SOFTWARE ENGINEERING PROJECT

Team:
Keisi Breshanaj
Ifigjenia Sopiqoti
Dejvi Kocilja
Loard Bejko

Lecturer: Aida Bitri
Assistant Lecturer: Enesh Orazova

TABLE OF CONTENTS

● CHAPTER 1

Project Details

Project Title: Supermarket ERP System

Problem Statement:
Managing a supermarket involves various complex tasks, such as inventory management, sales transactions, and staff supervision. The existing manual systems can be time-consuming and error-prone, leading to inefficiencies in operations. There is a need for an integrated Supermarket ERP software to streamline and enhance the overall management of the supermarket. This software will centralize all the supermarket's data and processes into one system, making the daily operations of its users easier and more efficient.

Solution Proposed:
We propose the development of EasyMart, a comprehensive Supermarket ERP software that caters to the specific needs of cashiers, managers, and administrators. EasyMart will automate various processes, providing a user-friendly interface for efficient supermarket management.

Project Scope:

- Aim: The aim of our project is to create a user-friendly Supermarket ERP software that simplifies and enhances overall supermarket management.
- Main Objectives:

Develop a secure login system for Cashiers, Managers, and Administrators.
Implement modules for sales transactions, inventory management, and staff supervision.
Create a loyalty points system to encourage customer retention.
Ensure data security and integrity in the storage and retrieval of information.

Application Description:
EasyMart is envisioned as a comprehensive Supermarket ERP software designed to simplify and enhance overall supermarket management. The software is crafted with a user-centric approach, ensuring a seamless and efficient experience for Cashiers, Managers, and Administrators.

Seamless User Experience: EasyMart will boast an easy-to-navigate and user-friendly interface, ensuring that each user—be it a cashier, manager, or administrator—can effortlessly carry out their tasks. The design prioritizes simplicity without compromising the depth of functionality.

Integration of Modules: The software integrates distinct modules tailored to the unique responsibilities of each user role. The Cashier Module focuses on swift and accurate transaction processing, making customer interactions smoother. The Manager Module provides comprehensive tools for inventory management, sales monitoring, and staff oversight. Simultaneously, the Administrator Module empowers administrators to efficiently manage user accounts and access permissions.

Loyalty Points System: Recognizing the significance of customer loyalty in the retail landscape, EasyMart implements a loyalty points system. This feature rewards customers based on their purchase history, providing incentives and discounts to foster a sense of appreciation and loyalty.

Scalability and Adaptability: EasyMart is not just a solution for the present; it's designed with an eye on the future. The architecture of the software is scalable, accommodating the potential growth of the supermarket. Additionally, we ensure that the software is adaptable to emerging technologies and industry best practices, future-proofing the investment made by the supermarket.

Roles and Tasks Distribution

Team Leader (Dejvi Kocilja):

- Coordinate team members and ensure project milestones are met.
- Selection and design of a suitable development model for EasyMart.
- Development of the interfaces for each user of the system and assist on the development of each module.

Main Roles and Tasks:

- Loard Bejko

Develop a maintenance plan and software evolution strategy.
Implement the cashier module functionalities, focusing on sales transactions and customer interactions.

- Ifigjenia Sopiqoti

Facilitate the submission of user requirements and application specifications.
Develop the manager module with a focus on inventory management and sales monitoring.

- Keisi Breshanaj

Work in the description of the software design and modeling.
Implement the administrator module functionalities, emphasizing user management and system logs.

- CHAPTER 2

Development Model:
Developing a Supermarket ERP System requires a systematic and structured development process. The EasyMart Supermarket ERP System is a project with well predefined requirements, which is why we decided to implement the Waterfall development model for our system. The waterfall model allows us to divide our work into distinct phases and provide us with a clear structure of our project.

The Waterfall development model consists of five phases: Requirements analysis and definition, System and Software design, Implementation and Unit testing, Integration and system testing, Operation and maintenance.

