending on the nature of her job.

Attribute

A data attribute is a characteristic common to all or most instances of a particular entity. Synonyms include property, data element, field. E.g. Name, address, Employee Number, pay rate are all attributes of the entity employee. An attribute or combination of attributes that uniquely identifies one and only one instance of an entity is called a **primary key** or **identifier**.

E.g.

Employee Number is a primary key for Employee.

AN ENTITY RELATIONSHIP DIAGRAM METHODOLOGY: (One way of doing it)

1. Identify Entities	Identify the roles, events, locations, tangible things or concepts about which the end-users want to store data.
2. Find Relationships	Find the natural associations between pairs of entities using a relationship matrix.
3. Draw Rough ERD	Put entities in rectangles and relationships on line segments connecting the entities.
4. Fill in Cardinality	Determine the number of occurrences of one entity for a single occurrence of the related entity.
5. Define Primary Keys	Identify the data attribute(s) that uniquely identify one and only one occurrence of each entity.
6. Draw Key-Based ERD	Eliminate Many-to-Many relationships and include primary and foreign keys in each entity.
7. Identify Attributes	Name the information details (fields) which are essential to the system under development.
8. Map Attributes	For each attribute, match it with exactly one entity that it describes.
9. Draw fully attributed ERD	Adjust the ERD from step 6 to account for entities or relationships discovered in step 8.
10. Check Results	Does the final Entity Relationship Diagram accurately depict the system data?

AUTOMATED FOOD ORDERING SYSTEM

A restaurant operates an automated food ordering system to streamline the process of managing customers, orders, and staff. Customers can place orders consisting of one or more menu items, which are prepared by chefs. Orders are either delivered by delivery staff or picked up by the customer. Payments are linked to orders, and menu items are categorized by type and availability. The system efficiently manages staff roles and ensures smooth order processing and payment handling.

1. Identify Entities

The entities in this system are Customer, Order, MenuItem, Payment, DeliveryStaff, Chef. True entities must have more than one instance.

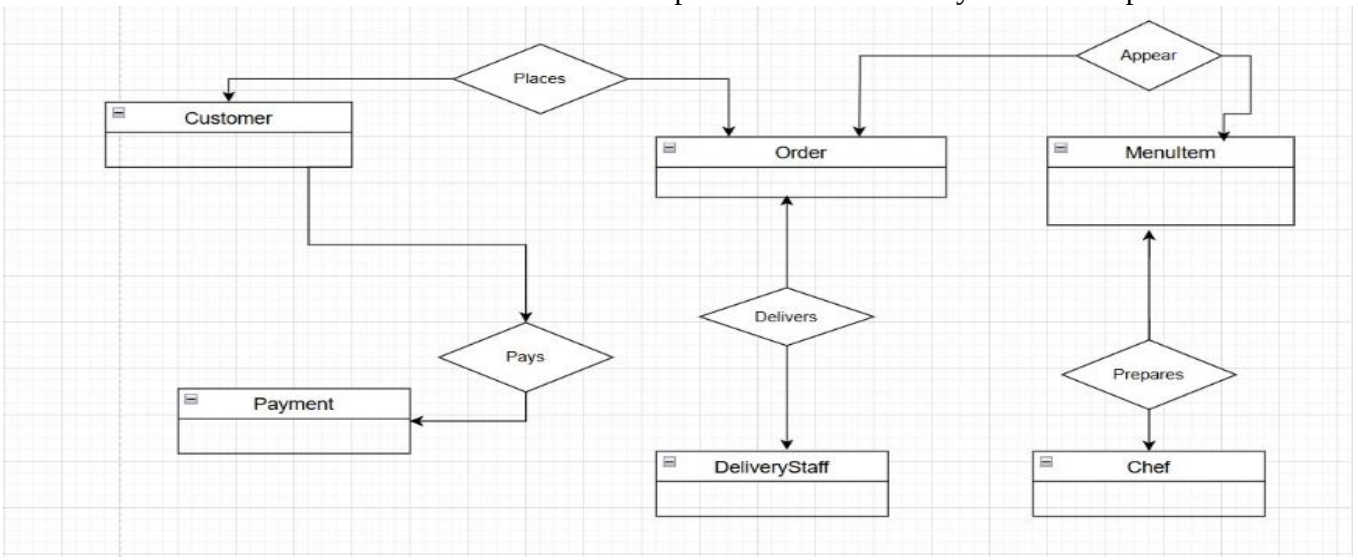
2. Find Relationships

We construct the following Entity Relationship Matrix:

	Customer	Order	MenuItem	Payment	DeliveryStaff	Chef
Customer		Places		Pays		
Order	Placed by		Appear		Delivered by	
MenuItem		Appeared in				Prepared by
Payment	Paid by					
DeliveryStaff		Delivers				
Chef			Prepares			

3. Draw Rough ERD

We connect the entities whenever a relationship is shown in the entity Relationship Matrix.

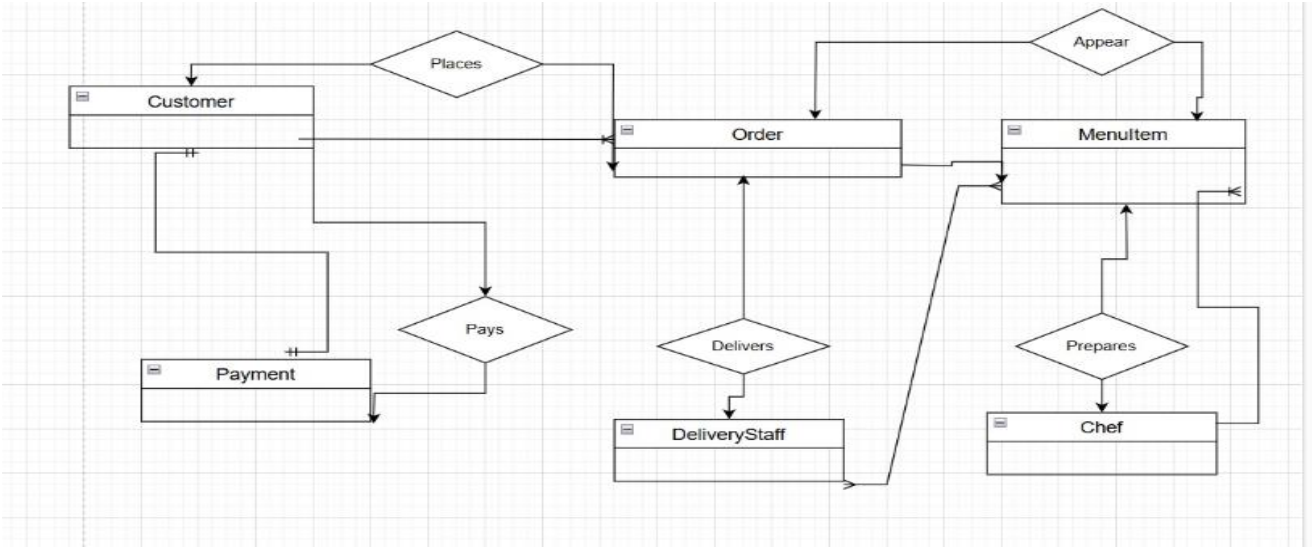


4. Fill in Cardinality

From the description of the problem we see that:

- Each customer can place 0 or more orders
- Each order is placed by exactly one customer.

- Each order consists of one or more menu items.
- A menu item can appear in 0 or more orders.
- Each order is linked to exactly one payment.
- Each payment is linked to exactly one order.
- Each order is prepared by at least one chef.
- Each chef can prepare 0 or more orders.
- Each order is delivered by exactly one delivery staff member.
- Each delivery staff member can deliver 0 or more orders.



5. Define Primary Keys

The primary keys are User_id, OrderId, ItemId, Payment.

6. Draw Key-Based ERD

In the Automated Food Ordering System, several relationships require key-based ERD management. The **Order-MenuItem** relationship is many-to-many, requiring an associative entity OrderItem with a primary key of OrderId + ItemId. The **Order-Customer** and **Order-DeliveryStaff** relationships are one-to-many, so no associative entities are needed for these. The **Order-Chef** relationship is many-to-many, requiring an associative entity OrderChef with a primary key of OrderId + ChefId. These associative entities help manage the many-to-many relationships in the system's database design.

7. Identify Attributes

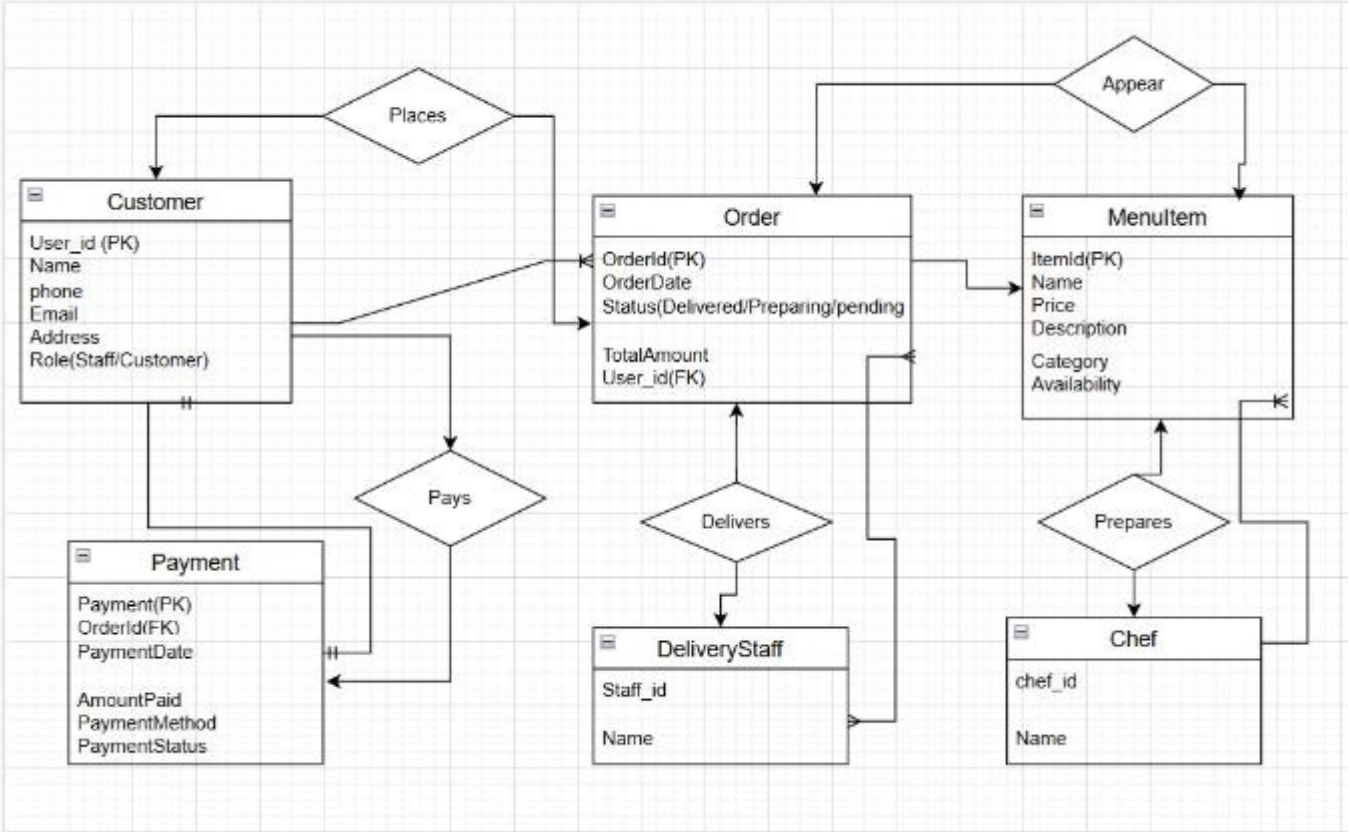
In the Automated Food Ordering System, the attributes identified for each entity include **Order** (OrderId, OrderDate, OrderStatus), **MenuItem** (ItemId, ItemName, ItemPrice), **Customer** (CustomerId, CustomerName, CustomerAddress, CustomerPhone), **DeliveryStaff**

(StaffId, StaffName, StaffPhone), **Chef** (ChefId, ChefName, ChefSpecialty), as well as the associative entities **OrderItem** (OrderId, ItemId) and **OrderChef** (OrderId, ChefId). These attributes represent the key data elements required to manage orders, menu items, customers, delivery staff, chefs, and their relationships in the system.

8. Map Attributes

Attribute	**Entity**	**Attribute**	**Entity**
User_id	Customer	Chef_id	Chef
Name	Customer	Chef Name	Chef
Email	Customer	Staff_id	DeliveryStaff
Address	Customer	Staff Name	DeliveryStaff
Role(Staff/Customer)	Customer	OrderId	Order
ItemId	MenuItem	OrderDate	Order
Item Name	MenuItem	Status	Order
Price	MenuItem	Total Amount	Order
Description	MenuItem	Payment Id	Payment
PaymentDate	Payment	AmountPaid	Payment

9. Draw Fully Attributed ERD



10. Check Results

The final ERD appears to model the data in this system well.

FURTHER DISCUSSION:

Step 1. Identify Entities

A data entity is anything real or abstract about which we want to store data. Entity types fall into five classes: roles, events, locations, tangible things, or concepts. The best way to identify entities is to ask the system owners and users to identify things about which they would like to capture, store and produce information. Another source for identifying entities is to study the forms, files, and reports generated by the current system. E.g. a student registration form would refer to Student (a role), but also Course (an event), Instructor (a role), Advisor (a role), Room (a location), etc.

Step 2. Find Relationships

There are natural associations between pairs of entities. Listing the entities down the left column and across the top of a table, we can form a relationship matrix by filling in an active verb at the intersection of two entities which are related. Each row and column should have at least one relationship listed or else the entity associated with that row or column does not interact with the rest of the system. In this case, you should question whether it makes sense to include that entity in the system.

. A student is enrolled in one or more
courses subject verb objects

Step 3. Draw Rough ERD

Using rectangles for entities and lines for relationships, we can draw an Entity Relationship Diagram (ERD).

Step 4. Fill in Cardinality

At each end of each connector joining rectangles, we need to place a symbol indicating the minimum and maximum number of instances of the adjacent rectangle there are for one instance of the rectangle at the other end of the relationship line. The placement of these numbers is often confusing. The first symbol is either 0 to indicate that it is possible for no instances of the entity joining the connector to be related to a given instance of the entity on the other side of the relationship, 1 if at least one instance is necessary or it is omitted if more than one instance is required. For example, more than one student must be enrolled in a course for it to run, but it is possible for no students to have a particular instructor (if they are on leave).

The second symbol gives the maximum number of instances of the entity joining the connector for each instance of the entity on the other side of the relationship. If there is only one such instance, this symbol is 1. If more than 1, the symbol is a crows foot opening towards the rectangle.

If you read it like a sentence, the first entity is the subject, the relationship is the verb, the cardinality after the relationship tells how many direct objects (second entity) there are.

I.e. A student is enrolled in one or morecourses subject verb objects.

Step 5. Define Primary Keys

For each entity we must find a unique primary key so that instances of that entity can be distinguished from one another. Often a single field or property is a primary key (e.g. a Student ID). Other times the identifier is a set of fields or attributes (e.g. a course needs a department identifier, a course number, and often a section number; a Room needs a Building Name and a Room Number). When the entity is written with all its attributes, the primary key is underlined.

Step 6. Draw Key-Based ERD

Looking at the Rough Draft ERD, we may see some relationships which are non-specific or many-to-many. I.e., there are crows feet on both ends of the relationship line. Such relationships spell trouble later when we try to implement the related entities as data stores or data files, since each record will need an indefinite number of fields to maintain the many-to-many relationship.

Fortunately, by introducing an extra entity, called an associative entity for each many-to-many relationship, we can solve this problem. The new associative entity's name will be the hyphenation of the names of the two originating entities. It will have a concatenated key consisting of the keys of these two entities. It will have a 1-1 relationship with each of its parent entities and each parent will have the same relationship with the associative entity that they had with each other before we introduced the associative entity. The original relationship between the parents will be deleted from the diagram.

The key-based ERD has no many-to-many relationships and each entity has its primary and foreign keys listed below the entity name in its rectangle.

Step 7. Identify Attributes

A data attribute is a characteristic common to all or most instances of a particular entity. In this step we try to identify and name all the attributes essential to the system we are studying without trying to match them to particular entities. The best way to do this is to study the forms, files and reports currently kept by the users of the system and circle each data item on the paper copy.

Cross out those which will not be transferred to the new system, extraneous items such as signatures, and constant information which is the same for all instances of the form (e.g. your company name and address). The remaining circled items should represent the attributes you need. You should always verify these with your system users. (Sometimes forms or reports are out of date.)

Step 8. Map Attributes

For each attribute we need to match it with exactly one entity. Often it seems like an attribute should go with more than one entity (e.g. Name). In this case you need to add a modifier to the

attribute name to make it unique (e.g. Customer Name, Employee Name, etc.) or determine which entity an attribute "best' describes. If you have attributes left over without corresponding entities, you may have missed an entity and its corresponding relationships. Identify these missed entities and add them to the relationship matrix now.

