EXPERIMENT 1

AIM

Write down the problem statement for a VIDEO GAME DISTRIBUTION SYSTEM.

THEORY

A video game digital distribution platform is used to distribute games and related media online and has community features such as friend lists and groups, the marketplace, in-game voice chat, and chat functionality.

PROBLEM STATEMENT FOR VIDEO GAME DISTRIBUTION SYSTEM

The objective is to develop a comprehensive Video Game Distribution System to automate and enhance the current manual processes within the industry. The system should be designed to facilitate efficient distribution of video games across various platforms. The functionalities required are as follows:

1. Game Distribution: Users should be able to browse and select video games for distribution. The system should provide a platform for game developers or publishers to submit their games, including relevant information such as game title, genre, platform compatibility, and pricing details.
2. Inventory Management: The system should maintain a centralized inventory of available video games, keeping track of stock levels, version updates, and any additional content associated with each game. This includes downloadable content (DLC), patches, and expansion packs.
3. Order Processing: Users, including distributors and retailers, should be able to place orders for video games through the system. The system must verify product availability, process orders efficiently, and generate invoices. It should also manage order status and provide notifications for successful transactions.
4. DRM Integration: Implement Digital Rights Management (DRM) mechanisms to protect intellectual property and prevent unauthorized distribution of video games. Ensure secure delivery of digital copies and enforce licensing agreements.
5. Reporting and Analytics: The system should offer robust reporting capabilities, allowing administrators to generate insights into sales trends, popular titles, and inventory turnover. This data can be instrumental in making informed business decisions.
6. User Authentication: Implement secure login mechanisms for different user roles, such as administrators, distributors, and retailers. Access to sensitive information and functionalities should be restricted based on user roles.
7. Platform Compatibility: Ensure the system is designed to support multiple distribution platforms, including online storefronts, physical retailers, and digital marketplaces. Compatibility with popular gaming consoles, PCs, and mobile devices is essential.

8. Marketing and Promotions: Provide tools for marketing and promotional activities, such as discounts, bundles, and special offers. The system should support the creation and management of promotional campaigns to boost sales.
9. Customer Support: Implement a customer support module to address queries, handle returns, and resolve issues related to game distribution. This includes a ticketing system and knowledge base for common troubleshooting.
10. Integration with Payment Gateways: Facilitate secure online transactions by integrating with reputable payment gateways. Support multiple payment methods, including credit/debit cards, digital wallets, and other relevant options.
11. Compliance and Licensing: Ensure compliance with industry regulations and licensing agreements. The system should support the tracking of licensing terms, royalties, and contractual obligations between developers, publishers, and distributors.

The successful implementation of this Video Game Distribution System aims to streamline the distribution process, enhance user experience, and contribute to the growth of the video game industry.

CONCLUSION

Problem statement of VIDEO GAME DISTRIBUTION SYSTEM has been written successfully.

EXPERIMENT 2

AIM

To do the requirement analysis and develop Software Requirement Specification Sheet for VIDEO GAME DISTRIBUTION SYSTEM.

REQUIREMENTS

1. SOFTWARE REQUIREMENT – Microsoft Word
2. HARDWARE REQUIREMENT – Computer, Keyboard, Mouse, CPU

THEORY

1. INTRODUCTION

1.1 Purpose

This document outlines the Digital Software and Games Distribution System - Steam, detailing its features, interfaces, and operational aspects. It serves as a guide for stakeholders and developers, influencing project approval or disapproval.

Objectives:

- Develop a secure digital distribution system.

- Enhance accessibility for customers, reducing physical store traffic.

- Minimize time spent on purchasing games.

- Streamline services requiring physical methods.

Audience:

(a) Customers: Project scope and security details.

(b) Developers (Clients): Project scope and use case module.

(c) Developers (Steam): Project scope and use case module.

(d) Project Manager: System features, hardware and software requirements, interface needs.

(e) Testers: Testing procedures.

(f) Documentation Writers.

1.2 Scope

The project focuses on online store operations, including account management, product purchases, refunds, and software updates. Key features:

- Any game provider can use the application for better customer service.

- Accessible account management for customers on various devices.
- Clients can publish games and plans.
- Reduced employee workload through online transactions.
- Potential for global communication via an online community/forum.

1.3 Definition, Acronym & Abbreviation

- HTML: Hypertext Markup Language.
- EJB: Enterprise Java Beans.
- J2EE: Java 2 Enterprise Edition.
- DB2: DB2 Database.
- WAS: Web sphere application server.
- WSAD: Web sphere studio application developer.
- HTTP: Hypertext Transfer Protocol.
- HTTPS: Secure Hypertext Transfer Protocol.
- TCP/IP: Transmission Control Protocol/Internet Protocol.

1.4 References

References from online sources and textbooks, including:

- "Software Requirements Specification and Analysis" by Nancy Day.
- IEEE Recommended Practice for Software Requirements Specifications.
- Wikipedia, the Free Encyclopedia.
- Various textbooks including "Java Complete Reference" and "Software Engineering."

1.5 Overview

The Digital Software and Games Distribution System - Steam project entails comprehensive information on account details, purchase history, social networking, and games accessibility. It facilitates account management for customers and allows Clients or Software Publishers to offer products on the platform for a 10% revenue share.

2. OVERALL DESCRIPTION

2.1 Product Perspective

2.1.1 System Interface

The system interface encompasses User Interface, Hardware Interface, and Software Interface.

2.1.2 Interface

Three user interactions:

- New User: Opens an account for game/software purchases.
- Existing User: Logs in, performs transactions, accesses social features.
- Administrator: Manages user database, permissions, and resolves discrepancies.

2.1.3 Hardware Interfaces

- Client Side: Requires 2.0 GHz processor, 2 GB RAM, 1GB disk space, DX10 Compatible graphics.
- Server Side: Requires 2.3 GHz Xeon E5 2686 CPU, 32 GB RAM, 10TB disk space.

2.1.4 Software Interfaces

- Operating System: Windows 7 32-bit SP1 or better.
- Web Browser: Google Chrome or Mozilla Firefox (JavaScript and Flash enabled).
- Front End: REACT JS.
- Back End: NODE JS.
- RDBMS: MS ACCESS.
- DBMS: MS-SQL SERVER.

2.1.5 Communication Interfaces

- Internet clients use HTTP/HTTPS protocol.
- Intranet clients use TCP/IP protocol.
- Web Browser like IE 6.0 or equivalent.

2.1.6 Memory Constraints

At least 2GB of RAM and 1GB of hard disk.

2.1.7 Operation

- Two methods: user-based (mobile/desktop/web) and publisher-based (web interface).
- Admin controls database, content, and permissions through a GUI.
- Functions include authentication, purchase reviews, customer usage stats, forum monitoring.

2.2 Product Functions

Internet-based system modules:

1. Login Process: Allows valid customer access.

2. Product Enquiry: Maintains software details.

3. Update Profile: Allows customer profile updates.

4. Purchase Product: Enables various purchase methods.

5. Change of Password: Allows customers to change passwords.

6. Mini Statements: Allows customers to view transaction details.

2.3 User Characteristics

Registered users can access accounts, purchase details, refund requests, change passwords, and participate in forums.

2.4 Constraints

- Identification through login and password.

- Multiple servers handle system modules.