- Requirements analysis and definition: In this phase, the team will gather and define requirements and user expectations which includes managers, cashiers and administrators. We will analyze user requirements and also (if applicable) previous systems to understand user requirements completely. In this phase the team will also establish technical specifications such as database requirements, frameworks, designs etc. By the end of phase one we should have obtained a detailed documentation of functional and non-functional requirements for our ERP system.
- System and Software Design: Our next phase consists of creating the system architecture and design based on the gathered user requirements. The development team will design documents such as UML diagrams, ER diagrams and interface mockups and prototypes. The design phase will provide a complete overall system architecture, user interfaces etc. We will gather feedback in order to make modifications if necessary, before jumping on to the next phase.
- Implementation and Unit testing: In this phase, the team will start coding based on specific design specifications for different modules such as sales, inventory etc. to build the EasyMart ERP system. We will make sure that integration between modules will be completed for seamless functionality. Security measures and error handling mechanisms will be integrated and implemented. By the end of the implementation phase we will test individual code units to verify functionality and behavior. We will closely monitor and record test results in order to investigate any error that might occur and fix any failure.
- Integration and system testing: For this next phase the team will test how modules interact with each other and how seamless their integration and functionality is in order to ensure proper data flow across the system. We will also be testing the system as a whole and evaluate how it meets specific requirements as expected in a real-world environment.
- Operation and maintenance: As per the last phase, the team will deploy the ERP system into production. This phase consists of user training, system monitoring and we will also

be involved in the ongoing support and maintenance of the system. Maintenance involves bug fixes and making sure that the system runs smoothly and meets all user requirements.

The Waterfall development model is a linear approach to system development. Each phase of the project must be completed before ongoing to the next one. This allows us to concentrate and dedicate our energy completely in each phase of the project development.

User Requirements:
The EasyMart software is crafted with a user-centric approach, ensuring a seamless and efficient experience for different users be it cashiers, managers, or administrators. Here are some user requirements:

- Cashier requirements:

Secure login and logout functionality
Swift and accurate transactions processing
Ability to scan product barcodes and translate them into valuable product information
The possibility to handle every payment method whether it is a card or cash payment
Printing receipts
Being able to check loyalty points for each customer
Performing end of day sales reports

- Manager requirements:

Secure login/logout with role based authentication
Inventory management tools that allow the management to add/remove products in real time, tracking stock etc.,
Sales management features that allows the management to monitor and analyze sales
Staff oversight features that allow scheduling, task assignment etc.,
Management tools for the loyalty points program

- Administrator requirements:

Manage user accounts and access permissions
System configuration settings such as tax rates, currency rates, any applicable discount
Ability to track system changes and user functions
Database management procedures
Security features that include data encryption and user authentication protocols
Maintenance tools

The specified user requirements, which are our first phase of the project, will guide the software development for our EasyMart Supermarket ERP system. These build the groundwork for an ERP system that ensures to meet every specific need of each user.

Functional Requirements

Brief description:

- User-friendly design: Our system is not hard to operate. Cashiers, managers and administrators can use it simply without needing extra training.
- Distinct Modules: Our software combines several modules, where each one focuses on specific tasks for different roles in the supermarket.
- Loyalty Points System: This system is made for rewarding and promoting consumers by their past purchases.

The ability to adapt: As the supermarket changes and grows, this software is built to adapt to these changes by implementing new features.

Acceptance Criteria

- User-friendly interface:

With just three clicks or less, you can access your necessary functionalities.
Buttons and menus are simple to use.

- Distinct Modules:

Module of Cashier: The cashier is able to handle sales transactions precisely without the system breaking down.
Module of the Manager: The system provides the necessary tools to manage inventory, by keeping track of sales and supervising employees.
Module of the Administrator: The administrator is able to manage all user accounts.

- Loyalty Points System:

Based on their past history of purchases, users can earn loyalty points.
The total amount of loyalty points is displayed correctly.
Smooth point redemption during sales transactions.

- The ability to adapt:

The capacity of the system to handle big transactions without experiencing a drop in performance.
It is simple to adapt changes while also interfering with regular supermarket operations.
Regular updates to develop new technologies needed for new industry standards.

- Managing Data:

A database will be kept up to date by the system.
Mechanisms for data recovery are in place so there is no data loss.

- Security:

Strong security measures to guard user data.
Different users have different access to features and information.

Non-functional Requirements

Brief Description:

Performance: The system should be able to handle multiple users simultaneously without slowing down.
Reliability: The system should work at all times and recover quickly from errors.
Security: Only authorized users should be able to access important information.
Flexibility: The system must be able to withstand growing loads and adapt to change.
Usability: All users should be able access the system easily.
Maintenance: To continue operation, the system must be simple to fix and update.

Acceptance Criteria:

- Performance:

Common tasks should respond in less than 2 seconds.
The system should support at least 100 users at once.

- Reliability:

99.9% uptime is expected from the system, with planned maintenance resulting in least amount of downtime.
If a breakdown occurs, the system needs to recover without deleting any data in 5 minutes.
Security:
Since there is sensitive information, data encryption methods should be applied.
Role-based permissions should be enforced by access control.

- Flexibility:

If transactions increase by 20%, the system should be able to continue its functionality.
The system should allow new modules to be added without impacting the existing modules.