Step 9. Draw Fully-Attributed ERD

If you introduced new entities and attributes in step 8, you need to redraw the entity relationship diagram. When you do so, try to rearrange it so no lines cross by putting the entities with the most relationships in the middle. If you use a tool like Systems Architect, redrawing the diagram is relatively easy.

Even if you have no new entities to add to the Key-Based ERD, you still need to add the attributes to the Non-Key Data section of each rectangle. Adding these attributes automatically puts them in the repository, so when we use the entity to design the new system, all its attributes will be available.

Step 10. Check Results

Look at your diagram from the point of view of a system owner or user. Is everything clear? Check through the Cardinality pairs. Also, look over the list of attributes associated with each entity to see if anything has been omitted.

Conclusion: The entity relationship diagram was made successfully by following the steps described above.

EX. NO: 4 Date:	DATA FLOW DIAGRAM

AIM: To prepare DATA FLOW DIAGRAM for any project.

REQUIREMENTS:

Hardware Interfaces

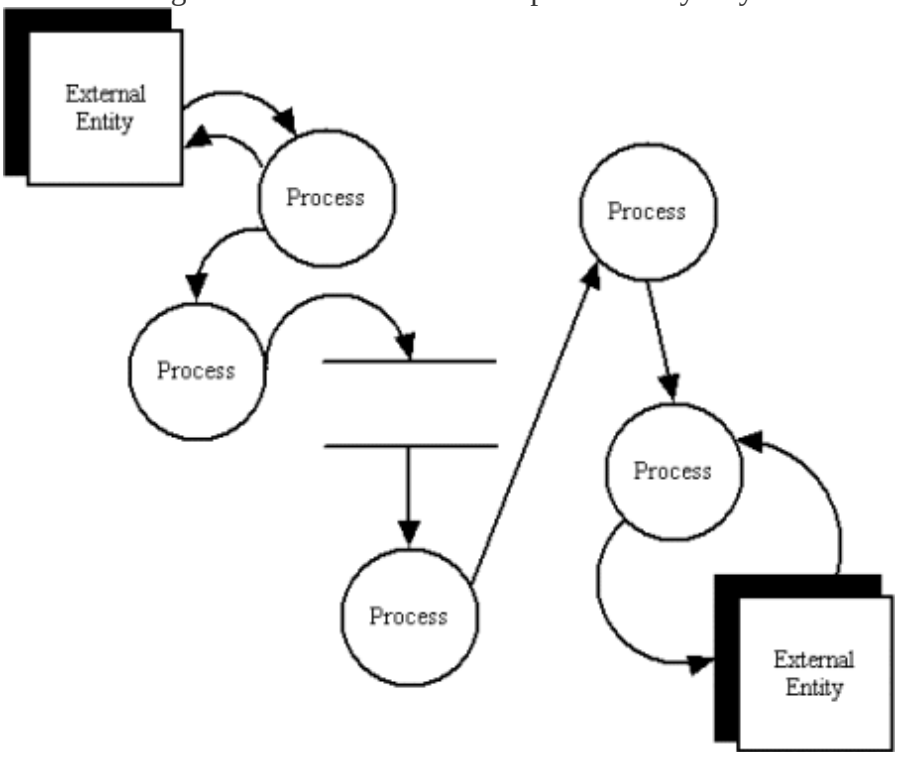
- Pentium(R) 4 CPU 2.26 GHz, 128 MB RAM
- Screen resolution of at least 800 x 600 required for proper and complete viewing of screens. Higher resolution would not be a problem.
- CD ROM Driver

Software Interfaces

- Any window-based operating system (Windows 95/98/2000/XP/NT)
- WordPad or Microsoft Word

THEORY

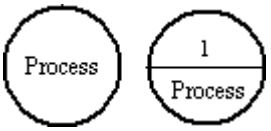
Data flow diagrams illustrate how data is processed by a system in terms of inputs and outputs.



Data Flow Diagram Notations

You can use two different types of notations on your data flow diagrams: Yourdon & Coad or Gane & Sarson.

Process Notations



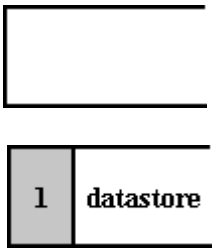
Yourdon and Coad Process Notations

Process

A process transforms incoming data flow into outgoing data flow.

Datastore Notations

Yourdon and Coad Datastore Notations

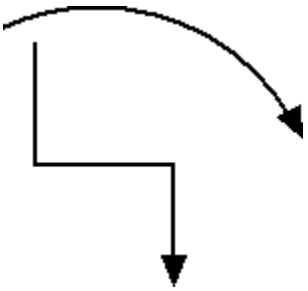


Gane and Sarson Datastore Notations

Data Store

Datastores are repositories of data in the system. They are sometimes also referred to as files.

Dataflow Notations



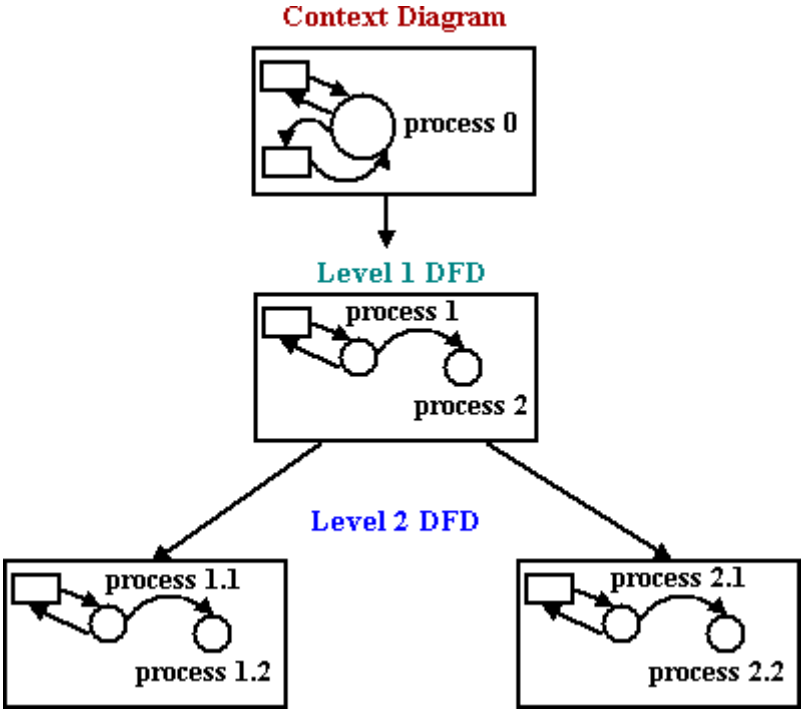
Dataflow

Dataflows are pipelines through which packets of information flow. Label the arrows with the name of the data that moves through it.

HOW TO DRAW DATA FLOW DIAGRAMS (cont'd)

Data Flow Diagram Layers

Draw data flow diagrams in several nested layers. A single process node on a high level diagram can be expanded to show a more detailed data flow diagram. Draw the context diagram first, followed by various layers of data flow diagrams.

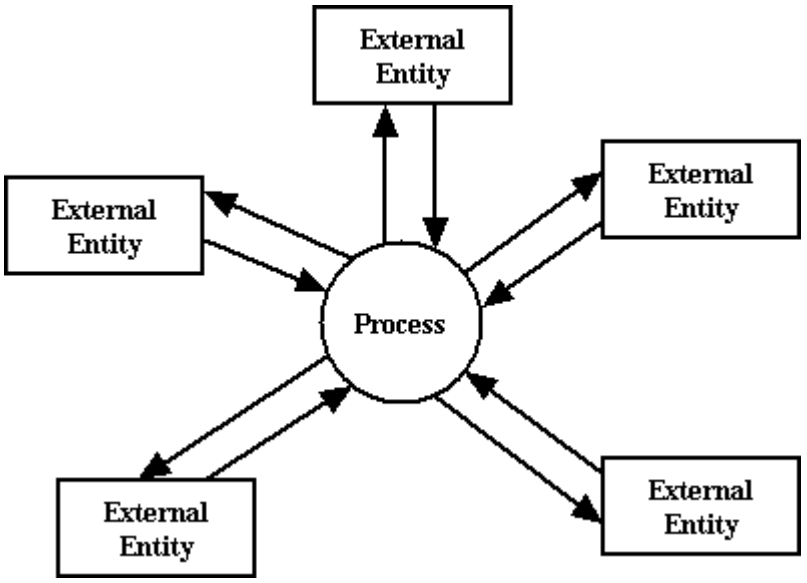


The nesting of data flow layers

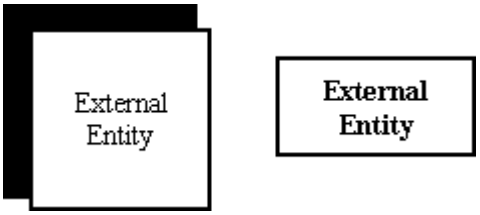
Context Diagrams

A context diagram is a top level (also known as Level 0) data flow diagram. It only contains

one process node (process 0) that generalizes the function of the entire system in relationship to external entities.



External Entity Notations

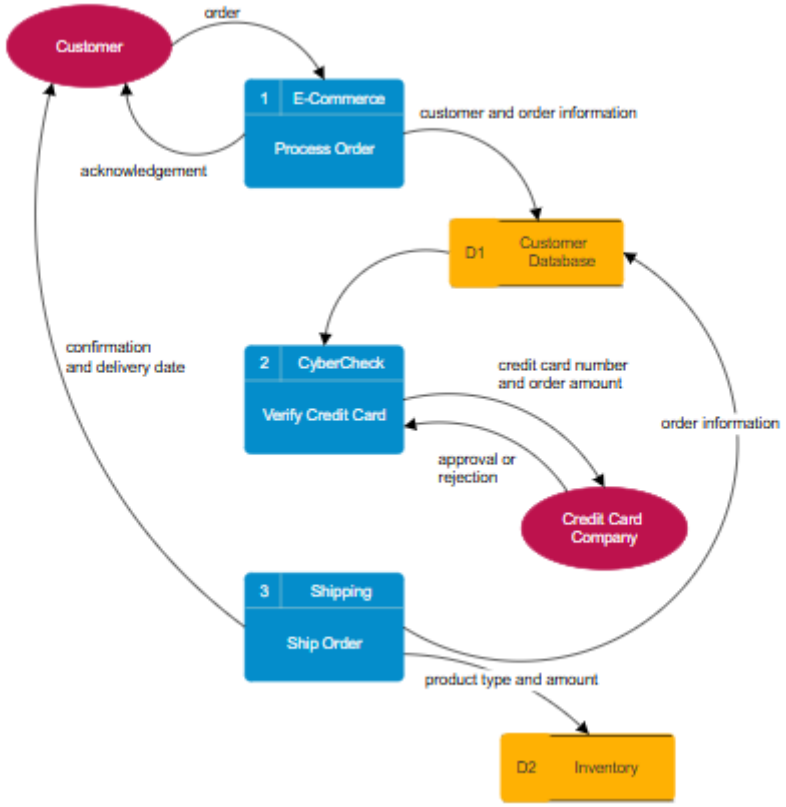


External Entity

External entities are objects outside the system, with which the system communicates. External entities are sources and destinations of the system's inputs and outputs.

DFD levels

The first level DFD shows the main processes within the system. Each of these processes can be broken into further processes until you reach pseudocode.



Data flow diagram

Conclusion: The dataflow diagram was made successfully by following the steps described above.

EX. NO: 5 Date:	USE CASE DIAGRAM

Aim:

To prepare the Use Case Diagram for Automated food ordering System

Hardware Requirements:

Pentium 4 processor (2.4 GHz), 128 Mb RAM, Standard keyboard n mouse, colored monitor.

Software Requirements:

Rational Rose, Windows XP,

Theory:

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:

1. Use Cases

A use case describes a sequence of actions that provide a measurable value to an actor. A use 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:

- An association between an actor and a use case
- An association between two use cases
- A generalization between two actors
- 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 Like‖ Rule to Use Case Generalization
12. Place an Inheriting Use Case Below The Base Use Case
13. Apply the ―Is Like‖ Rule to Actor Inheritance
14. Place an Inheriting Actor Below the Parent Actor

4. System Boundary Boxes

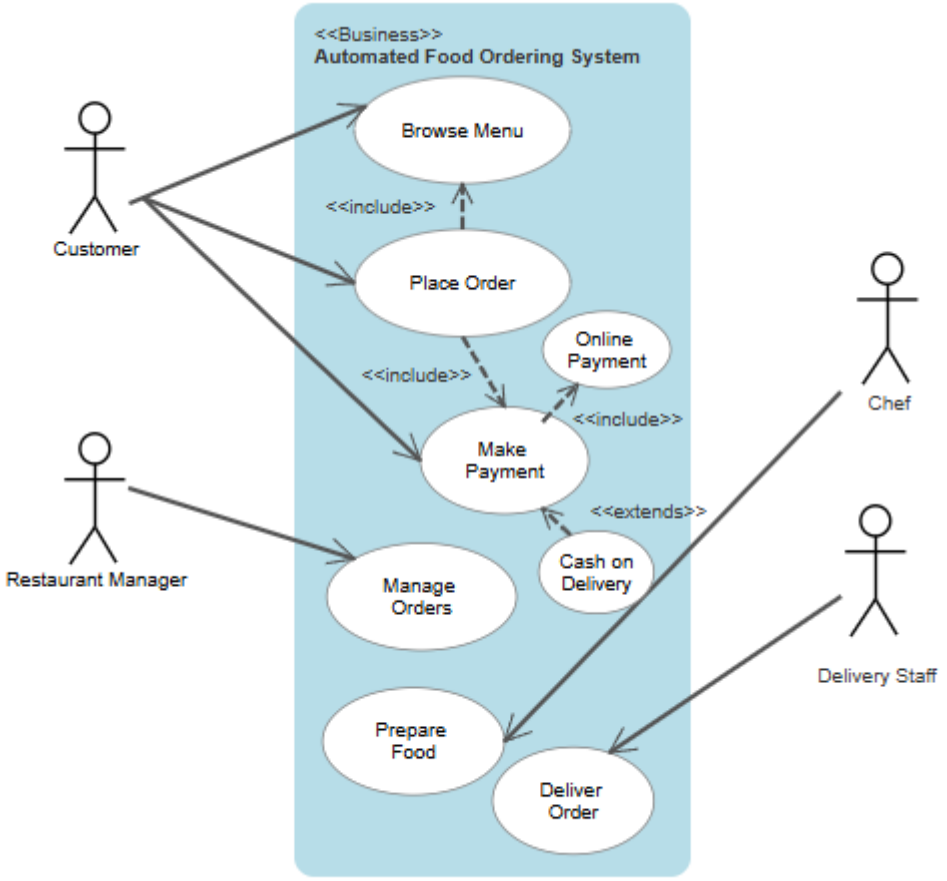
The rectangle around the use cases is called the system boundary box and as the name suggests it indicates the scope of your system – the use cases inside the rectangle represent the functionality that you intend to implement.

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.

Use Case Diagram for Automated Food Ordering System



Conclusion: The use case diagram was made successfully by following the steps described above.

EX. NO: 6a Date:	ACTIVITY DIAGRAM

AIM :-

To draw a sample activity diagram for real project or system.

Hardware Requirements:

Pentium 4 processor (2.4 GHz), 128 Mb RAM, Standard keyboard n mouse, colored monitor.

Software Requirements:

Rational Rose, Windows XP,

THEORY

UML 2 activity diagrams are typically used for business process modeling, for modeling the logic captured by a single use case or usage scenario, or for modeling the detailed logic of a business rule. Although UML activity diagrams could potentially model the internal logic of a complex operation it would be far better to simply rewrite the operation so that it is simple enough that you don‘t require an activity diagram. In many ways UML activity diagrams are the object-oriented equivalent of flow charts and data flow diagrams (DFDs) from structured development.

Let‘s start by describing the basic notation :

- **Initial node**. The filled in circle is the starting point of the diagram. An initial node isn‘t required although it does make it significantly easier to read the diagram.
- **Activity final node**. The filled circle with a border is the ending point. An activity diagram can have zero or more activity final nodes.
- **Activity**. The rounded rectangles represent activities that occur. An activity may be physical, such as Inspect Forms, or electronic, such as Display Create Student Screen.
- **Flow/edge**. The arrows on the diagram. Although there is a subtle difference between flows and edges,never a practical purpose for the difference although.
- **Fork**. A black bar with one flow going into it and several leaving it. This denotes the beginning of parallel activity.
- **Join**. A black bar with several flows entering it and one leaving it. All flows going into the join must reach it before processing may continue. This denotes the end of parallel processing.
- **Condition**. Text such as [Incorrect Form] on a flow, defining a guard which must evaluate to true in order to traverse the node.
- **Decision**. A diamond with one flow entering and several leaving. The flows leaving include conditions although some modelers will not indicate the conditions if it is obvious.
- **Merge**. A diamond with several flows entering and one leaving. The implication is that one or more incoming flows must reach this point until processing continues,

based on any guards on the outgoing flow.