- Designed using HTML, CSS, and JavaScript.

- GUI is in English.

- Limited to HTTP/HTTPS protocols.

2.5 Assumptions & Dependencies

Assumptions:

- Users manually input details.

- Users are comfortable with computers and internet.

- Basic English knowledge required.

2.6 Apportioning Of Requirements

No delayed services for future versions.

3. SPECIFIC REQUIREMENTS

3.1 External Interfaces

The major interfaces: User, Hardware, Software, and Communication interfaces are covered above. There are no other external interfaces.

3.2 Functions

Services provided by the system include:

- Online Account check.
- Online shopping opportunity.
- Online data entry by the staff.
- Updating the data.
- Balance check.
- Refund services if the product has not been used for a long period.
- Community Forums for users to interact with each other and the publishers.

3.3 Performance Requirements

The system can withstand numerous customer requests. Access is given only to valid users requiring services like purchase enquiry, update profile, refund, mini statements, etc. It is available 24/7.

3.4 Logical Database Requirements

The database includes recent files and plug-ins.

3.5 Design Constraints

- Login and password are used for customer account identification.
- Designed using HTML, CSS.
- Language used is JavaScript.
- GUI is only in English.
- Limited to HTTP/HTTPS protocols.

3.6 Software System Attributes

3.6.1 Reliability

The system ensures safety; if abnormal operation occurs, the user is logged out, pausing actions until login. The system must be reliable due to the importance of data.

3.6.2 Availability

Request handling within 1 second under normal conditions. System available 100%, usable 24/7, operational every day.

3.6.3 Security

Security mechanisms prevent unwanted server pings and isolate the system from malicious code.

3.6.4 Maintainability

Design documents describe internal workings. Control panel and server access for upgrades.

3.6.5 Portability

No portability requirements.

3.7 Organizing Specific Requirement

3.7.1 System Mode

The application runs on Windows, Linux, Mac, Android, iOS, and compatible web browsers.

3.7.2 User Class

- Customers: Individuals purchasing software.
- Software Publisher Clients: Manage developed products.
- Steam Administrator: Manages product details.

3.7.3 Objects

- Viewers: Unauthenticated visitors.
- New User: Registers for an account.
- Existing User: Performs various operations after login.
- Administrator: Manages the database.

3.7.4 Features

Customers require a valid User Id and password to log in. They can view purchased products, check for updates, and buy new products. Refund requests and transaction statements are also available.

3.7.5 Stimulus

The increasing variety of software led to the need for a digital distribution system, simplifying software purchase, updating, and usage.

3.8 Additional Comments

NONE

4. CHANGE MANAGEMENT PROCESS

It will immediately respond to the change in platform (OS). There will be two different interfaces, one for Mac users and other for Windows to make it user friendly for both the platforms. The system for mobile devices will be updated quite regularly including beta versions.

5. Document Approval

This document is approved by Ms. Prerna Sharma, Professor, Department of Information

Technology, Maharaja Agrasen Institue of Technology, Delhi.

6. Supporting Information

To support the above software requirements specifications, there is a table of content:

EXPERIMENT 3

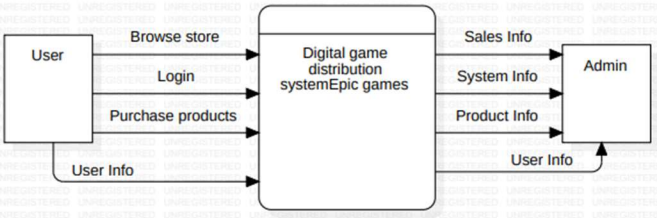
AIM

To Perform the function oriented diagram : Data Flow Diagram (DFD) and Structured diagram.

THEORY

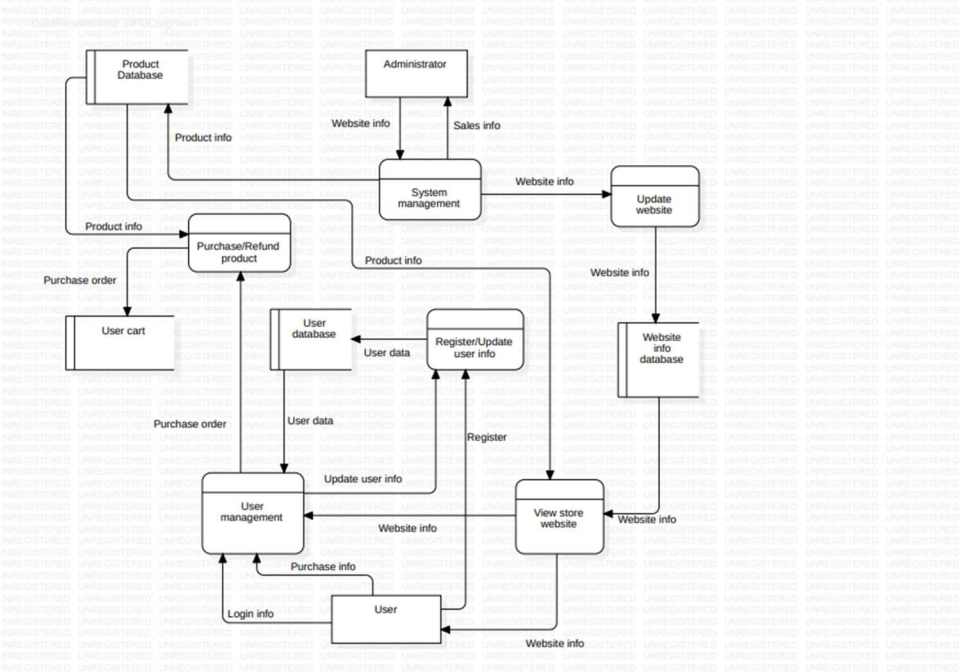
Level 0:

This level is also known as Context level DFD. These levels are Level 0 DFD . Both these levels are



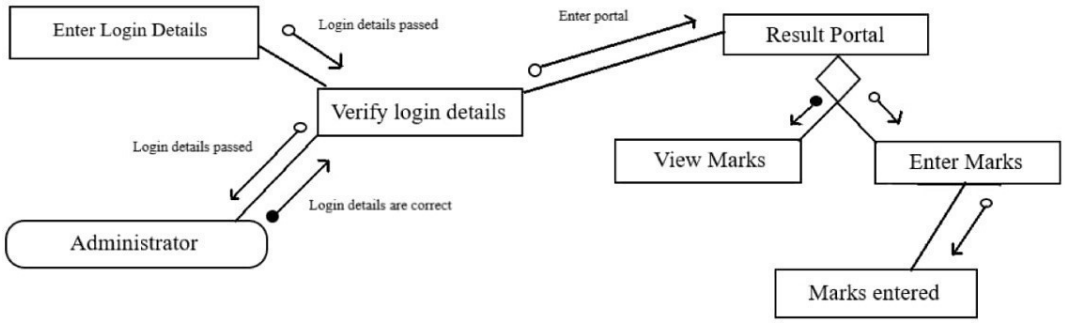
Level 1 DFD:

At this level, more detailed information is given about the processing of the student result management system. The DFD of this level is shown below:



STRUCTURE CHART

Structure chart represents hierarchical structure of modules. It breaks down the entire system into the lowest functional modules, describing function of each module of a system into a greater detail. Structure Chart partitions the system into the black boxes. Inputs are given to the boxes and appropriated outputs are generated.