- Usability:

For the system to be accessible, the user interface should support keyboard navigation and screen readers.
For new users, training should be less than 1 hour.

- Maintenance:

The code base should have clear comments.
Applying updates should be easy and not cause any disturbance.

Application Specifications

a. Architecture:

```
+---------------------+
| Presentation Layer  |
+---------------------+
           |
           v
+---------------------+
| Application Layer   |
+---------------------+
           |
           v
+---------------------+
| Business Logic Layer|
+---------------------+
           |
           v
+---------------------+
| Data Access Layer   |
+---------------------+
           |
           v
+---------------------+
|      Database       |
+---------------------+
```

- Presentation Layer:

This layer is responsible for presenting information to the users and gathering input from them. It includes the user interface components developed using JavaFX, such as screens, forms, buttons, and menus.
The presentation layer interacts with the application layer to retrieve data and trigger actions based on user input.

- Application Layer:

The application layer serves as an intermediary between the presentation layer and the business logic layer.

It contains the logic for processing user requests, handling business rules, and coordinating interactions between different components of the system.
This layer interacts with the presentation layer to receive user input and with the business logic layer to execute business processes.

- Business Logic Layer:

This layer encapsulates the core business logic and rules of the supermarket app.
It includes functionalities such as client management, billing, product inventory management, user authentication, and authorization.
The business logic layer interacts with the data access layer to retrieve and manipulate data from the database.

- Data Access Layer:

The data access layer is responsible for accessing and manipulating data stored in the database.
It includes components such as data access objects (DAOs) or repositories that abstract the underlying database operations.
This layer interacts directly with the database to perform CRUD (Create, Read, Update, Delete) operations on data entities.

- Database:

The database stores persistent data related to clients, bills, products, users, and other entities.
It may utilize a file-based storage system as indicated by the ".ser" files mentioned earlier, or it could be a relational database management system (RDBMS) such as MySQL, PostgreSQL, or SQLite.

Database Model:

Clients.ser: This file stores information about clients or customers who interact with the supermarket app. It likely includes details such as client ID, name, contact information, and any other relevant data.

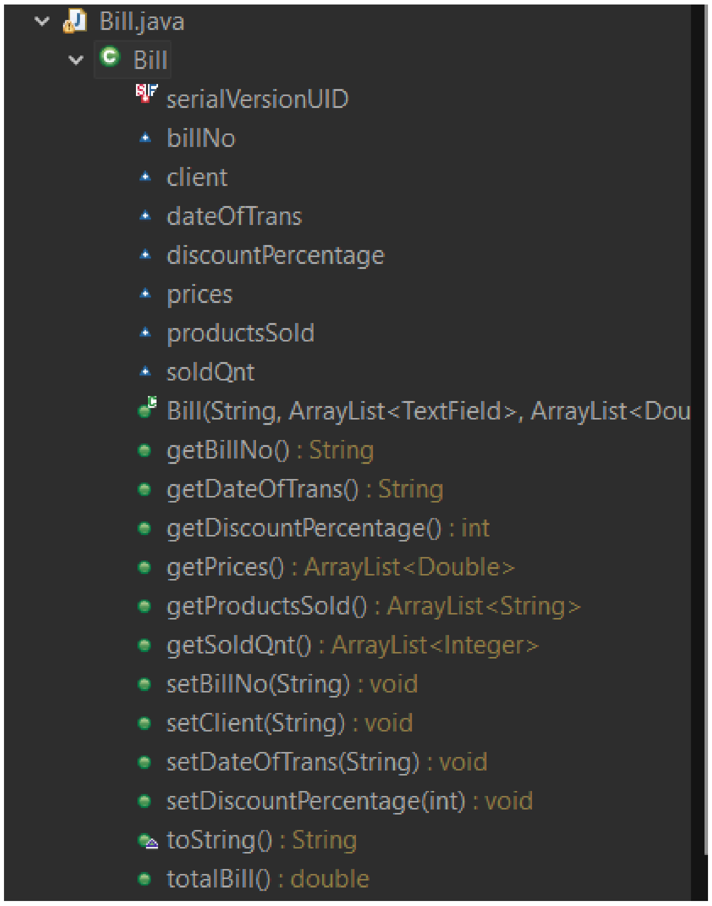
Bills.ser: The "Bills" file contains records of transactions made by clients, including details such as bill ID, date and time of purchase, items purchased, quantities, prices, and total amount.

Products.ser: This file serves as a repository for product information available in the supermarket. It likely includes details such as product ID, name, description, price, stock quantity, and any other attributes related to the products offered for sale.