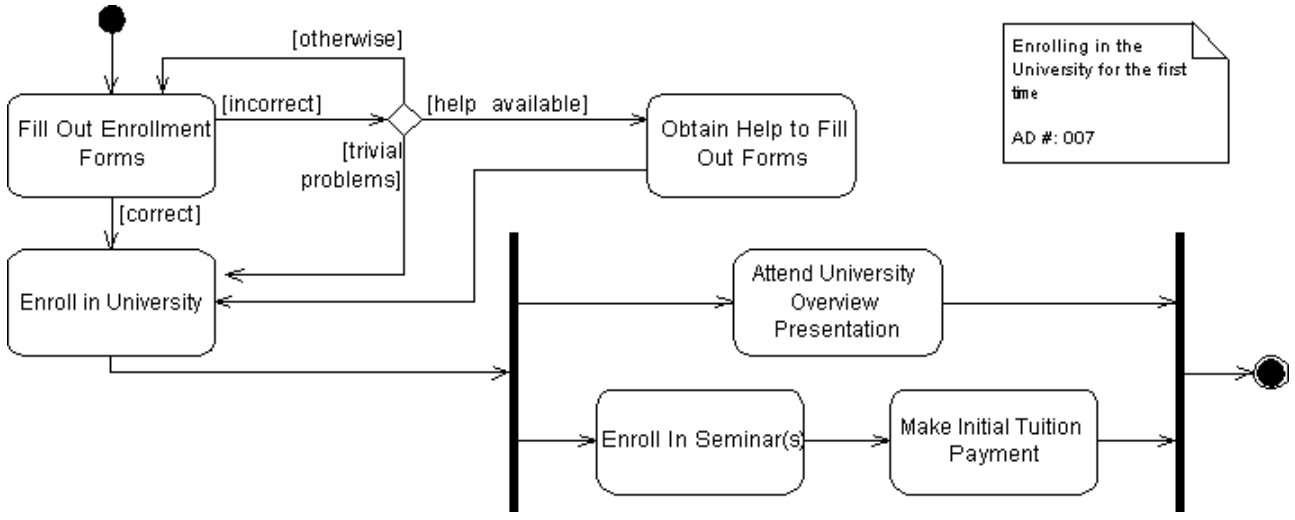
- **Partition**. If figure is organized into three partitions, it is also called swimlanes, indicating who/what is performing the activities (either the Applicant, Registrar, or System).
- **Sub-activity indicator**. The rake in the bottom corner of an activity, such as in the Apply to University activity, indicates that the activity is described by a more finely detailed activity diagram.
- **Flow final**. The circle with the X through it. This indicates that the process stops at this point.

GUIDELINES ASSOCIATED FOR DRAWING AN ACTIVITY DIAGRAM

1.General Guidelines
2.Activities
3.Decision Points
4.Guards
5.Parallel Activities
6.Swimlane Guidelines
7.Action-Object Guidelines

1. General Guidelines

Figure1. Modeling a business process with a UML Activity Diagram.



1. Place The Start Point In The Top-Left Corner. A start point is modeled with a filled in circle, using the same notation that UML State Chart diagrams use. Every UML Activity Diagram should have a starting point, and placing it in the top-left corner reflects the way that people in Western cultures begin reading. Figure1, which models the business process of enrolling in a university, takes this approach.
2. Always Include an Ending Point. An ending point is modeled with a filled in circle with a border around it, using the same notation that UML State Chart diagrams use. Figure1 is interesting because it does not include an end point because it describes a continuous process – sometimes the guidelines don‘t apply.
3. Flowcharting Operations Implies the Need to Simplify. A good rule of thumb is that if an operation is so complex you need to develop a UML Activity diagram to understand it that you should consider refactoring it.

2. Activities

An activity, also known as an activity state, on a UML Activity diagram typically represents the invocation of an operation, a step in a business process, or an entire business process.

1. Question ―Black Hole‖ Activities. A black hole activity is one that has transitions into it but none out, typically indicating that you have either missed one or more transitions.
2. Question ―Miracle‖ Activities. A miracle activity is one that has transitions out of it but none into it, something that should be true only of start points.

3. Decision Points

A decision point is modeled as a diamond on a UML Activity diagram.

1. Decision Points Should Reflect the Previous Activity. In figure1 we see that there is no label on the decision point, unlike traditional flowcharts which would include text describing the actual decision being made, we need to imply that the decision concerns whether the person was enrolled in the university based on the activity that the decision point follows. The guards, depicted using the format [description], on the transitions leaving the decision point also help to describe the decision point.
2. Avoid Superfluous Decision Points. The Fill Out Enrollment Forms activity in FIGURE1 includes an implied decision point, a check to see that the forms are filled out properly, which simplified the diagram by avoiding an additional diamond.

4. Guards

A guard is a condition that must be true in order to traverse a transition.

1. Each Transition Leaving a Decision Point Must Have a Guard
2. Guards Should Not Overlap. For example guards such as x <0, x = 0, and x > 0 are consistent whereas guard such as x <= 0 and x >= 0 are not consistent because they overlap – it isn‘t clear what should happen when x is 0.
3. Guards on Decision Points Must Form a Complete Set. For example, guards such as x < 0 and x >0 are not complete because it isn‘t clear what happens when x is 0.

4. Exit Transition Guards and Activity Invariants Must Form a Complete Set. An activity invariant is a condition that is always true when your system is processing an activity.
5. Apply a [Otherwise] Guard for ―Fall Through‖ Logic.
6. Guards Are Optional. It is very common for a transition to not include a guard, even when an activity includes several exit transitions.

5. Parallel Activities

It is possible to show that activities can occur in parallel, as you see in FIGURE 1 depicted using two parallel bars. The first bar is called a fork, it has one transition entering it and two or more transitions leaving it. The second bar is a join, with two or more transitions entering it and only one leaving it.

1. A Fork Should Have a Corresponding Join. In general, for every start (fork) there is an end (join). In UML 2 it is not required to have a join, but it usually makes sense.
2. Forks Have One Entry Transition.
3. Joins Have One Exit Transition
4. Avoid Superfluous Forks. FIGURE 2 depicts a simplified description of the software process of enterprise architectural modeling, a part of the Enterprise Unified Process (EUP). There is significant opportunity for parallelism in this process, in fact all of these activities could happen in parallel, but forks were not introduced because they would only have cluttered the diagram.

6. Swimlane Guidelines

A swimlane is a way to group activities performed by the same actor on an activity diagram or to group activities in a single thread. FIGURE 2 includes three swimlanes, one for each actor.

Figure2. A UML activity diagram for the enterprise architectural modeling (simplified).

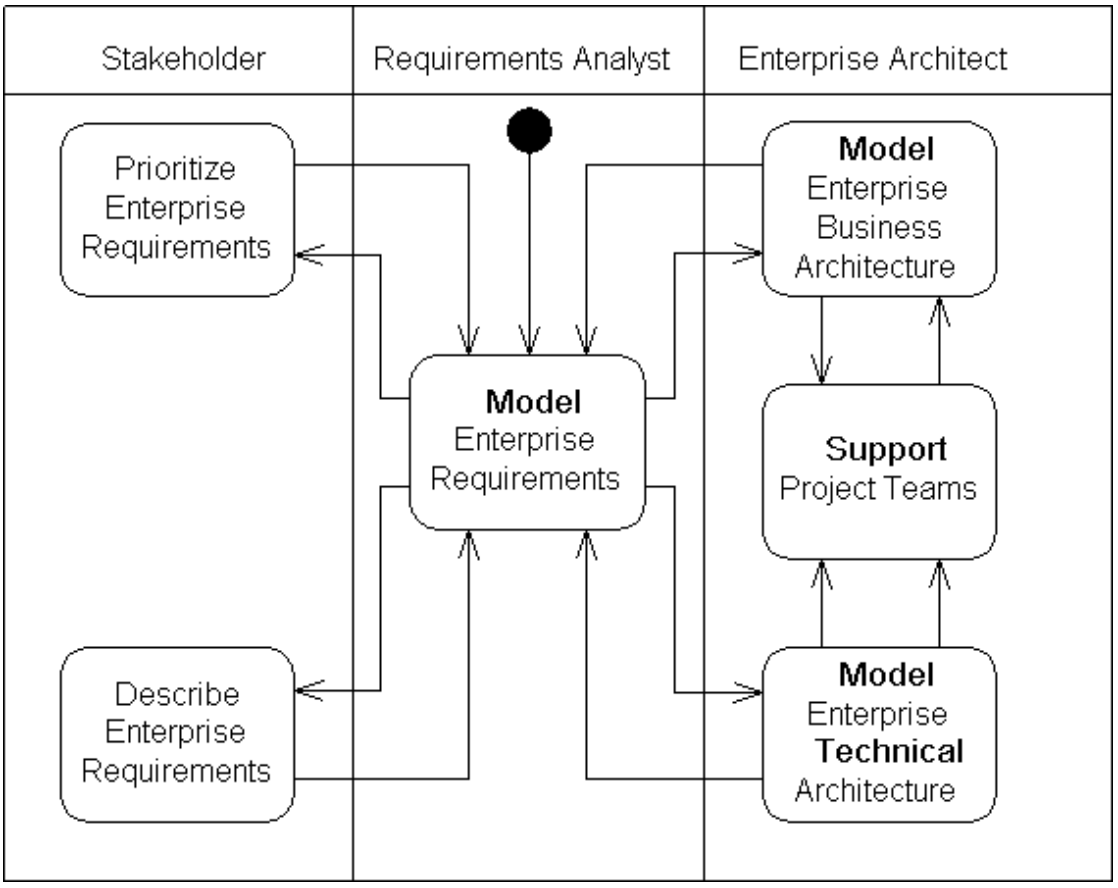
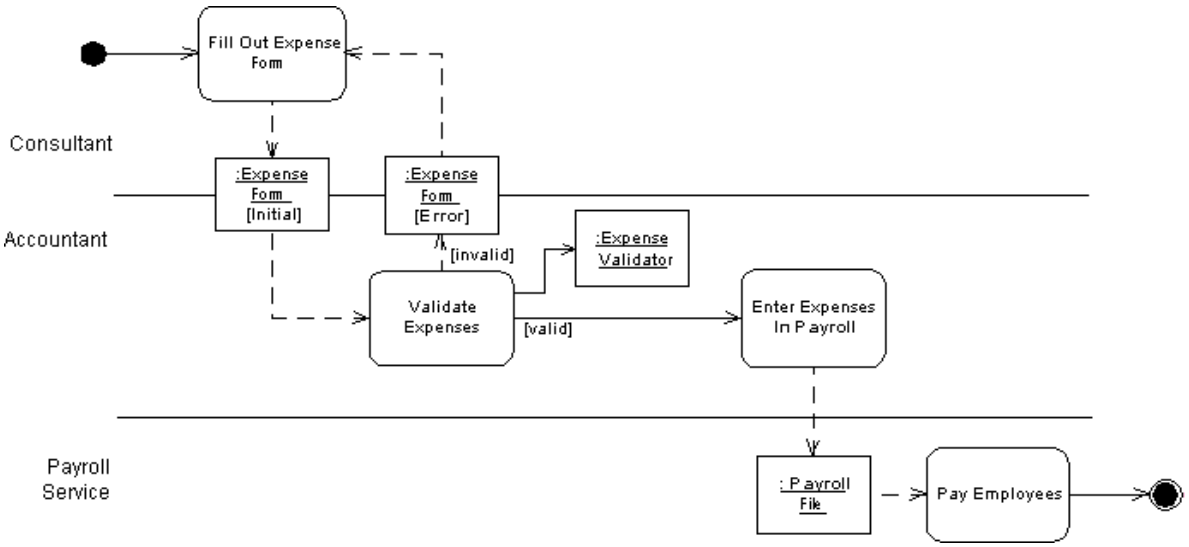


Figure 3. Submitting expenses.



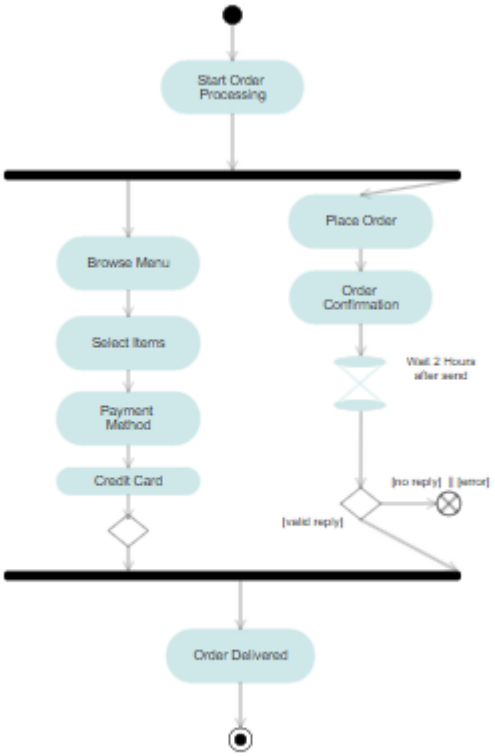
1. Order Swimlanes in a Logical Manner.
2. Apply Swim Lanes To Linear Processes. A good rule of thumb is that swimlanes are best applied to linear processes, unlike the one depicted in FIGURE 3.
3. Have Less Than Five Swimlanes.
4. Consider Swimareas For Complex Diagrams.
5. Swimareas Suggest The Need to Reorganize Into Smaller Activity Diagrams.
6. Consider Horizontal Swimlanes for Business Processes. In FIGURE 3 you see that the swimlanes are drawn horizontally, going against common convention of drawing them vertically.

7 Action-Object Guidelines

Activities act on objects, In the strict object-oriented sense of the term an action object is a system object, a software construct. In the looser, and much more useful for business application modeling, sense of the term an action object is any sort of item. For example in FIGURE 3 the ExpenseForm action object is likely a paper form.

1. Place Shared Action Objects on Swimlane Separators
2. When An Object Appears Several Time Apply State Names
3. State Names Should Reflect the Lifecycle Stage of an Action Object
4. Show Only Critical Inputs and Outputs
5. Depict Action Objects As Smaller Than Activities

ACTIVITY DIAGRAMS:



Conclusion: The activity diagram was made successfully by following the steps described above.

EX. NO: 6b Date:	STATE CHART DIAGRAM

AIM: To prepare STATE CHART DIAGRAM for any project.

REQUIREMENTS:

Hardware Interfaces

- Pentium(R) 4 CPU 2.26 GHz, 128 MB RAM
- Screen resolution of at least 800 x 600 required for proper and complete viewing of screens. Higher resolution would not be a problem.
- CD ROM Driver

Software Interfaces

- Any window-based operating system(Windows98/2000/XP/NT)
- IBM Rational Rose Software

THEORY:

✓ State Chart Diagrams provide a way to model the various states in which an object can exist.

✓ There are two special states: the start state and the stop state.

The **Start state** is represented by a **block dot**.

The **Stop state** is represented by a **bull's eye**.

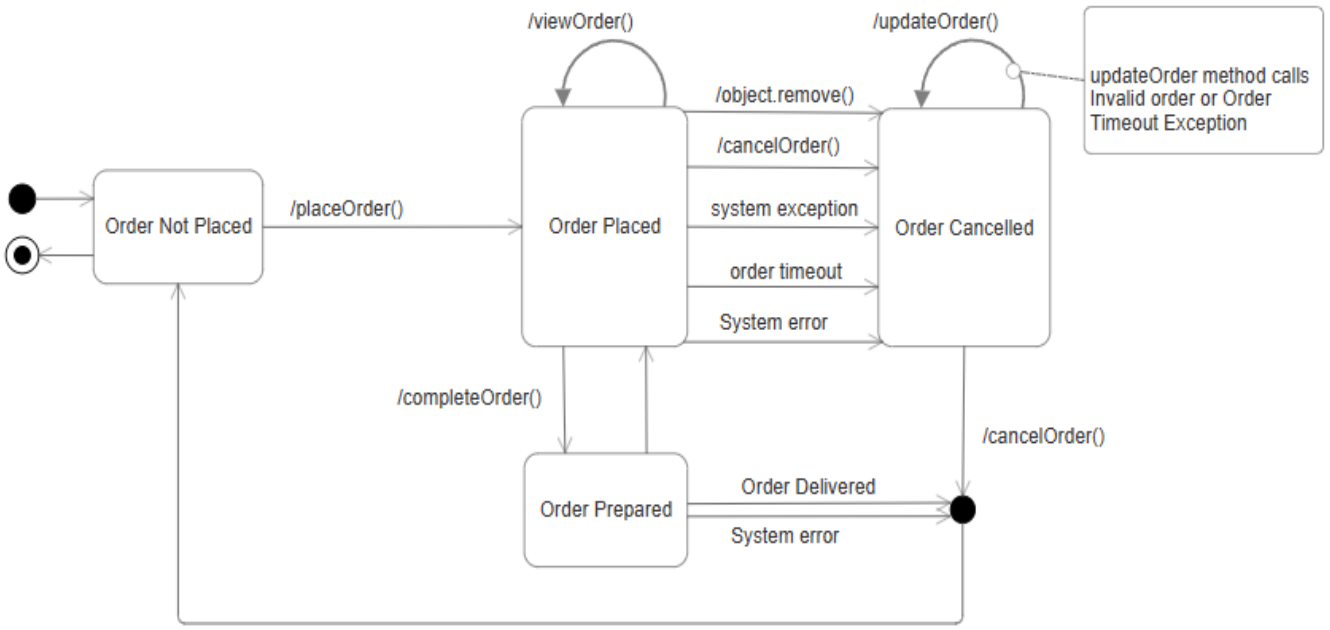
✓ A condition enclosed in square brackets is called a **guard condition,** and controls when a transition can or cannot occur.