CONCLUSION:-

Level – 0 and Level – 1 Data Flow Diagram (DFD) and Structure Chart for STUDENT RESULT MANAGEMENT SYSTEM have been drawn successfully.

EXPERIMENT 4

AIM

To prepare an ER diagram for online gaming platform

SOFTWARE USED

Star UML

THEORY

ER modeling is a data modeling method used in software engineering to produce an information system. It begins with a conceptual data model of the real world, and then refines it into a logical data model. The logical data model is then transformed into a physical database schema.

Entities

An entity is a person, place, thing, or event that is of interest to the business. Entities are represented by rectangles in ER diagrams.

Attributes

Attributes are the properties of entities. Entities are described by their attributes. Attributes are represented by ovals in ER diagrams.

Relationships

Relationships are the associations between entities. Relationships are represented by lines in ER diagrams.

Example:

Consider a simple online gaming platform. The following entities and attributes could be identified:

User (user_id, username, email, password)

Game (game_id, game_name, genre, platform)

Score (score_id, user_id, game_id, score)

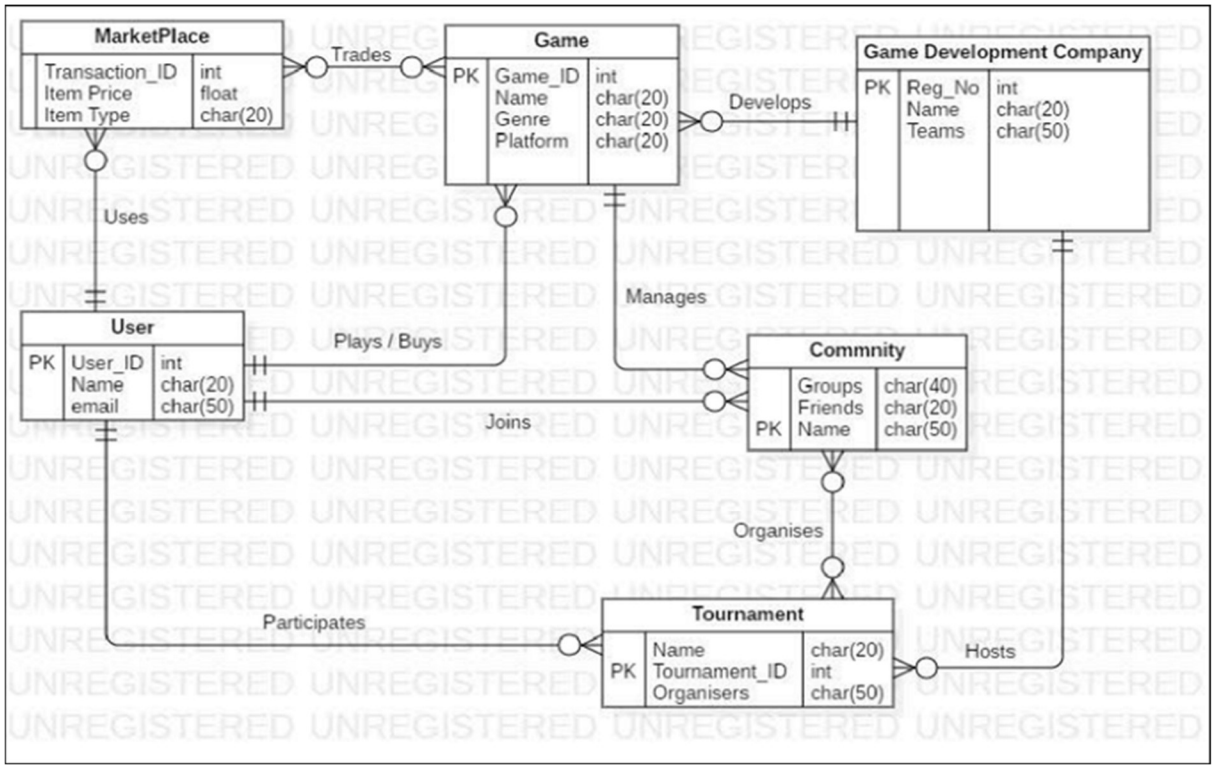
The following relationships could be identified:

A user can play multiple games.

A game can be played by multiple users.

A user can have multiple scores for a game.

A game can have multiple scores for a user.



CONCLUSION

The ER Diagram was successfully created.

EXPERIMENT 5

AIM

To design use case diagram of online gaming platform.

SOFTWARE USED

Star UML

THEORY

A Use Case Diagram in software engineering provides a visual representation of how a system interacts with external entities, known as actors, to accomplish specific goals or use cases. Each use case describes a particular functionality or unit of work, illustrating the sequence of interactions between the system and actors. Actors, represented as stick figures, initiate use cases and define the system's boundaries, encapsulated within a box. Associations depict relationships between actors and use cases, highlighting their involvement. Include relationships signify that one use case incorporates the functionality of another, while extend relationships indicate optional or conditional behavior extending a base use case. Generalization represents inheritance between use cases, and extension points are specific locations within a use case where extended behavior can be added.

The diagram may also show the system's modular structure through components, subsystems, and the associated use cases. Multiplicity notation indicates how many instances of an actor can be associated with a particular use case. Overall, Use Case Diagrams serve as a valuable tool for capturing and communicating high-level functional requirements, fostering collaboration among stakeholders during the early stages of system design.

- **Use Case:**

  Describes a specific interaction between a system and external entities to achieve a goal.

- **Actor:**

  Represents an external entity (user, system, or device) interacting with the system.

- **System Boundary:**

  A box that defines what's inside (use cases) and outside (actors) the system.

- **Association:**

  Shows relationships between actors and use cases, indicating involvement.

- **Include Relationship:**

  Depicts that one use case includes the functionality of another.

- **Extend Relationship:**

  Represents optional or conditional behavior extending a base use case.

- **Generalization:**

  Indicates inheritance between use cases, showing parent-child relationships.