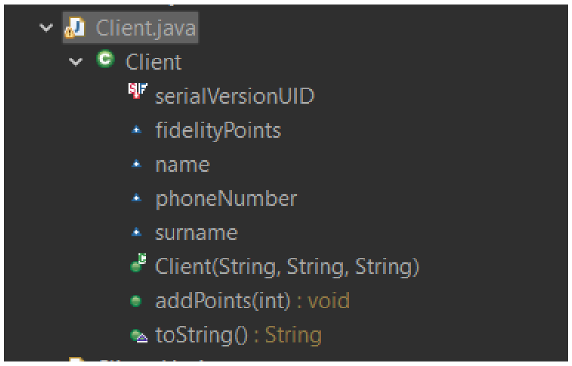
Users.ser: The "Users" file is responsible for storing information about users who have access to the supermarket app. This may include user IDs, usernames, hashed passwords, roles or permissions, and any other relevant user-related data.

Constraints and relationships between these files may vary depending on the specific requirements of the application. For example:

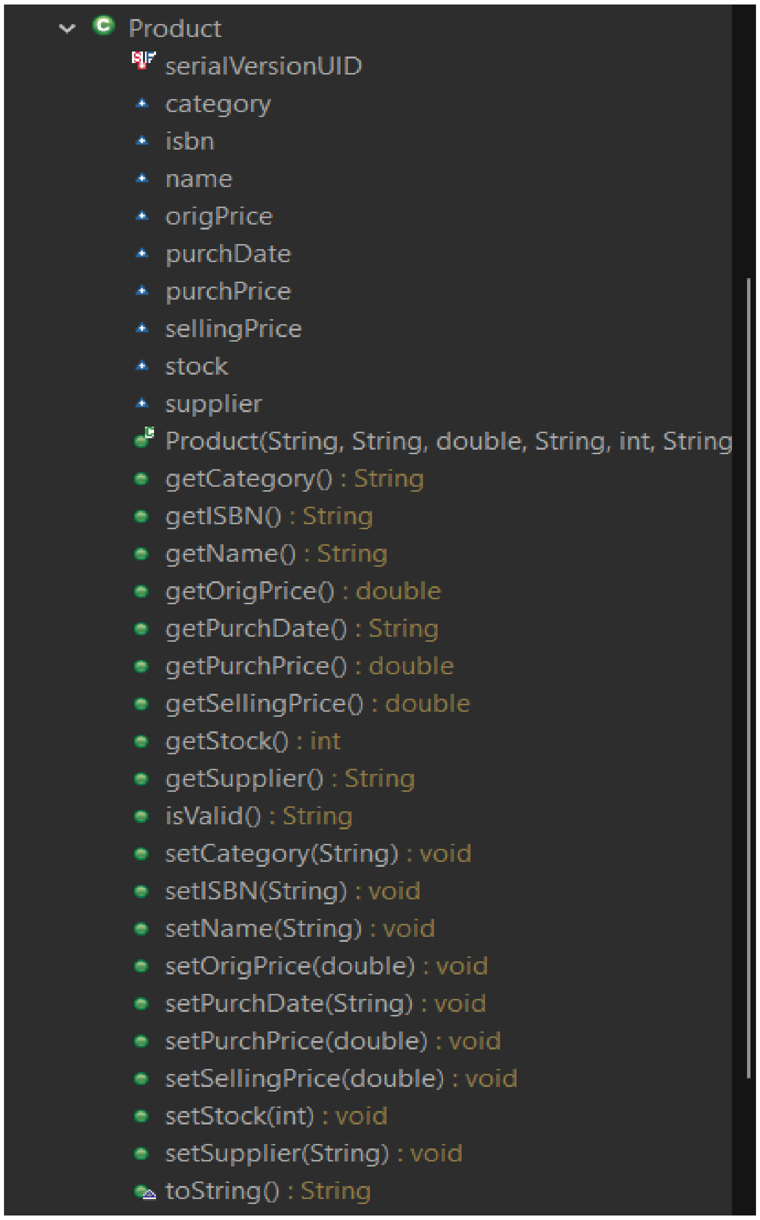
The "Clients" file may have a one-to-many relationship with the "Bills" file, indicating that each client can have multiple bills associated with them.
The "Bills" file may have a many-to-many relationship with the "Products" file, indicating that each bill can contain multiple products, and each product can be included in multiple bills.
The "Users" file may have constraints such as unique usernames to ensure each user has a distinct login identifier.



```
Bill.java
  Bill
    serialVersionUID
    billNo
    client
    dateOfTrans
    discountPercentage
    prices
    productsSold
    soldQnt
    Bill(String, ArrayList<TextField>, ArrayList<Dou
    getBillNo() : String
    getDateOfTrans() : String
    getDiscountPercentage() : int
    getPrices() : ArrayList<Double>
    getProductsSold() : ArrayList<String>
    getSoldQnt() : ArrayList<Integer>
    setBillNo(String) : void
    setClient(String) : void
    setDateOfTrans(String) : void
    setDiscountPercentage(int) : void
    toString() : String
    totalBill() : double
```



```
Client.java
  Client
    serialVersionUID
    fidelityPoints
    name
    phoneNumber
    surname
    Client(String, String, String)
    addPoints(int) : void
    toString() : String
```



```
Product
  serialVersionUID
  category
  isbn
  name
  origPrice
  purchDate
  purchPrice
  sellingPrice
  stock
  supplier
  Product(String, String, double, String, int, String
  getCategory() : String
  getISBN() : String
  getName() : String
  getOrigPrice() : double
  getPurchDate() : String
  getPurchPrice() : double
  getSellingPrice() : double
  getStock() : int
  getSupplier() : String
  isValid() : String
  setCategory(String) : void
  setISBN(String) : void
  setName(String) : void
  setOrigPrice(double) : void
  setPurchDate(String) : void
  setPurchPrice(double) : void
  setSellingPrice(double) : void
  setStock(int) : void
  setSupplier(String) : void
  toString() : String
```