✓ **Process** that occur while an object is in certain state are called **actions.**

STEPS TO DRAW STATE CHART DIAGRAM IN RATIONAL ROSE SOFTWARE

- To insert new state diagram secondary click on Logical View.
  - **Select---New State chart Diagram**.
  - A new diagram will be created, type in a name for the new diagram.
- Now double click on the new diagram to open it on the stage.
  - To begin the diagram click on the **"START STATE"** button.
  - Place a start state icon on the diagram by clicking the mouse once.
- Now add states to the diagram, these make up the content of the diagram. Click on the state button. Place the instances for each state into the diagram and type in names for them.
- Now arrange the states to fill the diagram better. Drag the states to new positions to make the easiest layout to work with.Add an end state to the diagram by clicking the **"END STATE"** button. Place an instance into the diagram. Now add relationships to the diagram.
- Click on the **"STATE TRANSITION"** button and drag arrows between the appropriate states.
- To edit the specification secondary click on the relation lines and select **"OPEN SPECIFICATION"** button. Add a name for the event in the specification. Then click on
  ―apply‖ and then on ―OK‖ button.
- Add details to the specifications of the other relationships in the same way.
- There may be instances on the diagram where a state can join more than one state. In this case add a relationship in the same way. Then enter the specification for the new state.

This is how a state chart diagram is made.



A STATE CHART DIAGRAM FOR AUTOMATED FOOD ORDERING SYSTEM

Conclusion: The state chart diagram was made successfully by following the steps described above.

EX. NO: 7 Date:	Identifying Domain Classes

Aim:

Steps to identifying domain classes from the problem statements.

Hardware Requirements:

Pentium 4 processor (2.4 GHz),
128 Mb RAM,
Standard keyboard and mouse,
colored monitor.

Software Requirements:

Microsoft Word, Windows 2010

Theory:

In Object Oriented paradigm Domain Object Model has become subject of interest for its excellent problem comprehending capabilities towards the goal of designing a good software system. Domain Model, as a conceptual model gives proper understanding of problem description through its highly effective component – the Domain Classes. Domain classes are the abstraction of key entities, concepts or ideas presented in the problem statement [iv]. As stated in [v], domain classes are used for representing business activities during the analysis phase.

Steps to Identify Domain Classes from Problem Statement

We now present the steps to identify domain classes from a given problem statement. This approach is mostly based on the "Grammatical approach using nouns" discussed above, with some insights.

1. Make a list of potential objects by finding out the nouns and noun phrases from narrative problem statement
2. Apply subject matter expertise (or domain knowledge) to identify additional classes
3. Filter out the redundant or irrelevant classes
4. Classify all potential objects based on categories. We follow the category table as described by Ross (table 5-3, pg 88, [1])
5. Group the objects based on similar attributes. While grouping we should remember that
    a. Different nouns (or noun phrases) can actually refer to the same thing (examples: house, home, abode)
    b. Same nouns (or noun phrases) could refer to different things or concepts (example: I go to school every day / This school of thought agrees with the theory)
6. Give related names to each group to generate the final list of top level classes
7. Iterate over to refine the list of classes.

Categories	Explanation
People	Humans who carry out some function
Places	Areas set aside for people or things
Things	Physical objects
Organizations	Collection of people, resources, facilities and capabilities having a defined mission
Concepts	Principles or Ideas not tangible
Events	Things that happen (usually at a given date and time), or as a steps in an ordered sequence

Identify the domain classes from the following problem statement

A chain of restaurants has approached you to develop an **Automated Food Ordering System**. The goal is to simplify and streamline the ordering, payment, and delivery process while enhancing customer experience. Below are the requirements:

- Restaurants want customers to browse menus and place orders directly from their phones or a web platform.
- Customers will add items to a cart, confirm orders, and make secure payments online or opt for cash on delivery (COD).
- The system will generate a receipt after payment and assign a delivery agent to fulfill the order.
- Delivery agents will pick up orders from restaurants and deliver them to the customers. In case of unserviceable areas, delivery will be made to the nearest landmark.
- Restaurants will register in the system and upload/update menus.
- The system must handle promotional offers, ratings, and feedback for continuous improvement.
- Customers want a feature for pre-ordering meals for specific dates and times.
- The system should generate reports for restaurants to track sales, popular items, and performance.

Learning Objectives

- **Identifying Potential Classes:** Extract classes and their attributes from the problem statement.
- **Categorization:** Categorize nouns (or noun phrases) to identify places, people, things, events, and concepts.
- **Iterative Modeling:** Refine the classes and attributes to ensure a comprehensive solution.

Limitations: The proposed solution provides a basic framework and focuses on the initial identification of classes, leaving room for refinement to handle edge cases and scalability. It is confined to the problem statement, excluding real-world complexities like multi-branch management, advanced features (e.g., loyalty programs, real-time tracking), and integration with external systems.

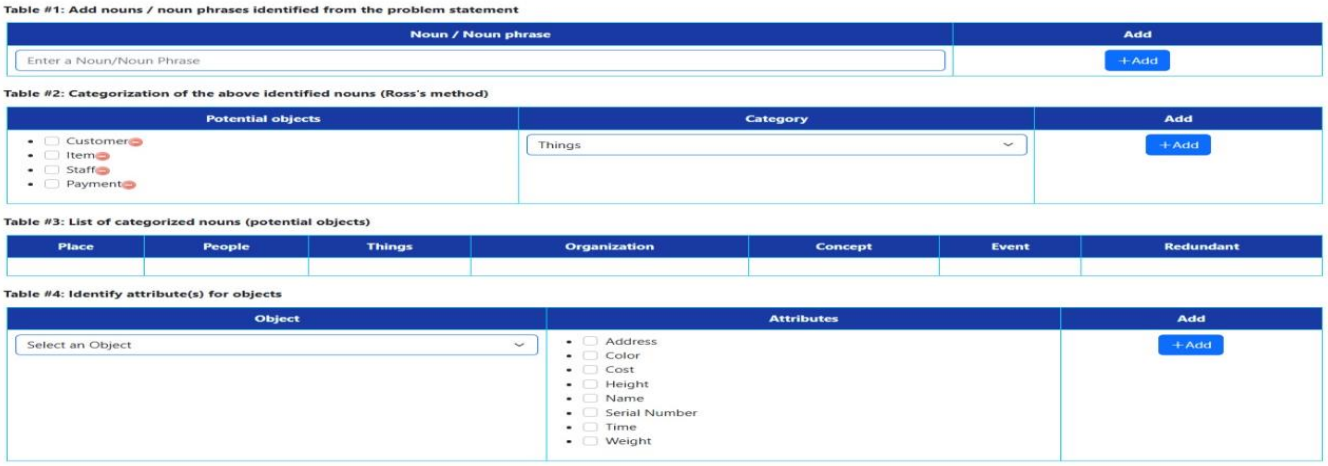


Table #6: List of objects and it's attribute(s)

Object	Attribute
Customer	• Address • Name • Serial Number
Item	• Cost • Name • Serial Number
Staff	• Name • Serial Number
Payment	• Cost • Serial Number

Table #7: Define classes from list of attribute(s)

Attributes	Class	Add
• ☐ Address • ☐ Color • ☐ Cost • ☐ Height • ☐ Name • ☐ Serial Number • ☐ Time • ☐ Weight	Enter a Class	+Add

Table #8: List of classes and their attribute(s)

Classes	Attribute
Customer	• Address • Name • Serial Number
Item	• Cost • Name • Serial Number
Staff	• Name • Serial Number
Payment	• Cost • Serial Number

Conclusion :

The identification of domain classes was made successfully by following the above steps

EX. NO: 8a Date:	CLASS DIAGRAM

AIM:

To draw class diagram for Automated Food Ordering System.

REQUIREMENTS:

Hardware Interfaces

- Pentium(R) 4 CPU 2.26 GHz, 128 MB RAM
- Screen resolution of at least 800 x 600 required for proper and complete viewing of screens. Higher resolution would not be a problem.
- CD ROM Driver

Software Interfaces

- Any window-based operating system(Windows98/2000/XP/NT)
- IBM Rational Rose Software

THEORY:

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

GENERAL GUIDELINES

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. **I**nterface 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.

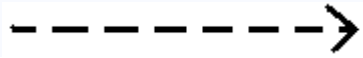
Aggregation occurs when a class is a collection or container of other classes, but where the contained classes do not have a strong **life cycle dependency** on the container--essentially, if the container is destroyed, its contents are not.

ASSOCIATION

Association are semantic connection between classes. When an association connects two classes , each class can send messages to other 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 .



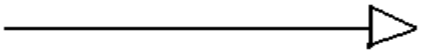
GENERALIZATION

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.

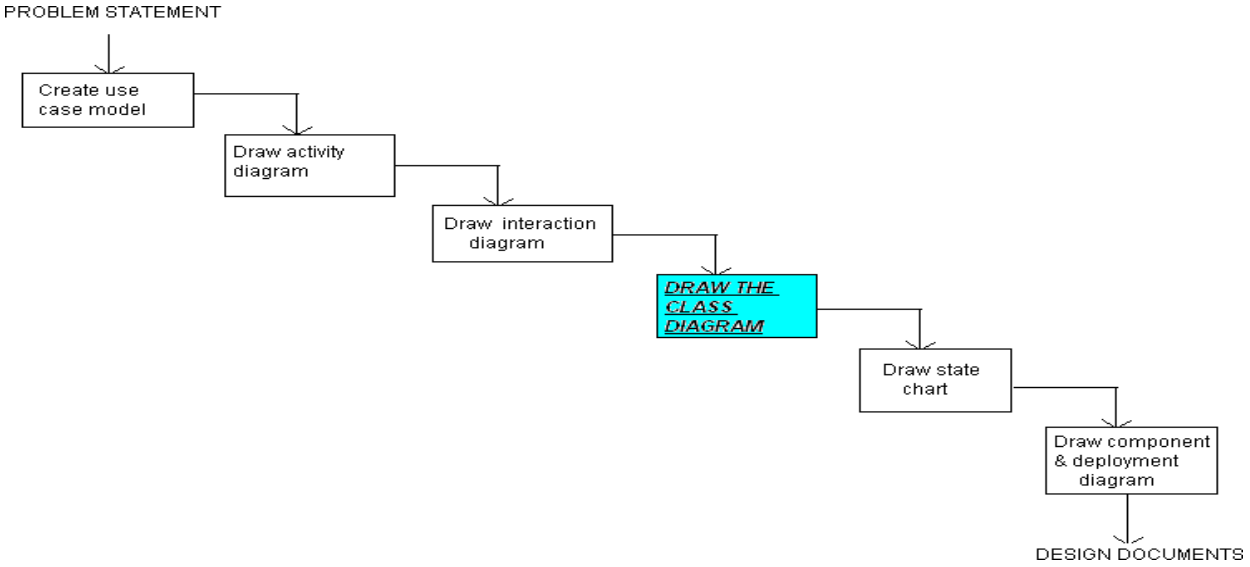


MULTIPLICITY

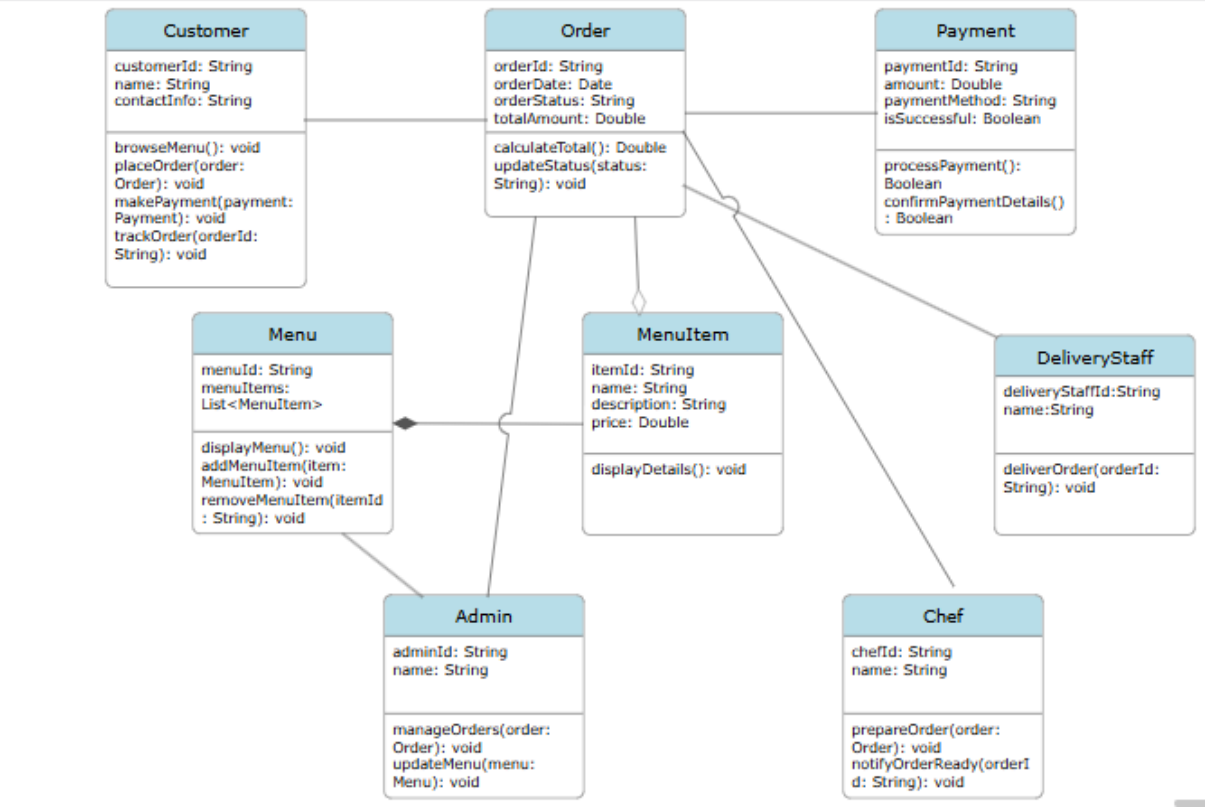
The association relationship indicates that (at least) one of the two related classes makes reference to the other.

HOW TO DRAW CLASS DIAGRAM

When designing classes, the attributes and operations it will have are observed. Then determining how instances of the classes will interact with each other. These are the very first steps of many in developing a class diagram. However, using just these basic techniques one can develop a complete view of the software system. There are various steps in the analysis and design of an object-oriented system.



STEPS FOR ANALYSIS AND DESIGN



Conclusion :

The class diagram was made successfully by following the above steps

EX. NO: 8b Date:	SEQUENCE DIAGRAM

Aim:

To create the Sequence Diagram for Automated Food Ordering System

Hardware Requirements:

Pentium 4 processor (2.4 GHz),
128 Mb RAM,
Standard keyboard and mouse,
colored monitor.

Software Requirements:

Star UML, Windows XP

Theory:

UML sequence diagrams model the flow of logic within the system in a visual manner, enabling the user both to document and validate the logic, and are commonly used for both analysis and design purposes. Sequence diagrams are the most popular UML artifact for dynamic modeling, which focuses on identifying the behavior within your system. Sequence diagrams, along with class diagrams and physical data models are the most important design-level models for modern application development.