- **Extension Points:**

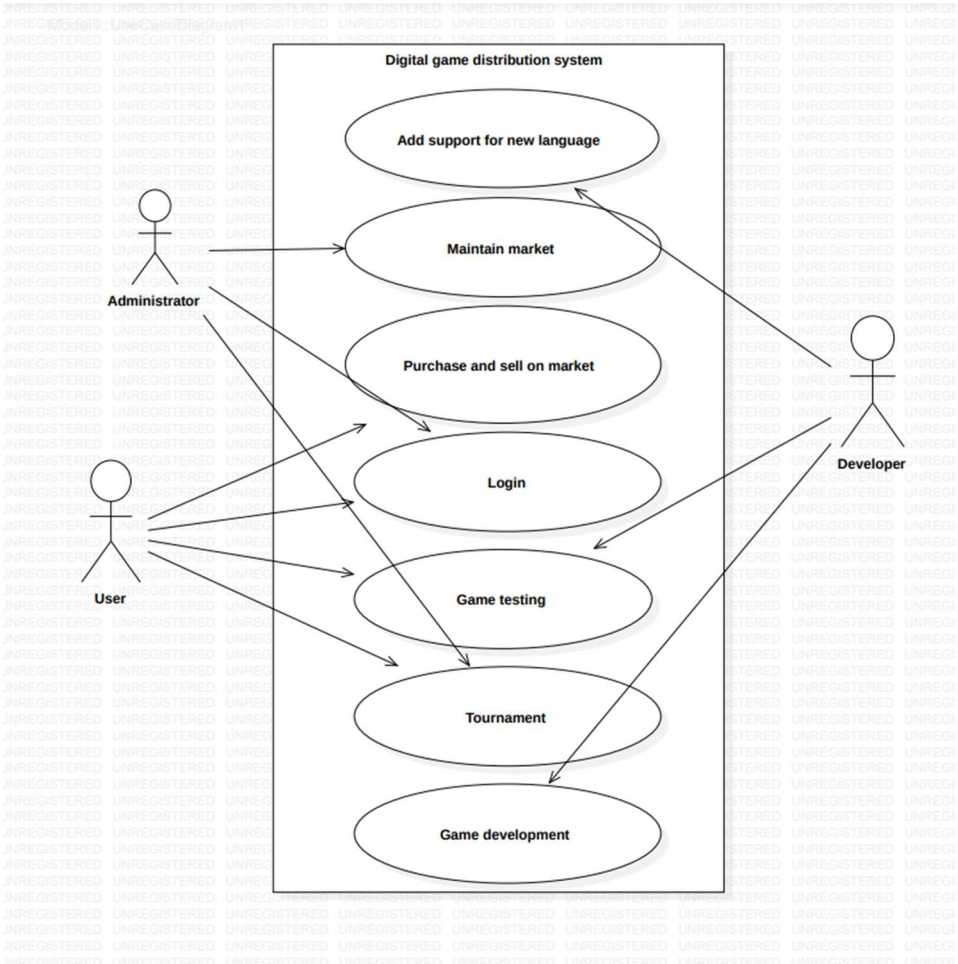
  Specific points in a use case where extended behavior can be added.

- **System, Subsystem, Component:**

Represents the modular structure of the system.

- **Multiplicity:**

  Indicates how many instances of an actor are associated with a use case.



CONCLUSION

The Use Case Diagram was successfully created.

EXPERIMENT 6

AIM

To draw the structural view diagram for the system: Class diagram, object diagram

SOFTWARE USED

Star UML

THEORY

Class Diagram:

A Class Diagram is a static structural diagram that depicts the classes in a system, their attributes, relationships, and the constraints governing these elements. Key theoretical aspects include:

1. **Class:**
    a. Represents a blueprint for creating objects, encapsulating data attributes and methods.
2. **Attributes:**
    a. Characteristics or properties of a class, describing the data it holds.
3. **Methods:**
    a. Functions or operations that can be performed by objects of a class.
4. **Association:**
    a. Represents relationships between classes, indicating how they are connected.
5. **Multiplicity:**
    a. Specifies the number of instances participating in a relationship between classes.
6. **Inheritance:**
    a. Represents an "is-a" relationship, where a subclass inherits attributes and behaviors from a superclass.
7. **Abstract Class:**
    a. A class that cannot be instantiated and may contain abstract methods to be implemented by its subclasses.

Object Diagram:

An Object Diagram is an instance of a Class Diagram, capturing a snapshot of the system at a specific point in time. The key theoretical concepts include:

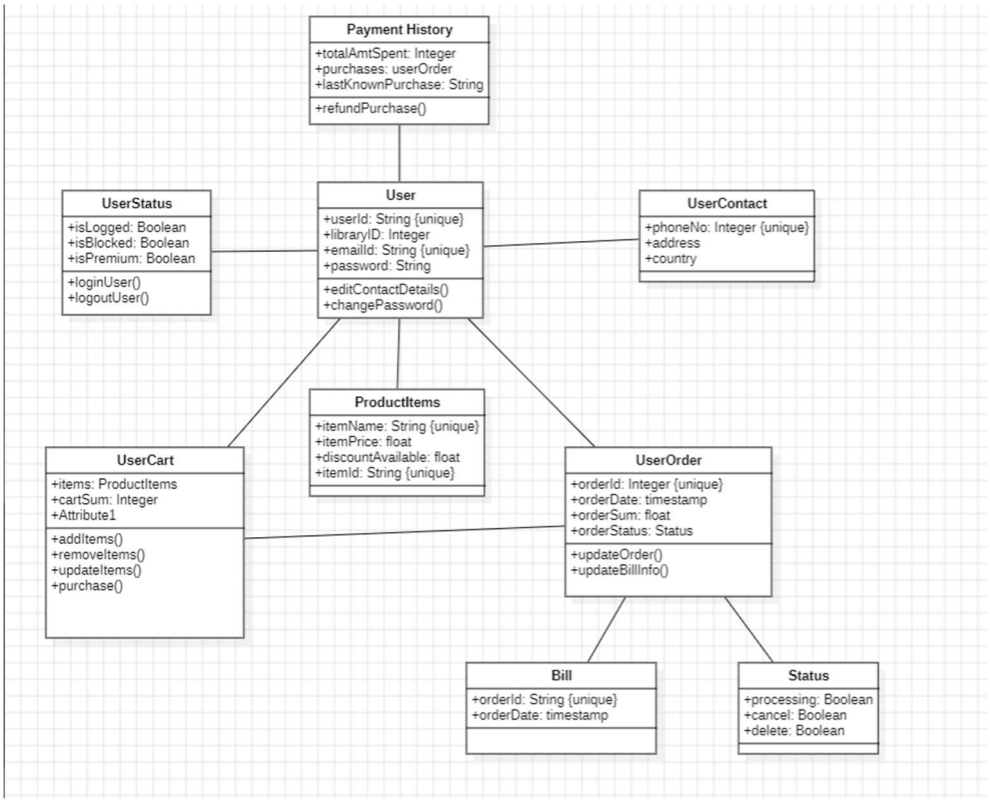
1. **Object:**
    a. Represents an instance of a class, with specific values for its attributes.
2. **Link:**
    a. Represents a connection between objects, showing relationships as they exist in a particular instance.
3. **Multiplicity:**
    a. Indicates the number of instances participating in a link between objects.
4. **Role:**
    a. Describes the way an object participates in a particular relationship.

Relationship Between Class and Object Diagrams:

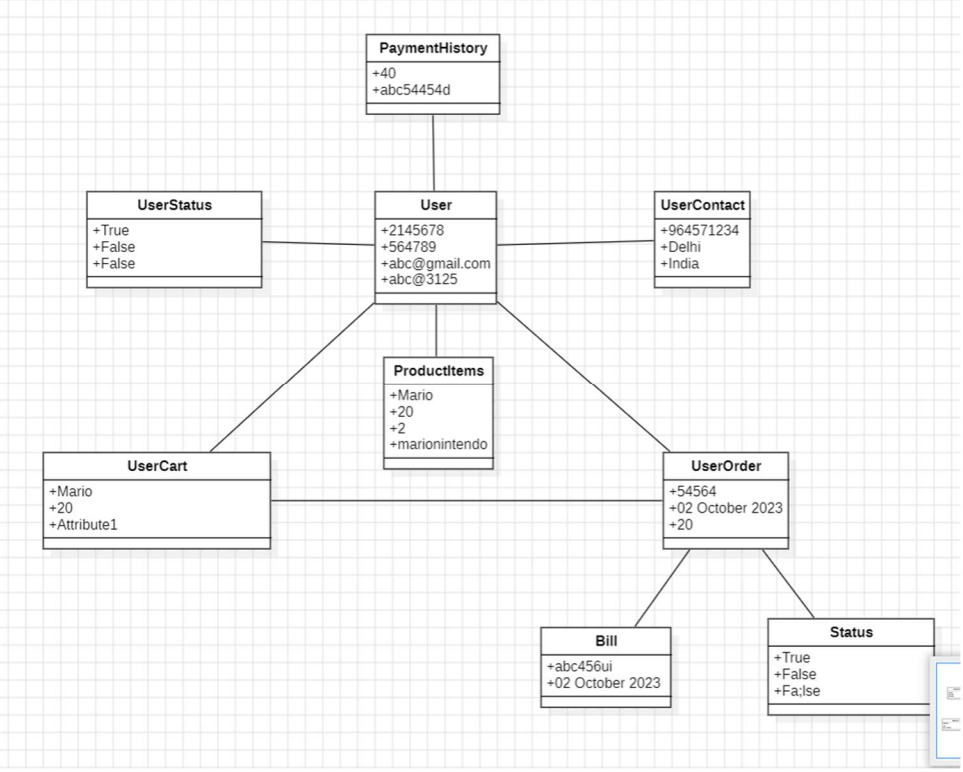
- **Abstraction:**

    - Class Diagrams provide a blueprint for the system's structure, while Object Diagrams instantiate these classes to represent specific instances.
- **Dynamic Aspect:**
    - Class Diagrams focus on static aspects, depicting the structure, while Object Diagrams capture the dynamic relationships between objects at a specific moment.
- **Visualization:**
    - Class Diagrams are used for system design and planning, while Object Diagrams are beneficial for visualizing specific scenarios and instances during implementation.

Together, Class and Object Diagrams form a powerful duo in UML, facilitating the understanding and communication of a system's static structure and dynamic behavior throughout the software development lifecycle.



Class Diagram



Object Diagram

CONCLUSION

The Class Diagram and Object Diagram were successfully created.

EXPERIMENT 7

AIM

To draw the behavioral view diagram: State-chart diagram, Activity diagram.

SOFTWARE USED

Star UML

THEORY

STATE CHART DIAGRAM:

The name of the diagram itself clarifies the purpose of the diagram and other details. It describes different states of a component in a system. The states are specific to a component/object of a system. A Statechart diagram describes a state machine. State machine can be defined as a machine which defines different states of an object and these states are controlled by external or internal events. Purpose of state chart diagrams Statechart diagrams are also used for forward and reverse engineering of a system. However, the main purpose is to model the reactive system. Following are the main purposes of using Statechart diagrams −

- To model the dynamic aspect of a system.
- To model the life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model the states of an object.

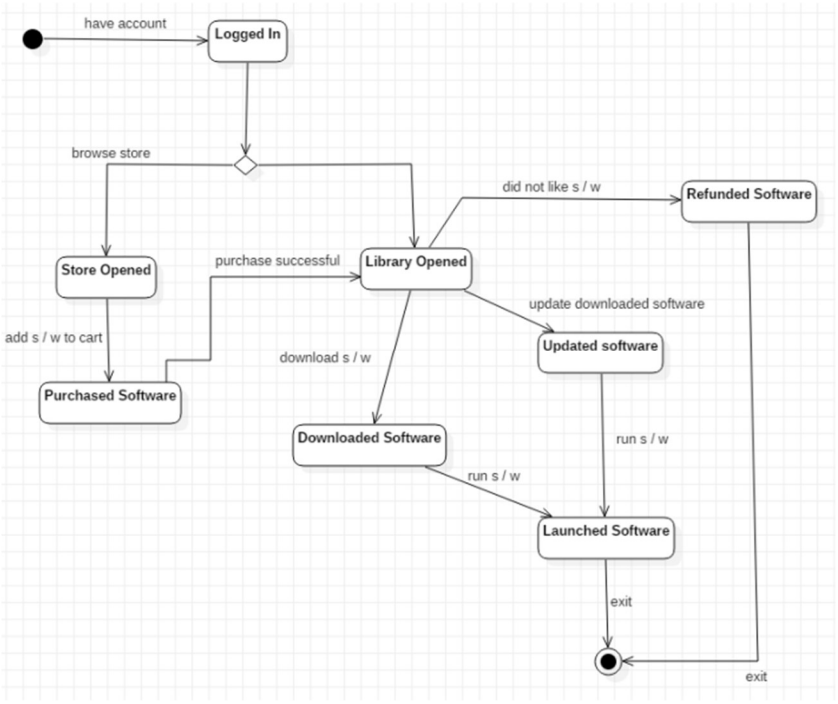


FIG 7.1: STATE CHART DIAGRAM

ACTIVITY DIAGRAM:

Activity diagrams describe the activities of a class. They are similar to state transition diagrams and use similar conventions, but activity diagrams describe the behavior/states of a class in response to internal processing rather than external events. They contain the following elements:

1. Swimlanes , which delegate specific actions to objects within an overall activity
2. Action States , which represent uninterruptible actions of entities, or steps in the execution of an algorithm
3. Action Flows , which represent relationships between the different action states on an entity
4. Object Flows , which represent utilization of objects by action states, or influence of action states on objects

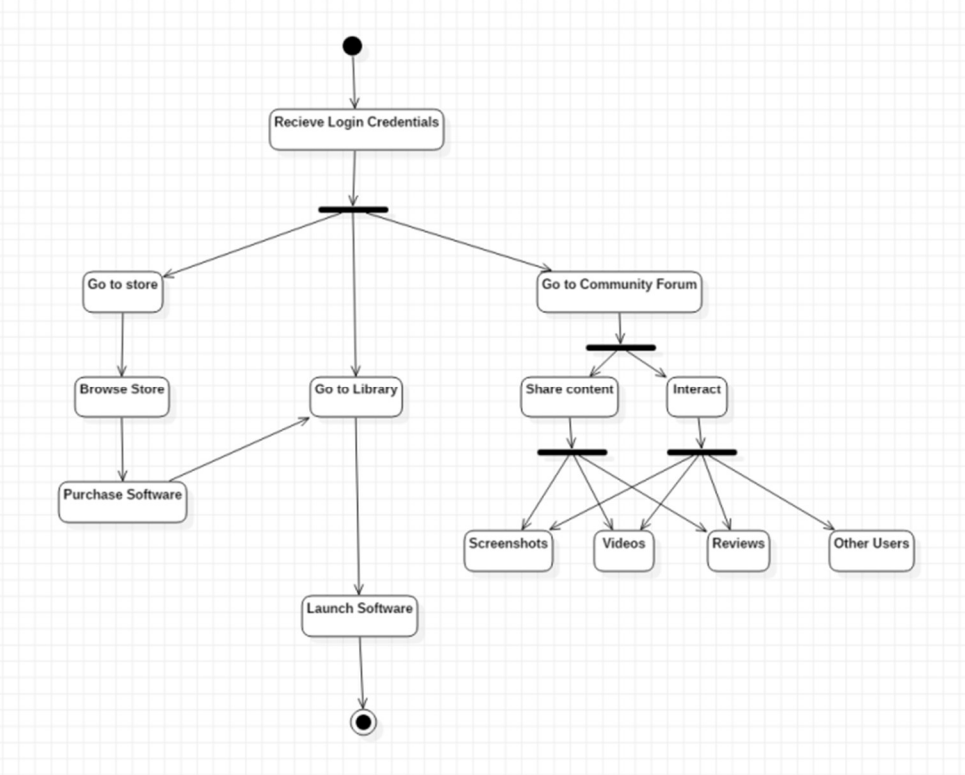


FIG 7.2: ACTIVITY DIAGRAM

CONCLUSION

The State Chart Diagram and Activity Diagram were successfully created.