- Technologies Used:

For this project, we chose to use Java as the programming language because it's widely supported and known for its reliability. For the front-end design, we turned to JavaFX, a tool that helps us create user-friendly interfaces with buttons, menus, and other interactive elements. By using Java and JavaFX together, we're able to build a solid and visually appealing application that meets our needs.

Cross-platform compatibility: Java is known for its platform independence, allowing the app to run on various operating systems without requiring major modifications.

Rich user interface: JavaFX provides a robust set of tools for creating visually appealing and interactive user interfaces.

Scalability: JavaFX is well-suited for building scalable applications, including enterprise-level solutions like a supermarket management system.

Integration with backend systems: JavaFX seamlessly integrates with Java backend technologies, such as Spring Boot or Java EE, enabling smooth communication between the front-end and backend components of the application.

d. User Interface Design:
Showcase wireframes, mockups, or describe the user interface:
Provide a visual representation of how users will interact with the system.

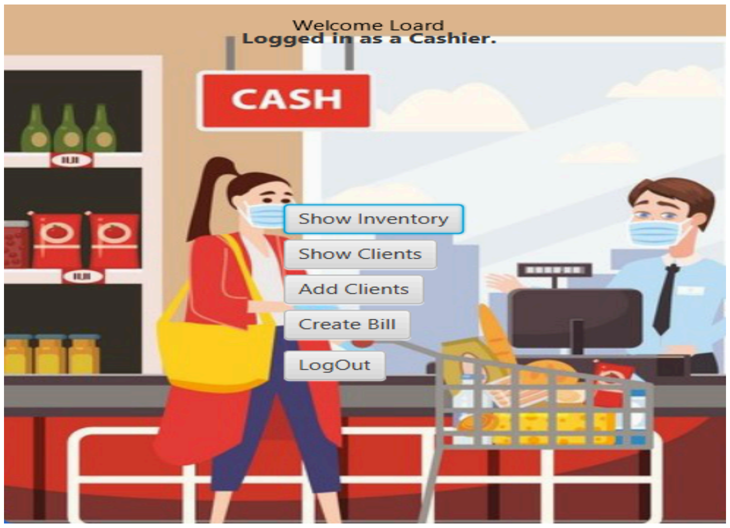
e. Security Measures:

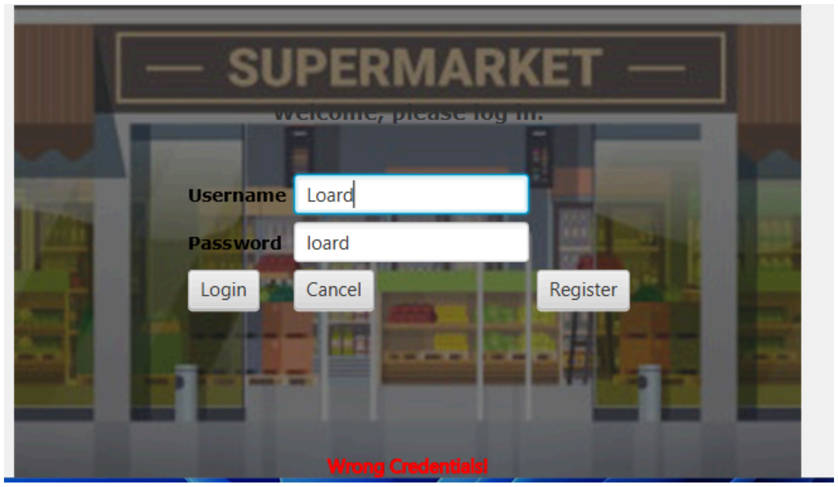
1.Authentication: Implementing authentication with username and password is a fundamental security measure to ensure that only authorized users can access the supermarket app. This helps prevent unauthorized access to sensitive data and functionalities.