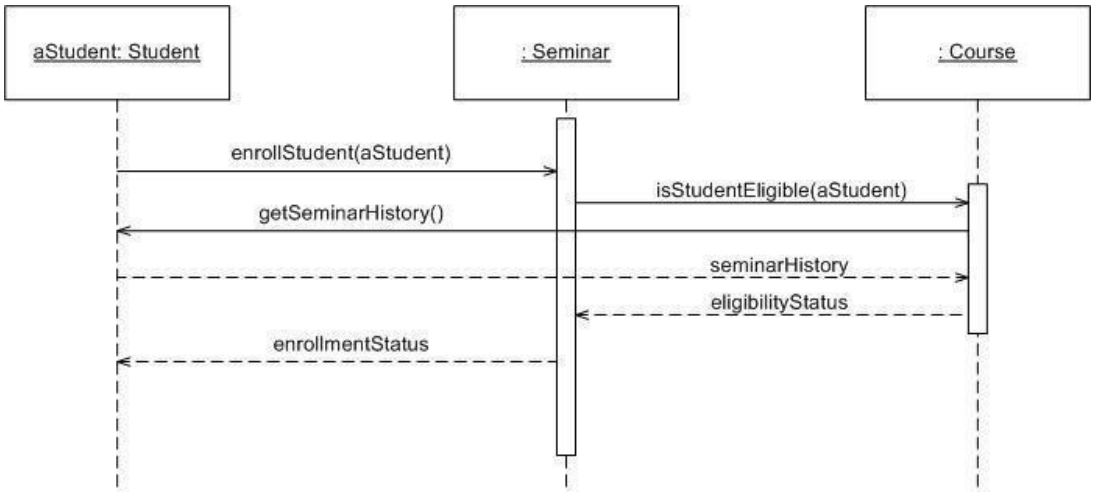
Sequence diagrams are typically used to model:

1. **Usage scenarios**. A usage scenario is a description of a potential way the system is used. The logic of a usage scenario may be part of a use case, perhaps an alternate course. It may also be one entire pass through a use case, such as the logic described by the basic course of action or a portion of the basic course of action, plus one or more alternate scenarios. The logic of a usage scenario may also be a pass through the logic contained in several use cases. For example, a student enrolls in the university, and then immediately enrolls in three seminars.
2. **The logic of methods**. Sequence diagrams can be used to explore the logic of a complex operation, function, or procedure. One way to think of sequence diagrams, particularly highly detailed diagrams, is as visual object code.

3. **The logic of services**. A service is effectively a high-level method, often one that can be invoked by a wide variety of clients. This includes web-services as well as business transactions implemented by a variety of technologies such as CICS/COBOL or CORBA-compliant object request brokers (ORBs).

FIG 3. shows the logic for how to enroll in a seminar. One should often develop a system-level sequence diagram to help both visualize and validate the logic of a usage scenario. It also helps to identify significant methods/services, such as checking to see if the applicant already exists as a student, which the system must support.

Figure 3. Enrolling in a seminar (method).



The dashed lines hanging from the boxes are called object lifelines, representing the life span of the object during the scenario being modeled. The long, thin boxes on the lifelines are activation boxes, also called method-invocation boxes, which indicate processing is being performed by the target object/class to fulfill a message.

How to Draw Sequence Diagrams

Sequence diagramming really is visual coding, even when you are modeling a usage scenario via a system-level sequence diagram.

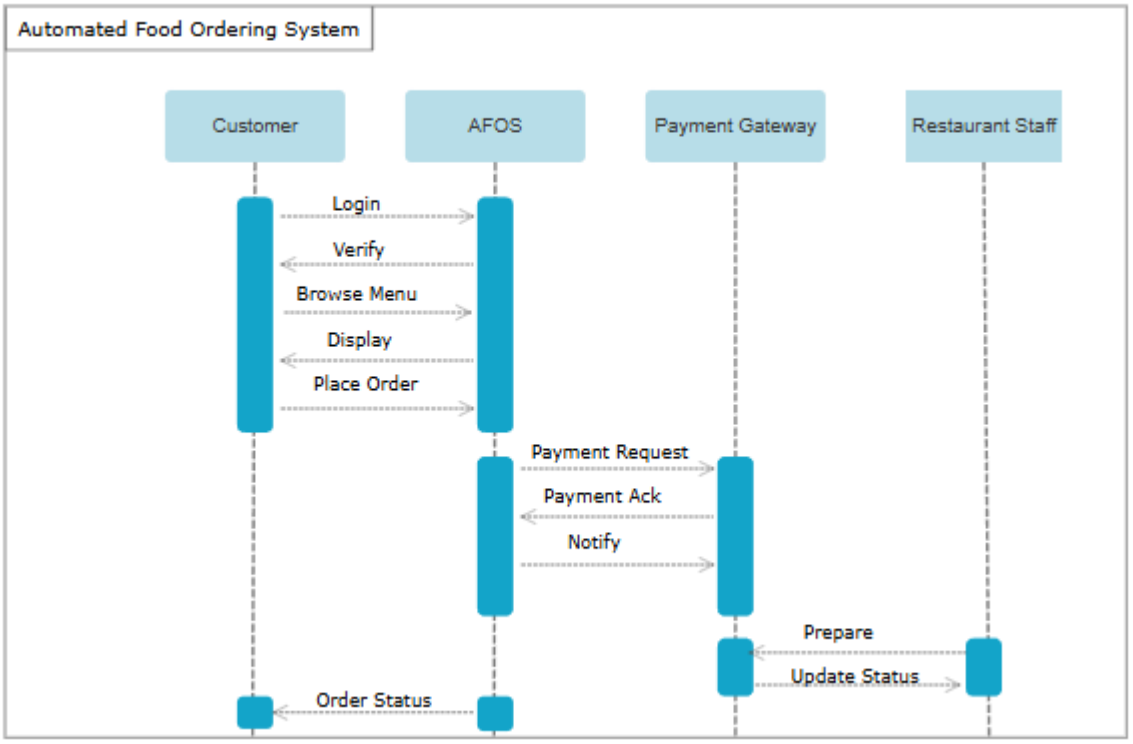
While creating a sequence diagram, start by identifying the scope of what you are trying to model. You should typically tackle small usage scenarios at the system level or a single method/service at the detailed object level.

You should then work through the logic with at least one more person, laying out classifiers across the top as you need them. . The heart of the diagram is in the messages, which you add to the diagram one at a time as you work through the logic. You should rarely indicate return values, instead you should give messages intelligent names which often make it clear what is being returned.

It is interesting to note that as you sequence diagram you will identify new responsibilities for classes and objects, and, sometimes, even new classes. The implication is that you may want

to update your class model appropriately, agile modelers will follow the practice Create Several Models in Parallel, something that CASE tools will do automatically. Remember, each message sent to a class invokes a static method/operation on that class each message sent to an object invokes an operation on that object.

Regarding style issues for sequence diagramming, prefer drawing messages going from left-to- right and return values from right-to-left, although that doesn‘t always work with complex objects/classes. Justify the label on messages and return values, so they are closest to the arrowhead. Also prefer to layer the sequence diagrams: from left-to-right. indicate the actors, then the controller class(es), and then the user interface class(es), and, finally, the business class(es). During design, you probably need to add system and persistence classes, which you should usually put on the right-most side of sequence diagrams. Laying your sequence diagrams in this manner often makes them easier to read and also makes it easier to find layering logic problems, such as user interface classes directly accessing persistence.



Conclusion: The sequence diagram was made successfully by following the steps described above.

EX. NO: 9 Date:	DESIGNING TEST SUITES

AIM:

To design unit test cases to verify the functionality and locate bugs in the module of an application.

Theory:

Testing software is an important part of the development life cycle of a software. It is an expensive activity. Hence, appropriate testing methods are necessary for ensuring the reliability of a program. According to the ANSI/IEEE 1059 standard, the definition of testing is the process of analyzing a software item, to detect the differences between existing and required conditions i.e. defects/errors/bugs and to evaluate the features of the software item.

The purpose of testing is to verify and validate a software and to find the defects present in a software. The purpose of finding those problems is to get them fixed.

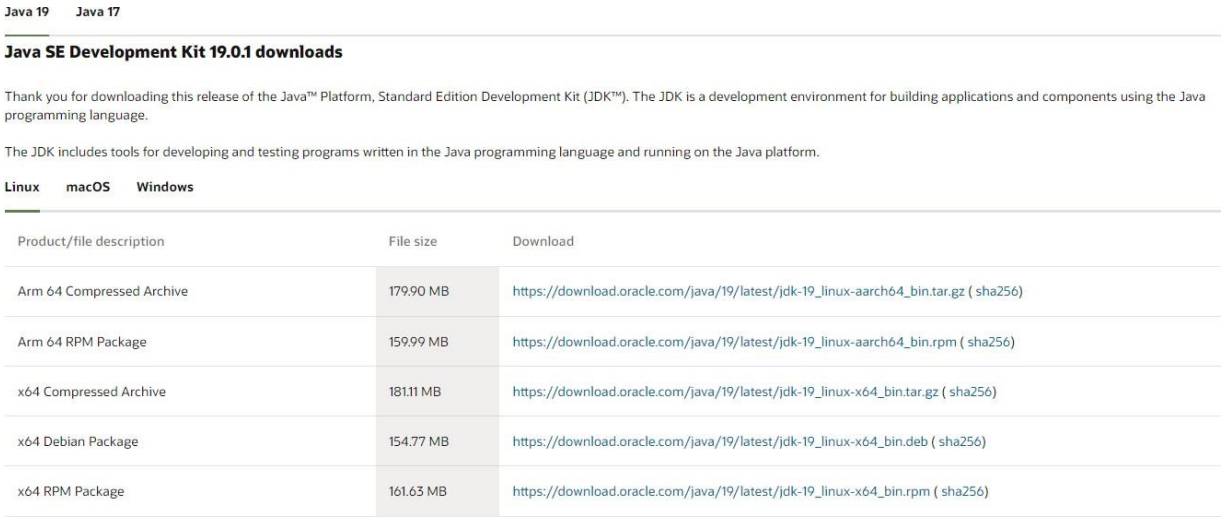
- **Verification** is the checking or we can say the testing of software for consistency and conformance by evaluating the results against pre-specified requirements.
- **Validation** looks at the systems correctness, i.e. the process of checking that what has been specified is what the user actually wanted.
- **Defect** is a variance between the expected and actual result. The defect's ultimate source may be traced to a fault introduced in the specification, design, or development (coding) phases.

Test Cases and Test Suite:

A test case describes an input descriptions and an expected output descriptions. Input are of two types: preconditions (circumstances that hold prior to test case execution) and the actual inputs that are identified by some testing methods. The set of test cases is called a test suite. We may have a test suite of all possible test cases.

Step 1 – Install Java Software Development Kit (JDK)

Download and install the Java Software Development Kit (JDK)



Java 19 Java 17

Java SE Development Kit 19.0.1 downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications and components using the Java programming language.

The JDK includes tools for developing and testing programs written in the Java programming language and running on the Java platform.

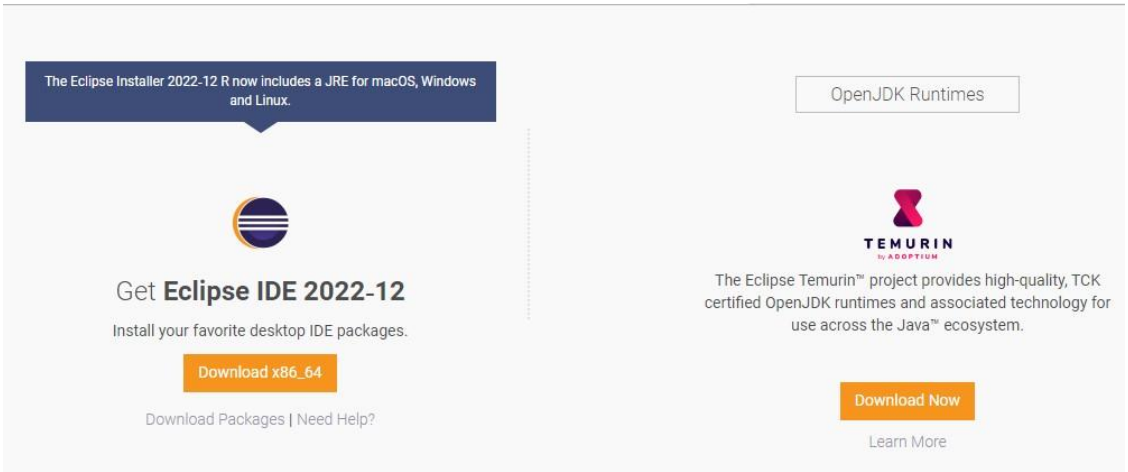
Linux macOS Windows

Product/file description	File size	Download
Arm 64 Compressed Archive	179.90 MB	https://download.oracle.com/java/19/latest/jdk-19_linux-aarch64_bin.tar.gz (sha256)
Arm 64 RPM Package	159.99 MB	https://download.oracle.com/java/19/latest/jdk-19_linux-aarch64_bin.rpm (sha256)
x64 Compressed Archive	181.11 MB	https://download.oracle.com/java/19/latest/jdk-19_linux-x64_bin.tar.gz (sha256)
x64 Debian Package	154.77 MB	https://download.oracle.com/java/19/latest/jdk-19_linux-x64_bin.deb (sha256)
x64 RPM Package	161.63 MB	https://download.oracle.com/java/19/latest/jdk-19_linux-x64_bin.rpm (sha256)

This JDK version comes bundled with Java Runtime Environment (JRE), so you do not need to download and install the JRE separately.

Step 2 – Install Eclipse IDE

Download the latest version of "Eclipse IDE for Java Developers". Be sure to choose correctly between Windows 32 Bit and 64 Bit versions.

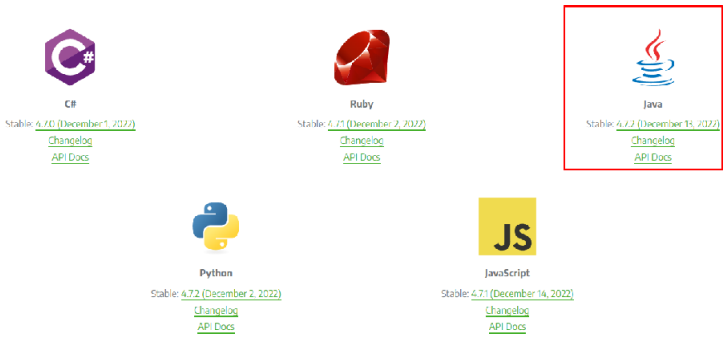


Double-click on a file to Install the Eclipse. A new window will open. Click Eclipse IDE for Java Developers.

After successful completion of the installation procedure, a window will appear. On that window click on Launch.

Step 3 – Selenium WebDriver Installation

You can download Selenium Webdriver for Java Client Driver. You will find client drivers for other languages there, but only choose the one for Java.

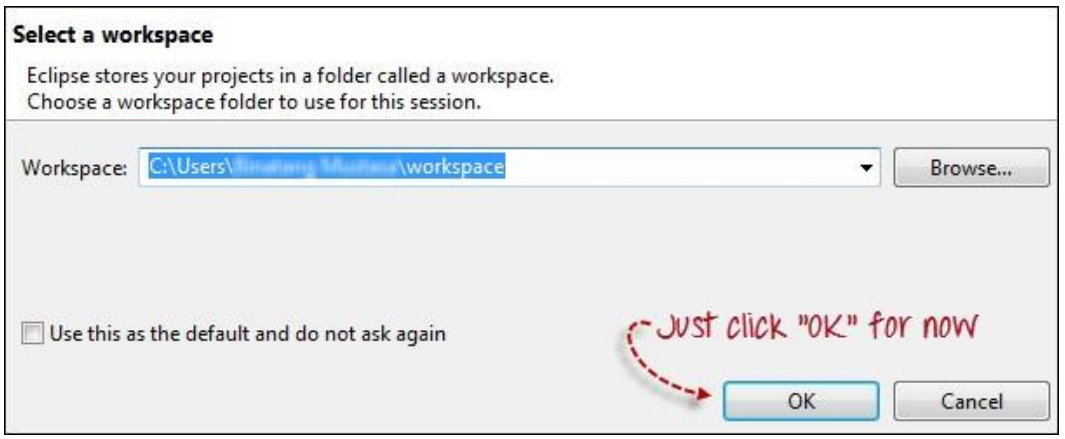


This download comes as a ZIP file named “selenium-3.14.0.zip”. For simplicity of Selenium installation on Windows 10, extract the contents of this ZIP file on your C drive so that you would

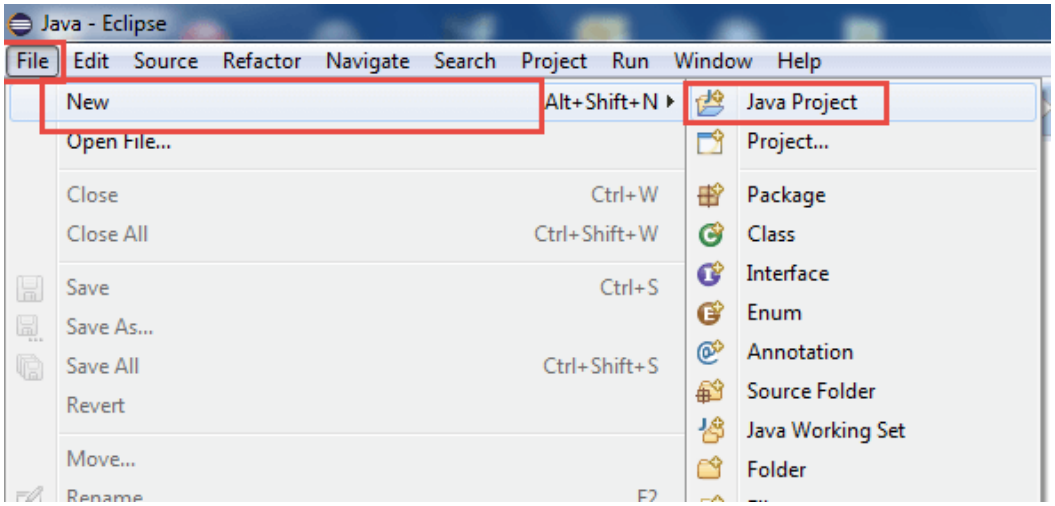
have the directory "C:\selenium-3.14.0\". This directory contains all the JAR files that we would later import on Eclipse for Selenium setup.