EXPERIMENT 8

AIM

To perform the implementation view diagram: Component diagram for the system.

SOFTWARE USED

Star UML

THEORY

COMPONENT DIAGRAM:

A component diagram is used to break down a large object-oriented system into the smaller components, so as to make them more manageable. It models the physical view of a system such as executables, files, libraries, etc. that resides within the node.It visualizes the relationships as well as the organization between the components present in the system. It helps in forming an executable system. A component is a single unit of the system, which is replaceable and executable. The implementation details of a component are hidden, and it necessitates an interface to execute a function. It is like a black box whose behavior is explained by the provided and required interfaces.

PURPOSE OF COMPONENT DIAGRAM

Since it is a special kind of a UML diagram, it holds distinct purposes. It describes all the individual components that are used to make the functionalities, but not the functionalities of the system. It visualizes the physical components inside the system. The components can be a library, packages, files, etc.

1. The component diagram also describes the static view of a system, which includes the organization of components at a particular instant. The collection of component diagrams represents a whole system. The main purpose of the component diagram are enlisted below:It envisions each component of a system.

2. It constructs the executable by incorporating forward and reverse engineering.

3. It depicts the relationships and organization of components.

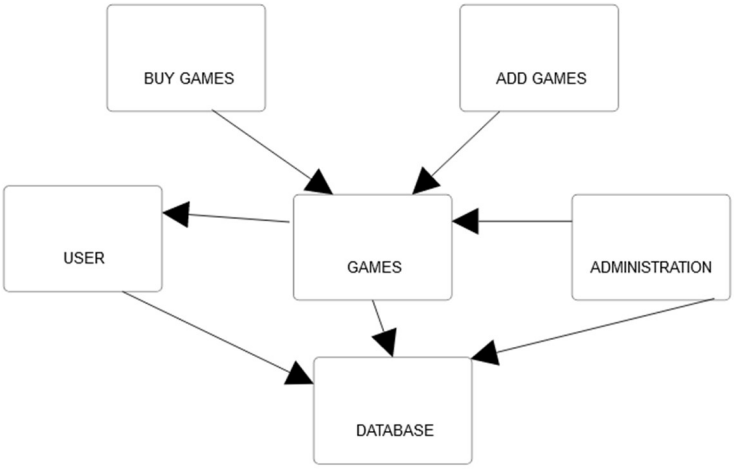


FIG 8.1: COMPONENT DIAGRAM

CONCLUSION

The Component Diagram was successfully created.

EXPERIMENT 9

AIM

To perform the behavioral view diagram for the suggested system: Sequence diagram, Collaboration diagram.

SOFTWARE USED

Star UML

THEORY

Sequence Diagram:

A Sequence Diagram is a dynamic model that represents the interactions between different components or objects over time. Key theoretical aspects include:

1. **Lifeline:** Vertical dashed lines representing the existence of an object over a period.
2. **Message:** Arrows indicating the flow of communication between objects or components.
3. **Activation Bar:** Represents the duration of an object's activity during a specific interaction.
4. **Object (Participant):** Entities involved in the interaction, usually represented along the top of the diagram.
5. **Focus of Control:** Indicates which object is currently in control of the flow of the interaction.
6. **Return Message:** Represents the flow of control back to the calling object after the execution of a method.
7. **Interaction Fragment (Combined Fragment):** A grouping construct to represent alternative or parallel flows in the sequence.

Sequence Diagrams are especially useful for visualizing the chronological order of interactions between objects or components, making them effective for understanding the dynamic behavior of a system.

Collaboration Diagram (Communication Diagram):

A Collaboration Diagram, also known as a Communication Diagram, focuses on the structural organization of objects and the messages they exchange. Key theoretical aspects include:

1. **Object (Entity):** Represents an instance of a class involved in the collaboration.
2. **Link (Association):** Represents relationships between objects, typically with navigability arrows.

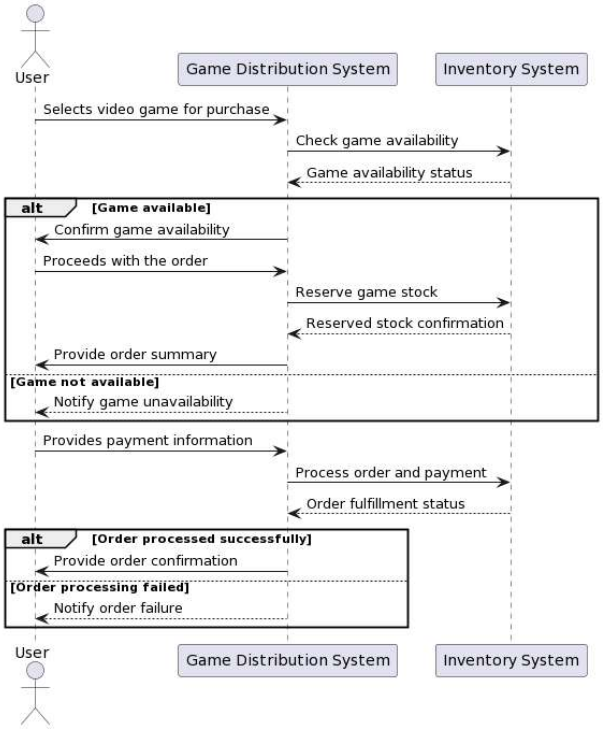
3. **Message:** Arrows indicating the flow of communication between objects.
4. **Multiplicity Notation:** Indicates the number of instances participating in an association or link.
5. **Role:** Describes the way an object participates in a particular collaboration.
6. **Self-Message:** Indicates a message sent by an object to itself.

Collaboration Diagrams are effective for depicting the relationships and interactions between objects in a more static manner, providing a complementary view to Sequence Diagrams.

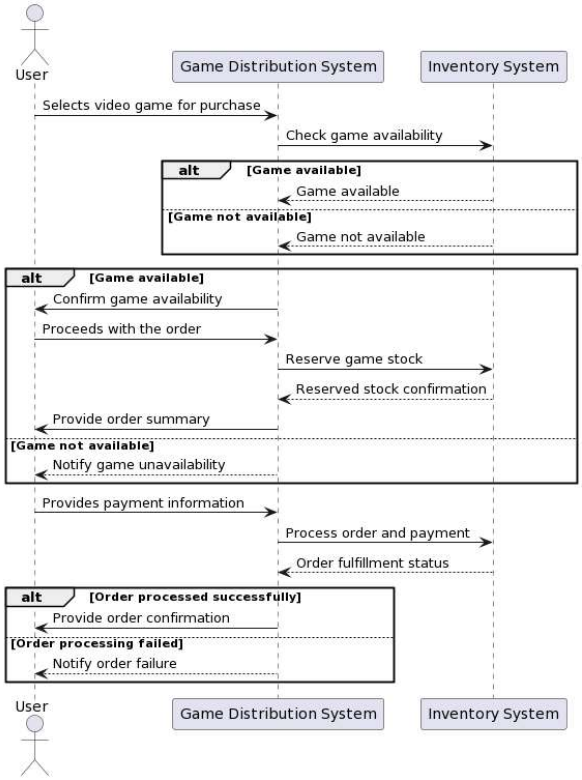
Relationship Between Sequence and Collaboration Diagrams:

- **Perspective:**
    - Sequence Diagrams emphasize the chronological order of interactions over time, while Collaboration Diagrams focus on the structural aspects of the system and the entities involved.
- **Complementary Views:**
    - Together, these diagrams provide a comprehensive understanding of both the dynamic behavior and the structural organization of a system.
- **Communication Clarity:**
    - Sequence Diagrams are particularly useful for illustrating the flow of messages during specific scenarios, while Collaboration Diagrams provide a clearer depiction of the relationships between objects.