2. Password Hashing: Hashing passwords is a critical step in protecting user credentials. By hashing passwords, you ensure that even if the database is compromised, attackers cannot directly access the plaintext passwords. Instead, they would need to perform a time-consuming brute-force attack to reverse-engineer the original passwords from the hashes.

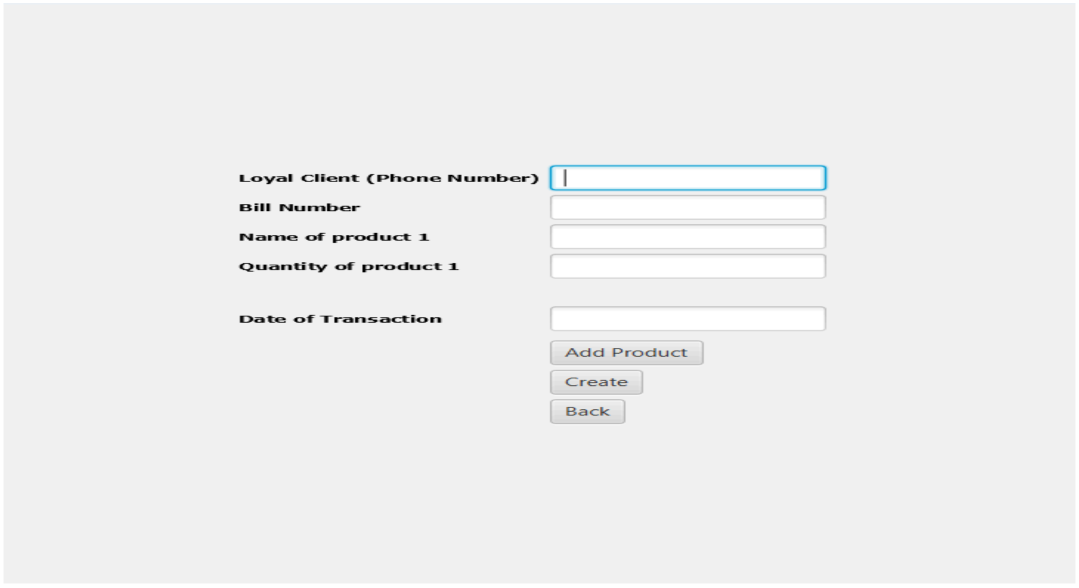
3. Input Validation: Validating user input is essential for preventing common vulnerabilities such as SQL injection, cross-site scripting (XSS), and command injection. By validating and sanitizing input data, you reduce the risk of malicious exploitation of your application.

- Here are some visualizations on how the users interact with the system:





Username Loard
Password loard
Wrong Credentials!



Loyal Client (Phone Number)
Bill Number
Name of product 1
Quantity of product 1
Date of Transaction