Step 4 – Configure Eclipse IDE with WebDriver

1. Launch the "eclipse.exe" file inside the "eclipse" folder that we extracted in step 2. If you followed step 2 correctly, the executable should be located on C:\eclipse\eclipse.exe.
2. When asked to select for a workspace, just accept the default location.

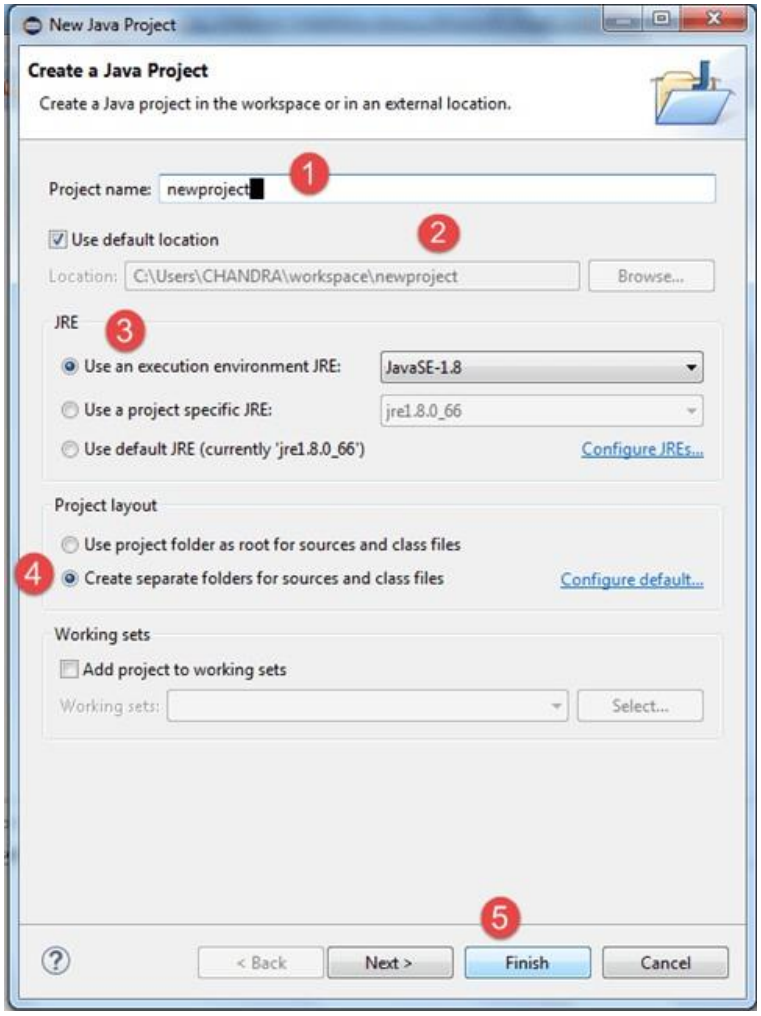


3. Create a new project through File > New > Java Project. Name the project as "newproject".



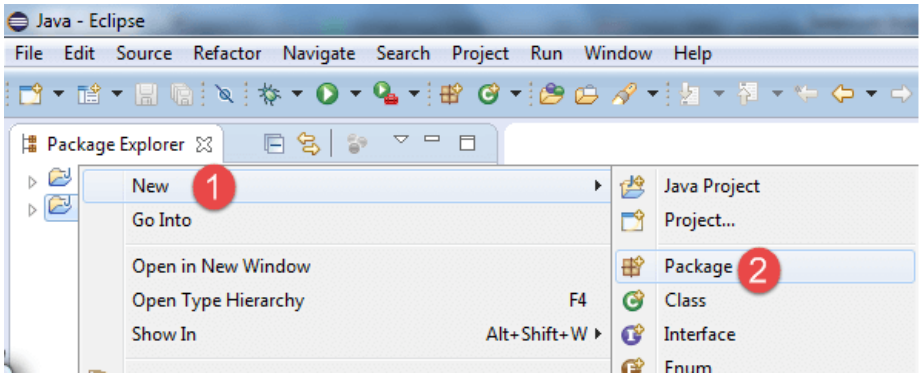
A new pop-up window will open. Enter details as follow

1. Project Name
2. Location to save a project
3. Select an execution JRE
4. Select the layout project option
5. Click on the Finish button



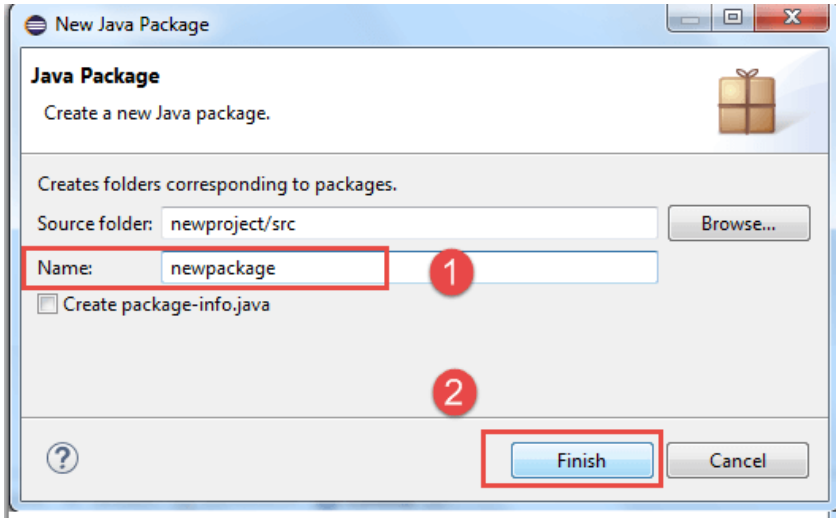
4. In this step,

1. Right-click on the newly created project and
2. Select New > Package, and name that package as “newpackage”.

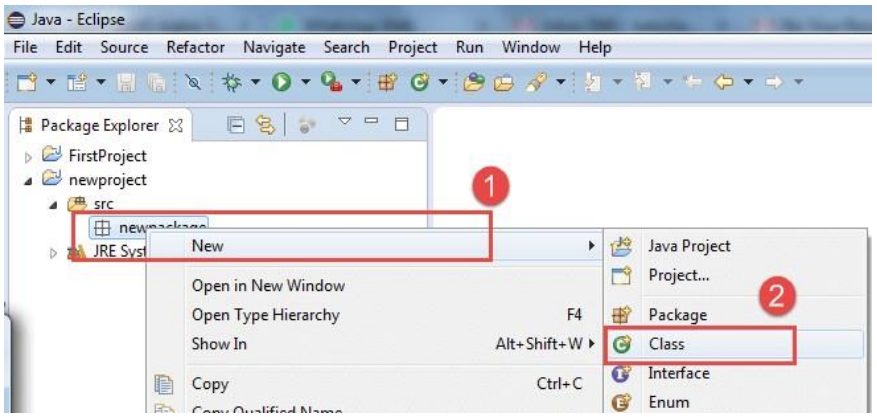


A pop-up window will open to name the package,

1. Enter the name of the package
2. Click on the Finish button

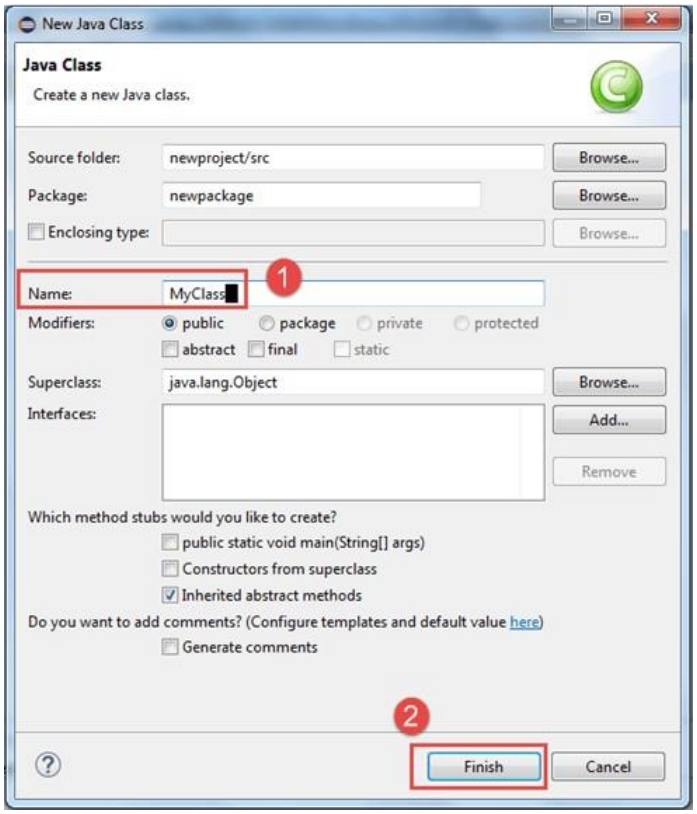


5. Create a new Java class under newpackage by right-clicking on it and then selecting- New > Class, and then name it as "MyClass". Your Eclipse IDE should look like the image below.

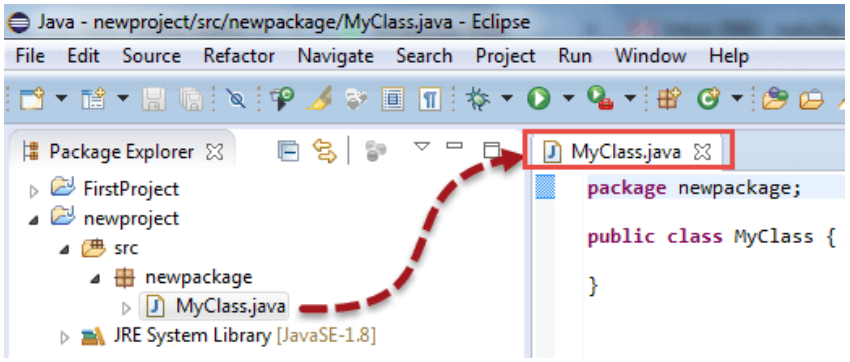


When you click on Class, a pop-up window will open, enter details as

1. Name of the class
2. Click on the Finish button



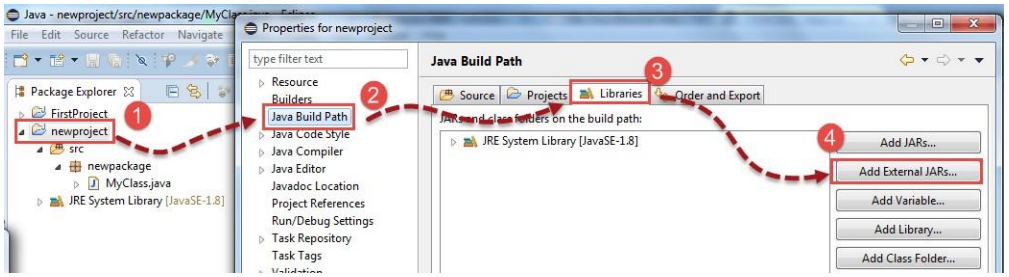
This is how it looks like after creating class.



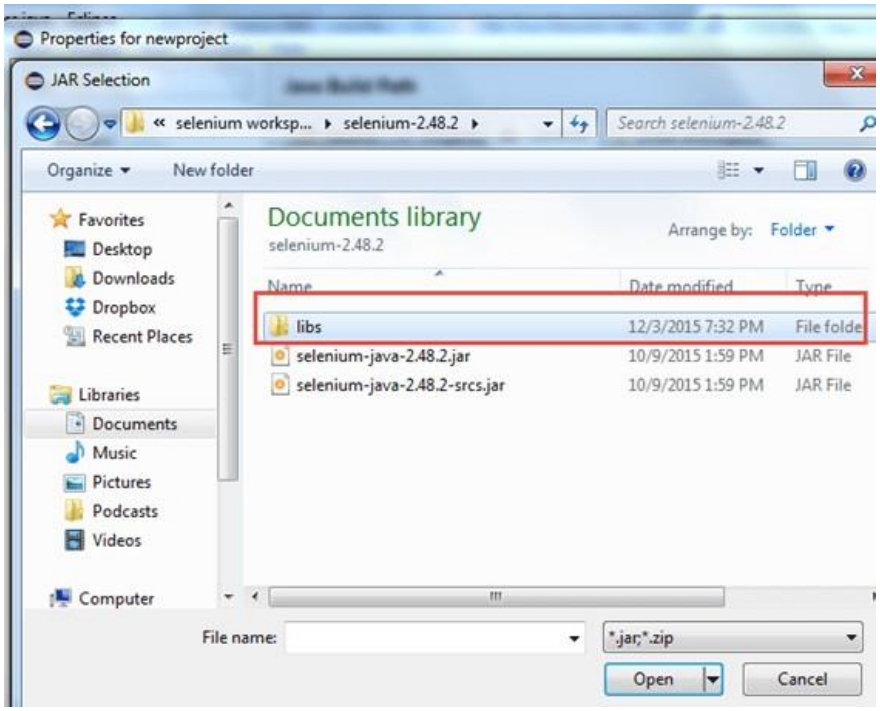
Now selenium WebDriver's into Java Build Path

In this step,

1. Right-click on "newproject" and select **Properties**.
2. On the Properties dialog, click on "Java Build Path".
3. Click on the **Libraries** tab, and then
4. Click on "Add External JARs.."

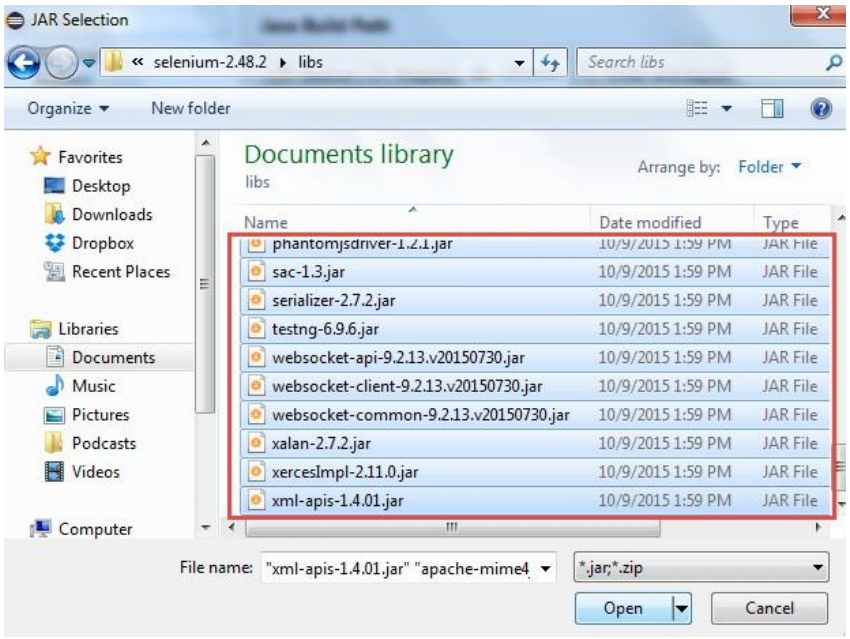


When you click on "Add External JARs.." It will open a pop-up window. Select the JAR files you want to add.

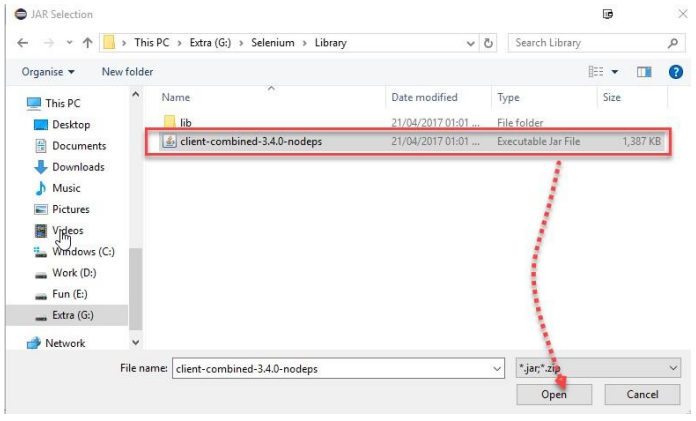


After selecting jar files, click on OK button.