By utilizing both Sequence and Collaboration Diagrams, software engineers can effectively model and communicate the dynamic aspects of a system, promoting a better understanding of its behavior and interactions.



Sequence Diagram



Collaboration Diagram

CONCLUSION

The sequence diagram and collaboration diagram are successfully created.

EXPERIMENT 10

AIM

To perform the environmental view diagram: Deployment diagram for the system.

SOFTWARE USED

Star UML

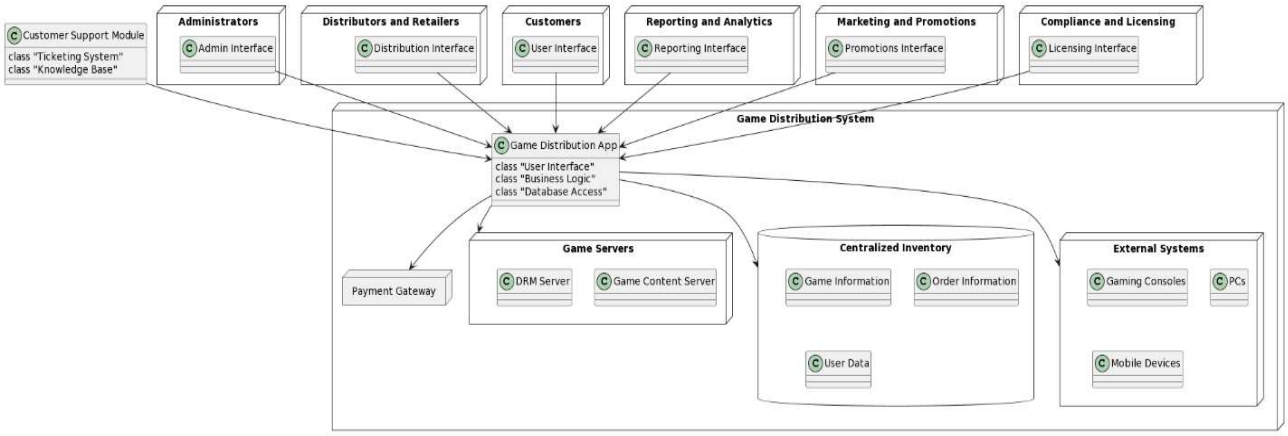
THEORY

A Deployment Diagram in software engineering is a type of UML diagram that models the physical deployment of software components on hardware nodes. It provides a visual representation of how software artifacts are distributed across the hardware components in a system. Here's some theoretical insight into the key concepts of a Deployment Diagram:

1. **Nodes:** Nodes represent hardware devices or software execution environments, such as servers, computers, or devices. Each node is depicted as a box.
2. **Components:** Components represent software modules or artifacts that are deployed on nodes. These could be executable files, libraries, or other software elements.
3. **Artifacts:** Artifacts are instances of physical data that are used or produced during the software development and deployment process. They can represent files, databases, or configuration files.
4. **Deployment Relationship:** Deployment relationships connect components to nodes, indicating which components are deployed on which nodes. These relationships are typically shown using arrows.
5. **Associations:** Associations between nodes represent communication paths or dependencies between hardware components. These can include network connections, communication channels, or any other relationships between nodes.
6. **Multiplicity Notation:** Multiplicity notation on deployment relationships indicates the number of instances of a component deployed on a node.
7. **Deployment Stereotypes:** Stereotypes such as "<<device>>" or "<<web server>>" can be used to provide additional information about the type of hardware or software element being deployed.
8. **Artifacts Stereotypes:** Artifacts may be stereotyped to represent different types of files, databases, or other physical data.
9. **Execution Environment:** An execution environment represents a node where components can be deployed and executed. It provides a context for understanding the physical deployment of software.

10. **Deployment Diagram Purpose:** The primary purpose of a Deployment Diagram is to illustrate how software components map to hardware nodes in a physical environment. It helps in understanding the distribution and configuration of the system's runtime elements.
11. **Deployment Unit:** A deployment unit is a package of related artifacts that are deployed together. It helps in organizing the deployment structure for better clarity.

Deployment Diagrams are particularly valuable for system architects and developers to visualize the physical aspects of a system, aiding in tasks such as system configuration, resource allocation, and understanding the deployment dependencies. They play a crucial role in ensuring that the software system can be effectively deployed and operated in its intended environment.



Deployment Diagram

CONCLUSION

The Deployment Diagram are successfully created.

EXPERIMENT 11

AIM

To perform various testing using the testing tool unit testing, integration testing for a sample code of the suggested system.

THEORY

Testing is a crucial aspect of software development to ensure the reliability, functionality, and performance of a system. Unit testing and integration testing are two fundamental types of testing that help developers identify and fix issues at different levels of the software.

Unit Testing:

1. **Definition:** Unit testing is a testing technique where individual components or functions of a software application are tested in isolation.
2. **Objective:** The primary goal of unit testing is to validate that each unit of the software performs as designed. A unit is the smallest testable part of an application, often a function or method.
3. **Isolation:** Unit tests are designed to be independent of each other, and each test should focus on a specific functionality or behavior of a unit.
4. **Automation:** Unit testing is often automated to allow for frequent and rapid testing during development. Testing tools such as JUnit (for Java), NUnit (for .NET), or pytest (for Python) are commonly used.
5. **Mocking:** In unit testing, it is common to use mock objects or stubs to simulate the behavior of components that a unit depends on. This helps isolate the unit being tested.
6. **Assertions:** Unit tests typically involve assertions to check whether the actual output of a unit matches the expected result. If there is a mismatch, the test fails.
7. **Test Cases:** Test cases in unit testing should cover normal and edge cases to ensure comprehensive coverage of the unit's behavior.

Integration Testing:

1. **Definition:** Integration testing is the process of testing the interaction between different components or systems to verify that they work together as expected.
2. **Objective:** The primary goal of integration testing is to expose faults in the interaction between integrated components. It ensures that the components can communicate, pass data, and collaborate effectively.