- CHAPTER 3

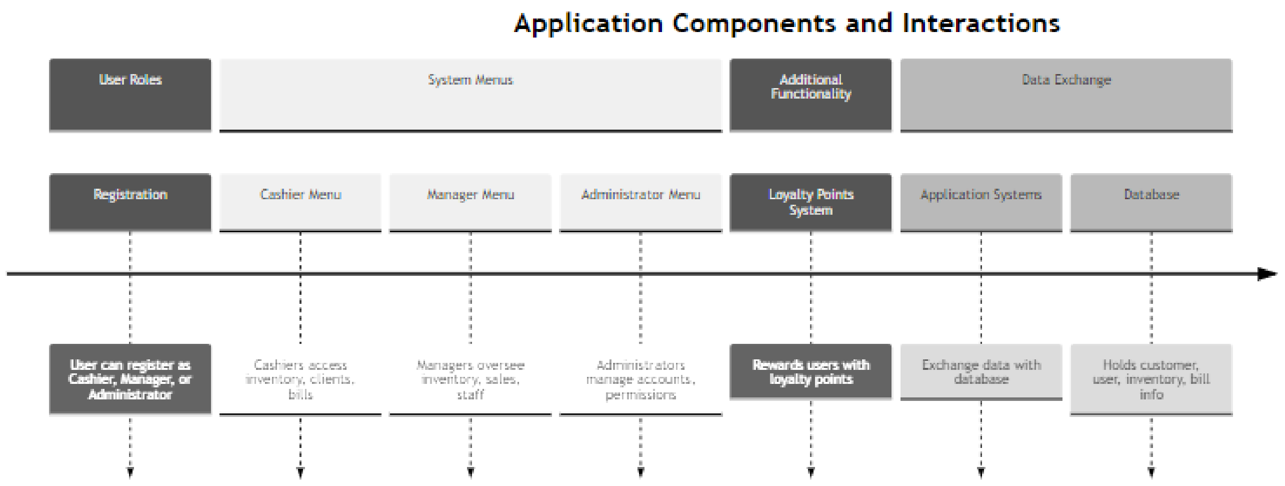
Software Architecture

System Architecture:
As mentioned in the previous phases of this project, EasyMart software has 3 fundamental roles that the Cashier, Manager, and Administrator Modules perform. The Cashier module handles sales transactions and customer interactions; the Manager module is in charge of general operations and inventory control; and the Administrator module is in charge of user accounts and permissions. Depending on the type of user role that was selected during the registration process, each type of user will have their own distinct menus to perform different operations. These modules work well together, utilizing real-time database data to facilitate seamless collaboration and effective decision-making.

On top of that, a loyalty program is in place to reward the most loyal customers of the supermarket with points which can be converted into discounts on future purchases.The system tracks customer purchases and dynamically updates customer loyalty points during checkout.Moreover, the Loyalty Points System interfaces with the database, storing and retrieving customer loyalty point balances with precision. This ensures accurate tracking and seamless redemption of points, enhancing the overall customer experience and solidifying the supermarket's relationship with its clientele.

A strong database system functions in tandem with these components, acting as the central repository for crucial data such as product details, sales histories, user accounts, billing information etc. By facilitating smooth communication between modules, this database makes sure that data is consistent and of high quality throughout the entire system. The EasyMart Supermarket ERP software optimizes user interaction and system functionality with a layered architecture that includes Presentation (Front-End), Application, Business Logic, and Data Access Layers. This gives supermarket managers and staff the ability to make well-informed decisions and provide outstanding customer service.

Component Diagram:


Application Components and Interactions
User Roles
System Menus
Additional Functionality
Data Exchange
Registration
Cashier Menu
Manager Menu
Administrator Menu
Loyalty Points System
Application Systems
Database
User can register as Cashier, Manager, or Administrator
Cashiers access inventory, clients, bills
Managers oversee inventory, sales, staff
Administrators manage accounts, permissions
Rewards users with loyalty points
Exchange data with database
Holds customer, user, inventory, bill info

Detailed design

Class diagram

EasyMart (main class)
cashiermodule: Cashier Module
managermodule: Manager Module
adminmodule: Administrator Module

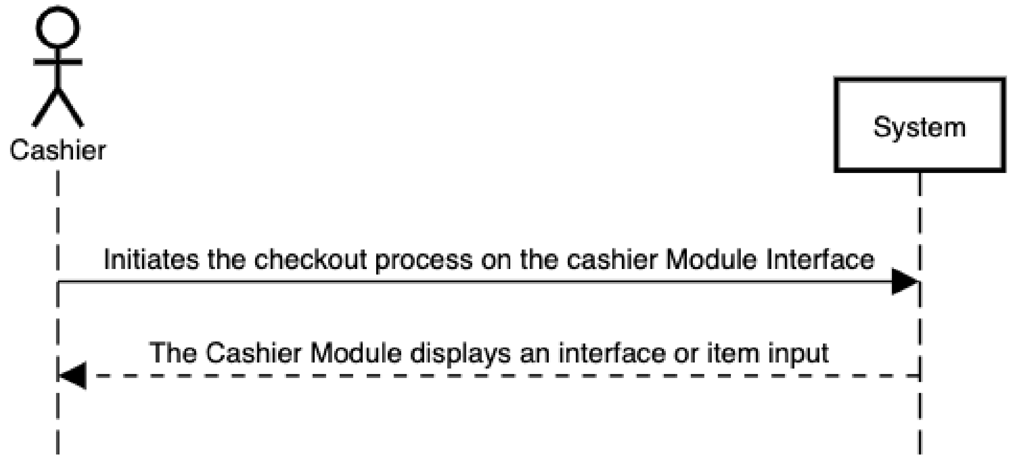
Cashier Module
processTransation()
applyLoyaltyPoints()

Manager Module
manageInventory()
monitorSales()
overseeStaff()