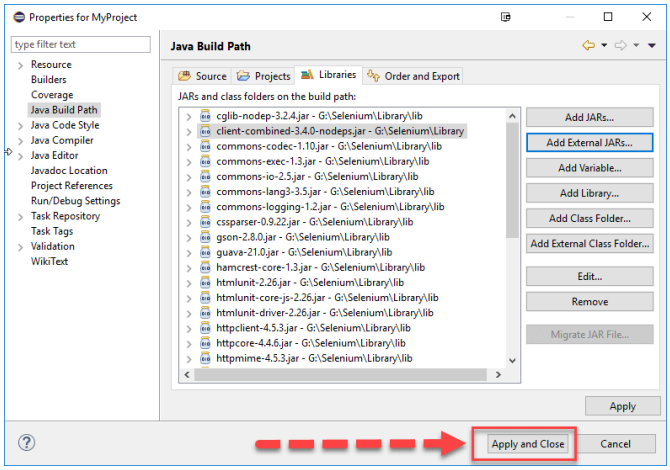
Select all files inside the lib folder.



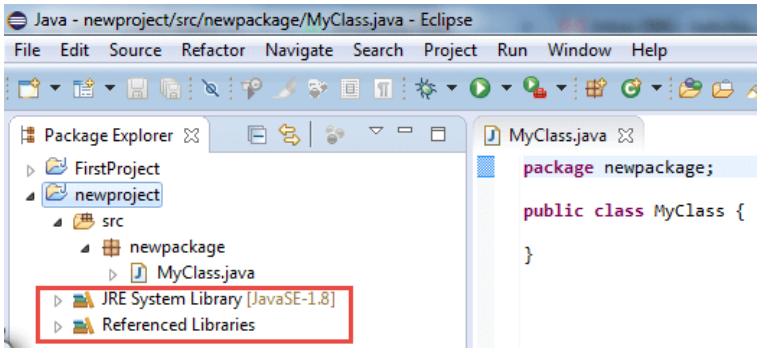
Select files outside lib folder



Once done, click “Apply and Close” button

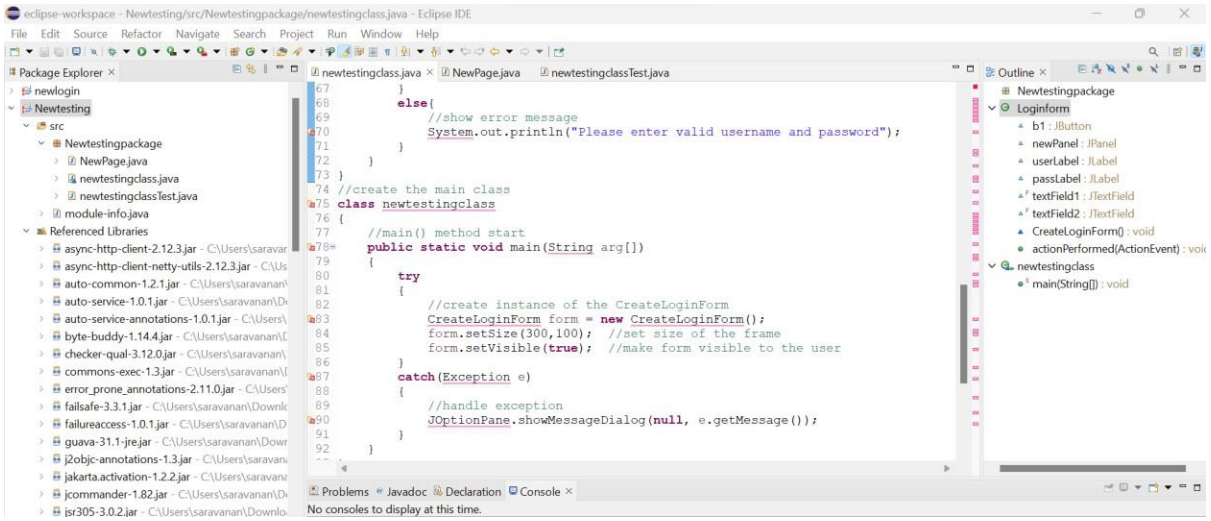


6. Add all the JAR files inside and outside the “libs” folder. Your Properties dialog should now look similar to the image below.



7. Finally, click OK and we are done importing Selenium libraries into our project.

Sample Test Case for the code



Result:

Thus, the designing unit test cases to verify the functionality and located bugs in the module of an application has been done successfully.

EX. NO: 10 Date:	IMPLEMENTATION AND TESTING OF THE MODIFIED SYSTEM USING SONAR CLOUD

AIM:

To design unit test cases to verify the functionality and locate bugs in the module of an application.

THEORY:

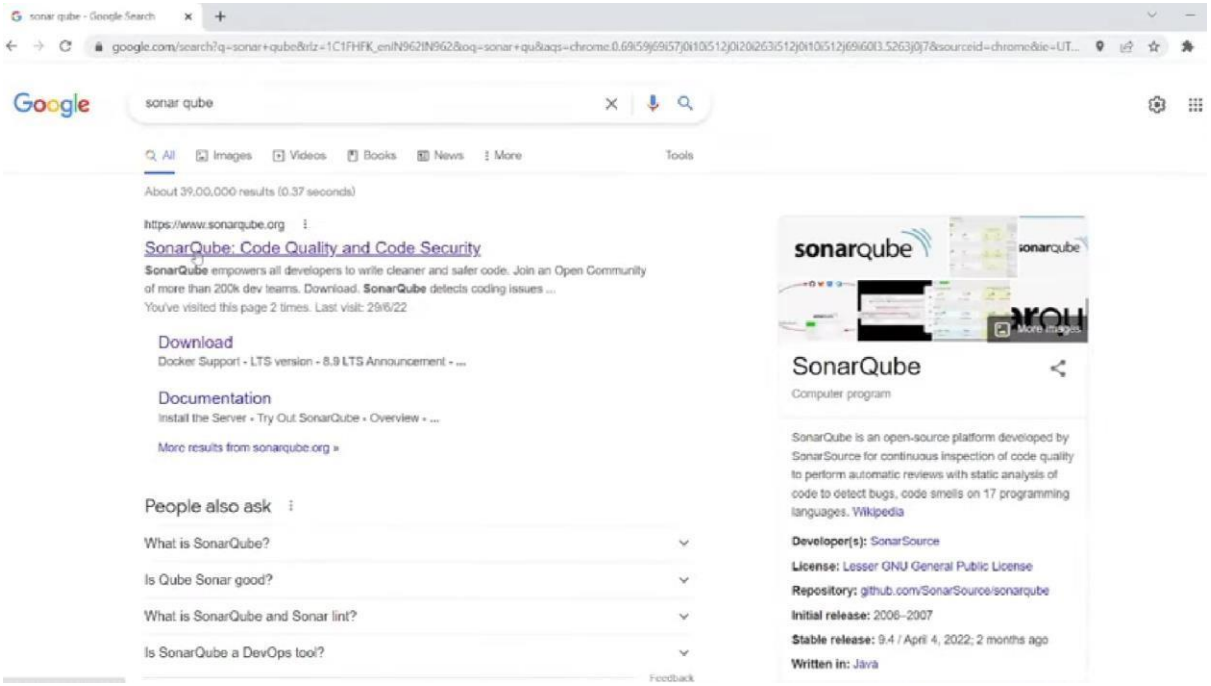
SonarQube platform significantly increases the lifetime of applications by reducing complexities, duplications and potential bugs in the code, by keeping neat and clean code architecture and increased unit tests. SonarQube increases maintainability of the software. It also has the ability to handle changes.

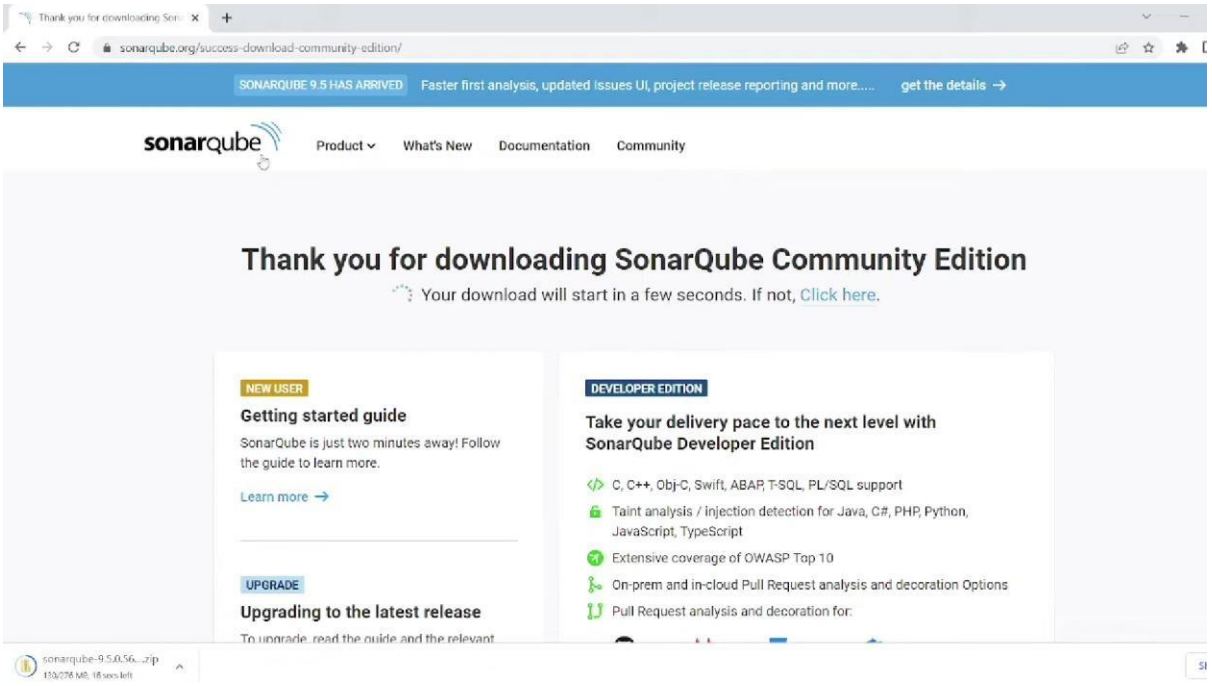
Benefits of SonarQube

- Sustainability – Reduces complexity, possible vulnerabilities, and code duplications, optimising the life of applications.
- Increase productivity – Reduces the scale, cost of maintenance, and risk of the application; as such, it removes the need to spend more time changing the code.

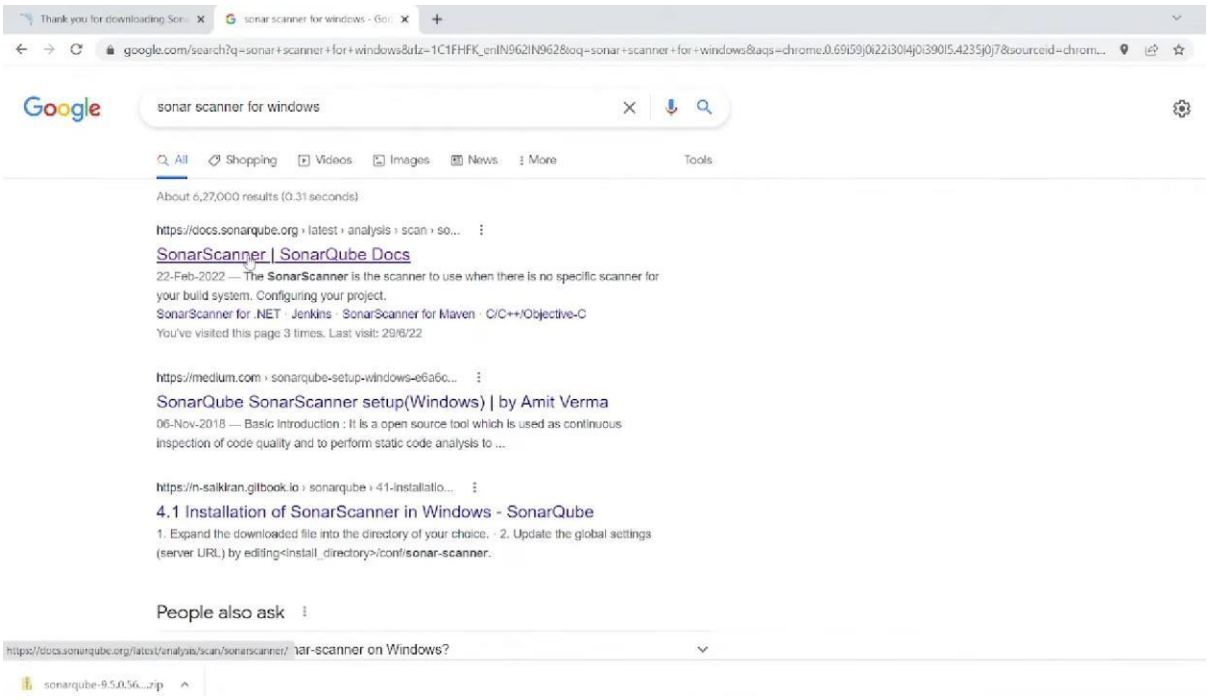
INSTALLATION AND CODE REVIEW PROCESS:

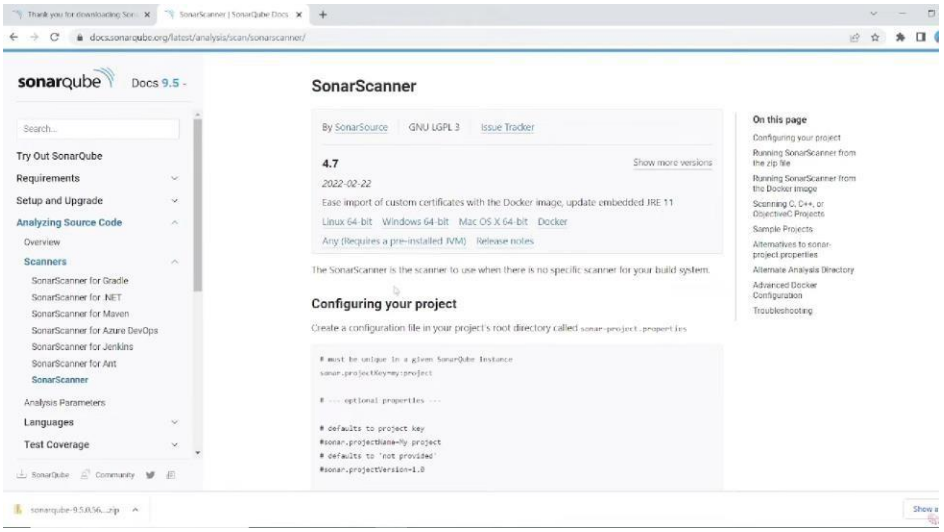
1. Download SonarQube



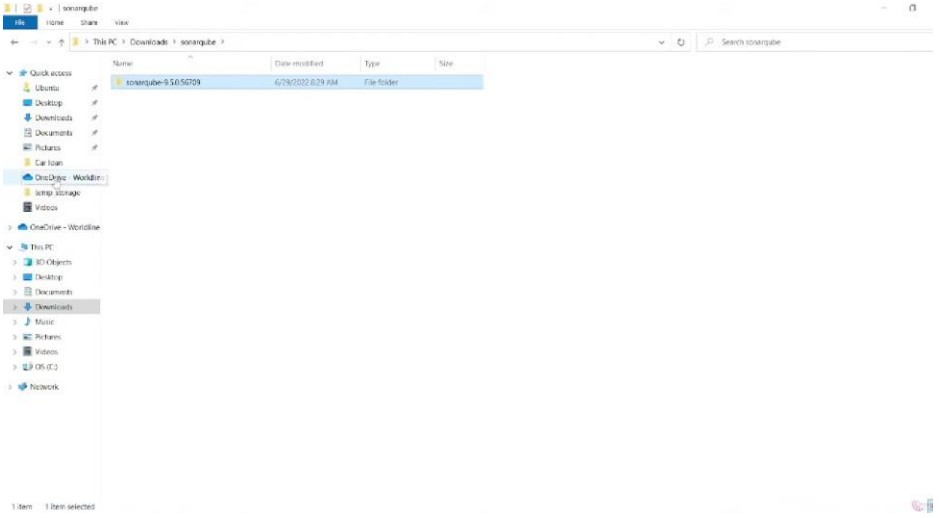


2. Download Sonar Scanner

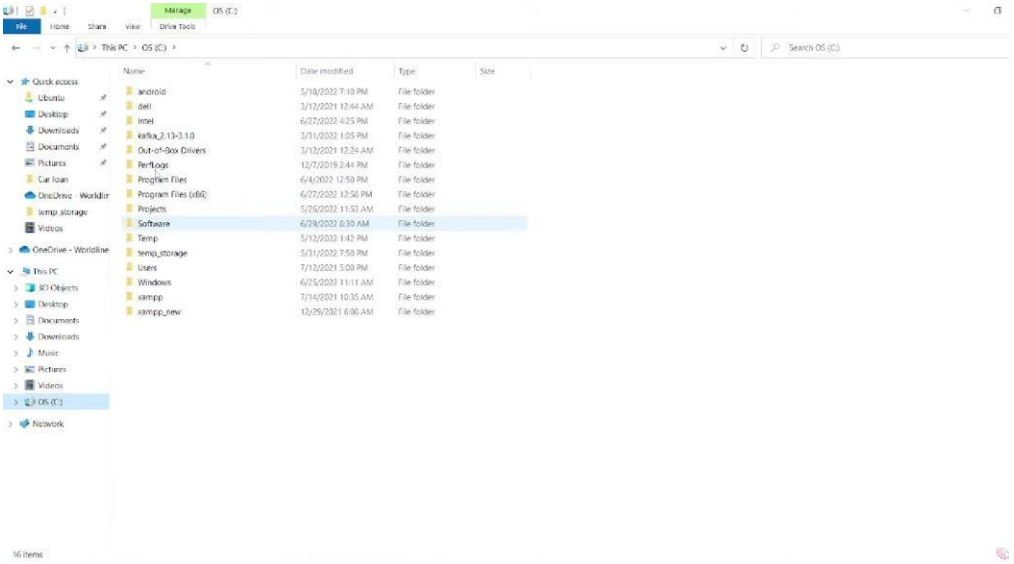




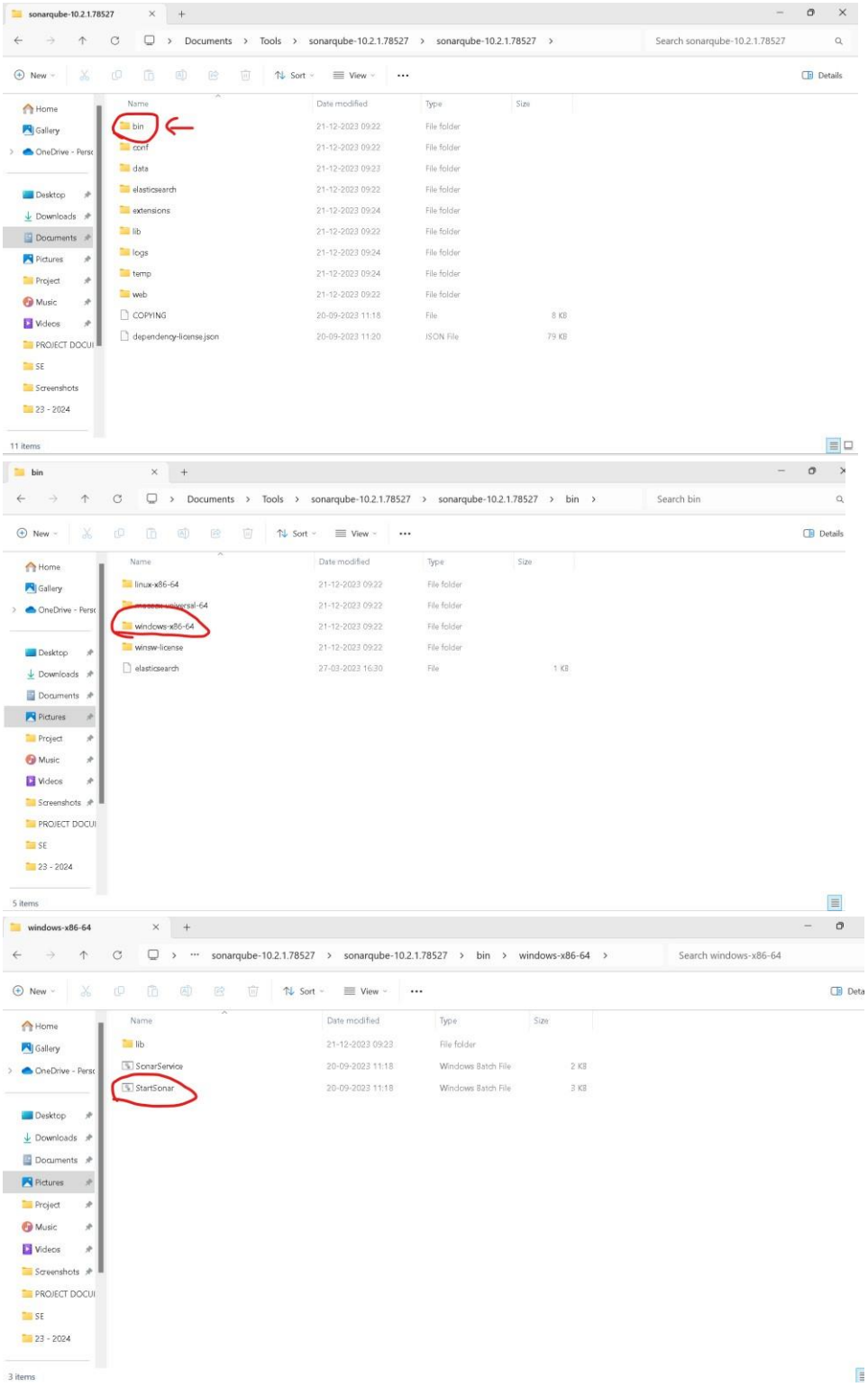
3. Copy the downloaded folder



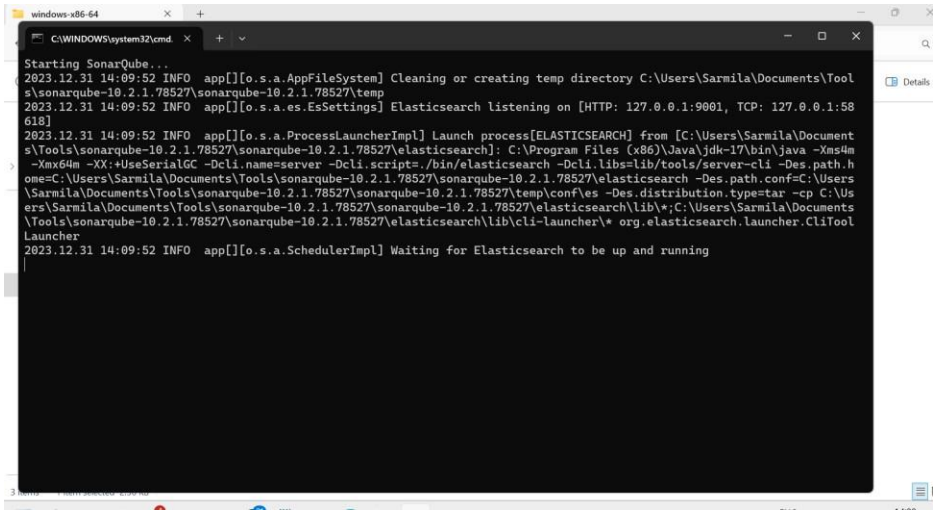
4. Paste it in C: drive inside. Place the both SonarQube and Sonar Scanner and code to be reviewed in same location



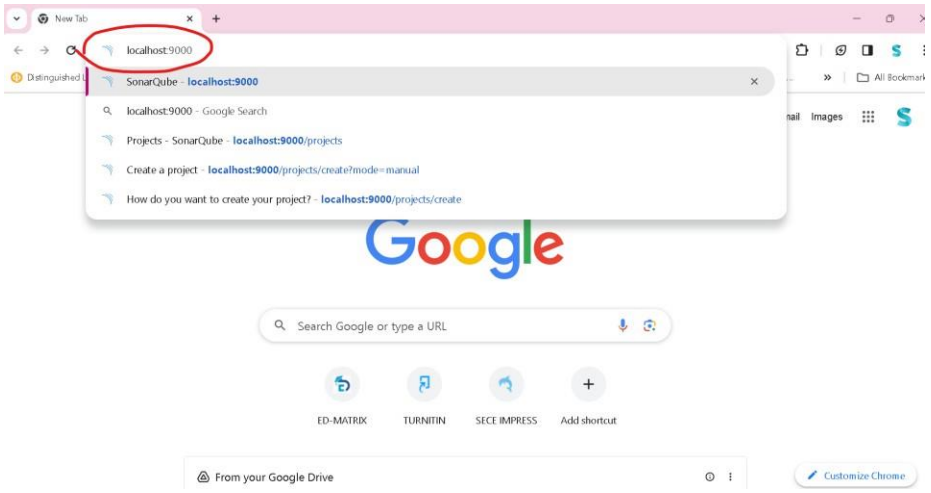
5. Go to SonarQube folder then go to bin-> select " windows x86-64" -> click on "StartSonar"



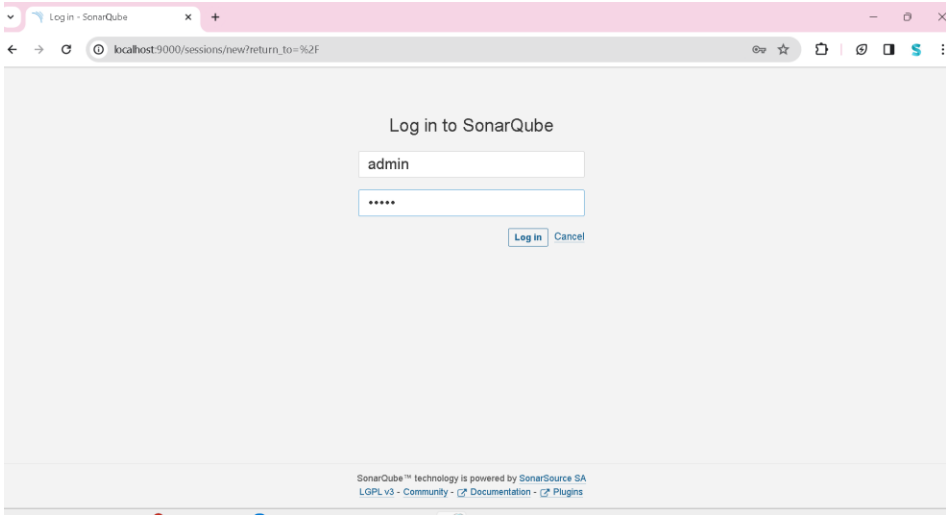
7. SonarQube will start



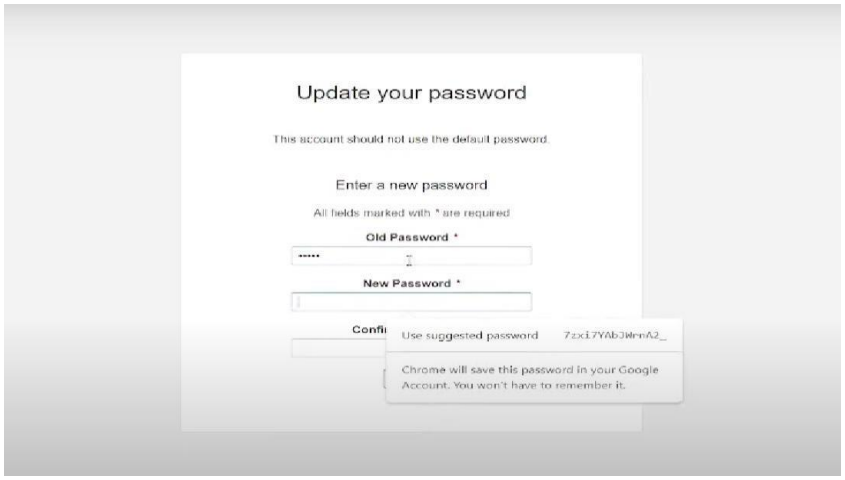
8. Go to web browser and type the URL “localhost:9000”



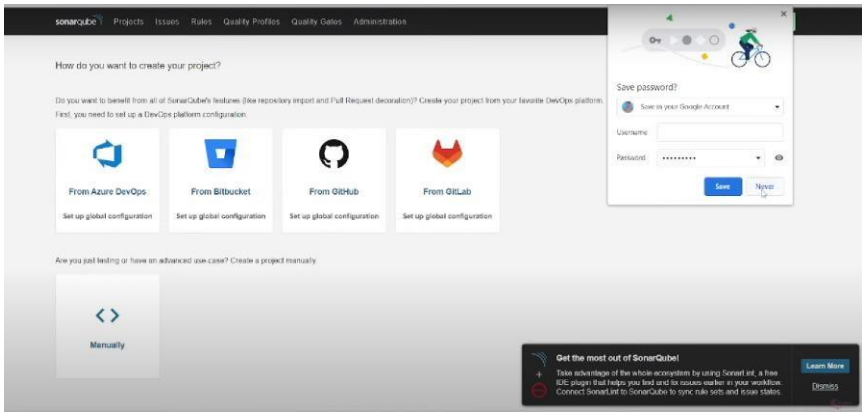
9. Login to SonarQube with username as “admin” and password as “admin”

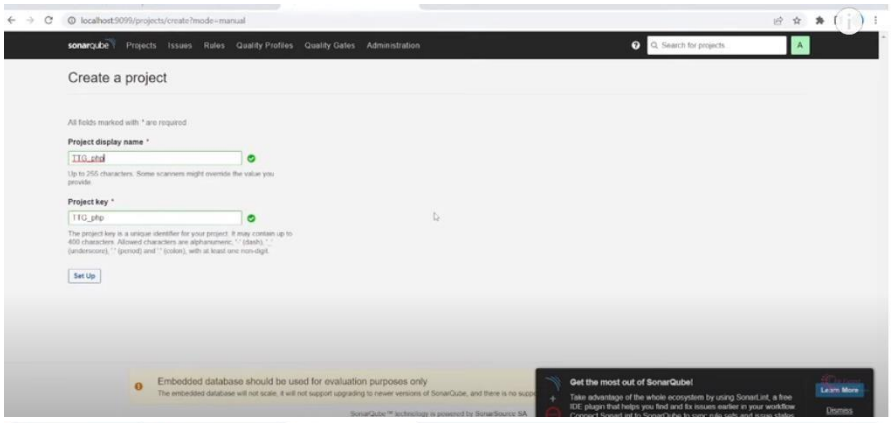


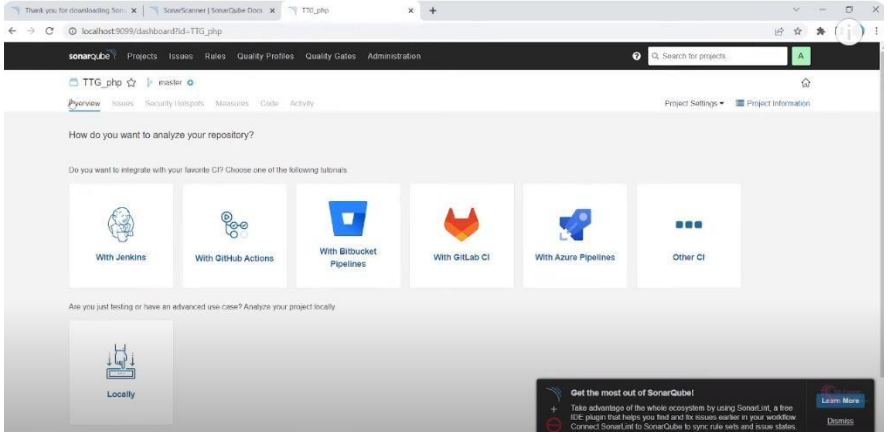
Update the password for future use

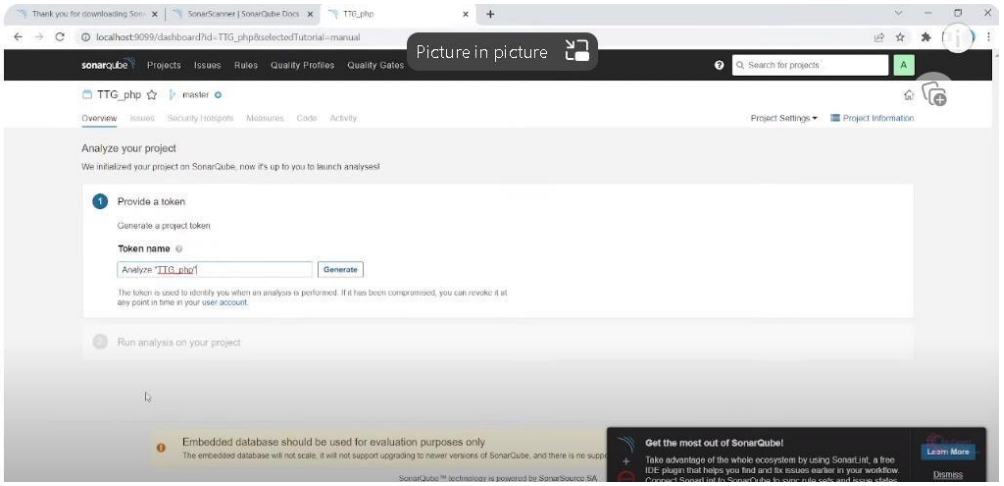


10. Select the project from GitHub or go to manually









Select the project and start the analysis of code

Result:

Thus, the implementation and testing of the modified System using Sonar Cloud has been done successfully.