3. **Levels:** Integration testing can be performed at different levels, such as module integration testing, system integration testing, and acceptance testing. It progressively validates the integration of components.
4. **Top-Down and Bottom-Up Approaches:** In a top-down approach, testing starts from the highest level components, and in a bottom-up approach, testing starts from the lowest level components. A combination of both approaches, known as the sandwich or hybrid approach, is also common.
5. **Types of Integration Testing:** There are various types of integration testing, including Big Bang, Incremental, Top-Down, and Bottom-Up integration testing. The choice depends on the development process and project requirements.
6. **Stubs and Drivers:** During integration testing, stubs (for lower-level components) and drivers (for higher-level components) are often used to simulate the behavior of the components that are not yet integrated.
7. **Data Flow and Interface Testing:** Integration testing includes testing data flow between integrated components and ensuring that interfaces are working correctly.

Testing Tools:

1. **JUnit, NUnit, pytest:** These are popular unit testing frameworks for Java, .NET, and Python, respectively. They provide a structured way to write and execute unit tests.
2. **Selenium, JUnit, TestNG:** These tools are commonly used for integration testing in web applications. Selenium is used for browser automation, while JUnit and TestNG are used for managing and executing test cases.
3. **Mockito, PowerMock (for Java):** These are mocking frameworks that help create mock objects for unit testing to isolate the units being tested.
4. **Postman, SoapUI:** These tools are commonly used for API testing and can be part of both unit testing and integration testing processes.

In practice, the testing process involves writing test cases, executing them, and analyzing the results to ensure that the software meets its specified requirements and behaves as expected. Both unit testing and integration testing play vital roles in achieving this goal.

CONCLUSION

The Unit Testing and Integration Testing are successfully done.

EXPERIMENT 12

AIM

Perform Estimation of effort using FP Estimation for chosen system..

THEORY

Function Point (FP) estimation is a software sizing technique used to estimate the effort and resources required for developing a software system. It is based on quantifying the functionality provided by the system from the user's perspective. The estimation process involves several steps:

1. Identify and Categorize Functions: Identify and categorize the various functions that the system will perform. Functions are classified into External Inputs (EI), External Outputs (EO), External Inquiries (EQ), Internal Logical Files (ILF), and External Interface Files (EIF).

2. Determine Function Point Values: Assign function point values to each function based on complexity and characteristics. Function points are assigned based on guidelines provided in Function Point Analysis (FPA) standards.

3. Assign Complexity Weights: Assess the complexity of each function based on the number of data elements, transactions, and files involved. Assign complexity weights (Low, Average, High) for each function.

4. Calculate Unadjusted Function Points (UFP): Sum the function point values for all identified functions to obtain the Unadjusted Function Points. The formula for UFP is generally: UFP=EI×CFEI+EO×CFEO+EQ×CFEQ+ILF×CFILF+EIF×CFEIFUFP=EI×CFEI+EO×CFEO +EQ×CFEQ+ILF×CFILF+EIF×CFEIF

Where CFEI,CFEO,CFEQ,CFILF,CFEI,CFEO,CFEQ,CFILF, and CFEIFCFEIF are complexity factors for each function type.

5. Calculate Technical Complexity Factor (TCF): Assess the technical complexity of the software by considering 14 general system characteristics, such as distributed data processing, performance, and complexity of installation. Assign weights to each characteristic and calculate TCF. TCF=0.01×∑i=114WiTCF=0.01×∑i=114Wi

Where WiWi is the weight assigned to each characteristic.

6. Calculate Environmental Complexity Factor (ECF): Assess the environmental complexity by considering 15 factors like the experience of the team, use of modern programming practices, and the stability of the requirements. Assign weights to each factor and calculate ECF. ECF=0.01×∑i=115WiECF=0.01×∑i=115Wi

Where WiWi is the weight assigned to each factor.

7. Calculate Adjusted Function Points (AFP): Multiply UFP by TCF and ECF to get the Adjusted Function Points (AFP). AFP=UFP×TCF×ECFAFP=UFP×TCF×ECF

8. Convert to Person-Month Effort: Use a productivity factor or conversion factor to convert Adjusted Function Points into Person-Month effort. The productivity factor is specific to the organization and the team's historical performance.

9. Consider Project-Specific Factors: Adjust the calculated effort based on project-specific factors, such as team experience, complexity, and other organizational factors.

10. Review and Refine: Review the estimation with the project team and stakeholders. Refine the estimates based on feedback, changes in requirements, and any additional information.

11. Documentation and Monitoring: Document the estimation process, assumptions made, and factors considered. Monitor the actual effort expended during the project and use feedback to improve future estimations.

FP estimation is a comprehensive approach that helps in early project planning, resource allocation, and risk management. It provides a structured way to estimate the effort required for a software project based on the perceived functionality and complexity of the system.

CONCLUSION

The FP Estimation is successfully done.

EXPERIMENT 13

AIM

To prepare time Line Chart / Gantt Chart / PERT Chart for selected software project.

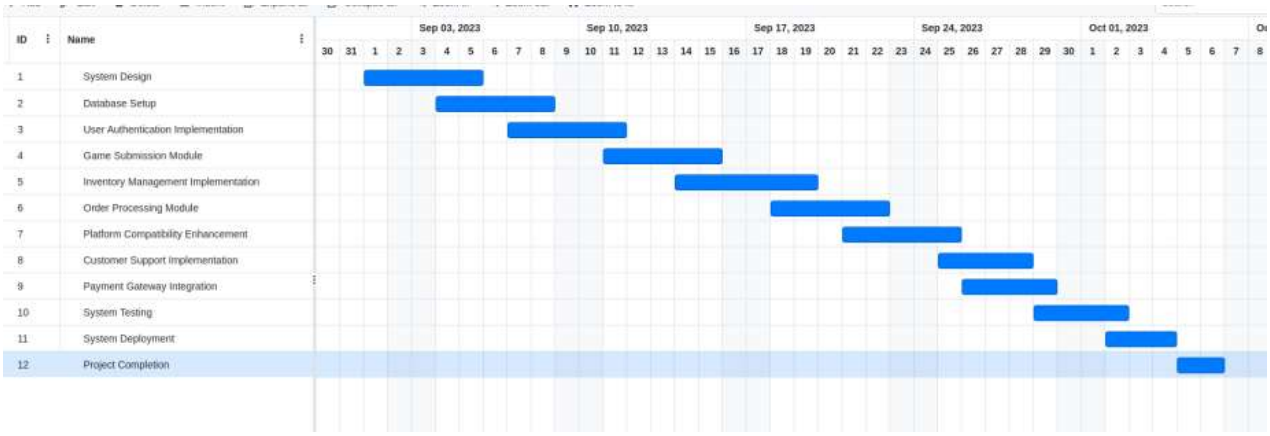
THEORY

Gantt Chart:

A Gantt chart is a horizontal bar chart that visually represents a project schedule. It shows tasks or activities along the y-axis and time along the x-axis. Key components include tasks/activities represented as horizontal bars, a timeline along the x-axis, and dependencies indicated by arrows connecting related tasks. Gantt charts provide clarity, help identify dependencies, and aid in resource allocation. They are widely used for project planning, scheduling, and monitoring progress in both short-term and long-term projects.

PERT Chart (Program Evaluation and Review Technique):

A PERT chart is a network diagram that represents a project schedule, focusing on depicting the relationships and dependencies between tasks. Key components include nodes that represent tasks or events and arrows that represent the dependencies between these tasks. PERT charts are useful for understanding the flow and dependencies of tasks in a project, making them valuable for project planning and scheduling.



CONCLUSION

The FP Estimation is successfully done.