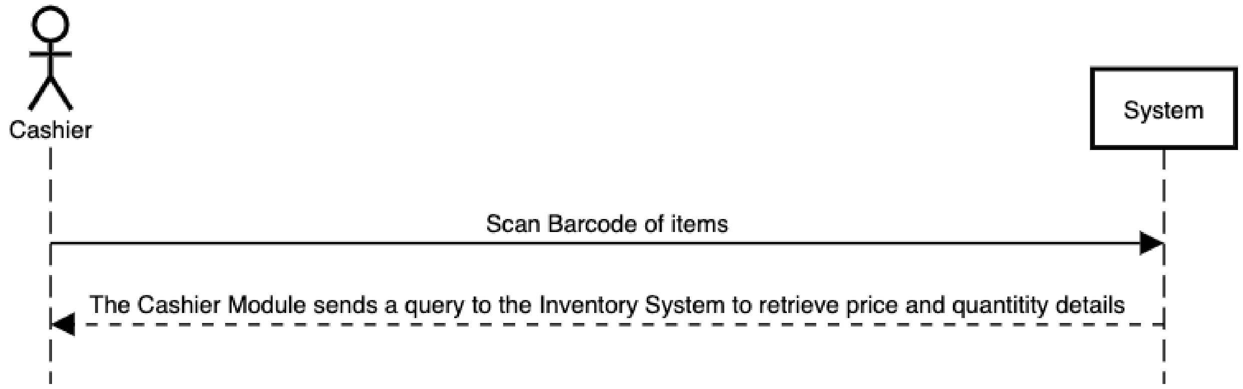
Administrator Module
manageUsers()
managePermissions()

Sequence Diagram

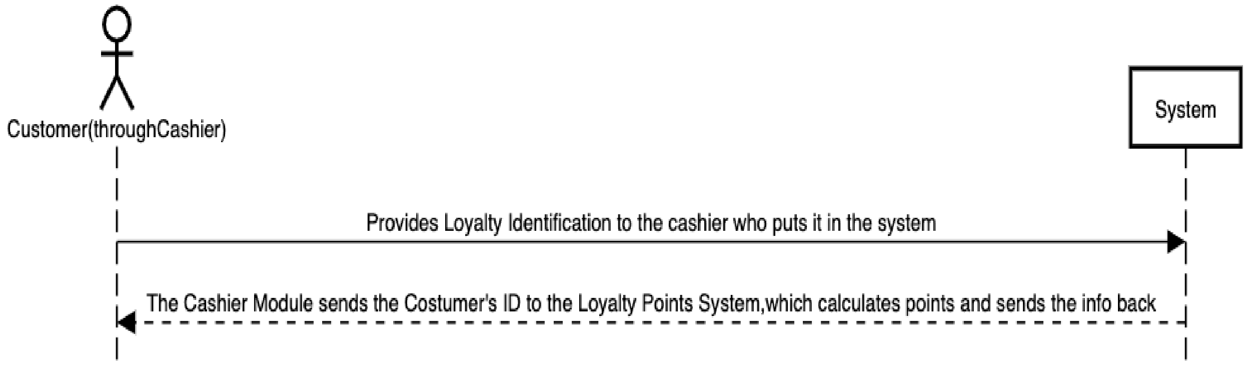
- Start Transaction


Cashier
System
Initiates the checkout process on the cashier Module Interface
The Cashier Module displays an interface or item input

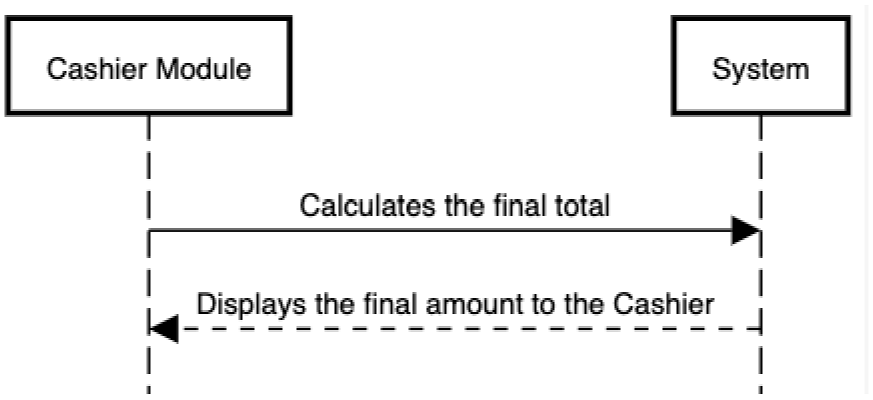
- Scan Items


Cashier
System
Scan Barcode of items
The Cashier Module sends a query to the Inventory System to retrieve price and quantitity details

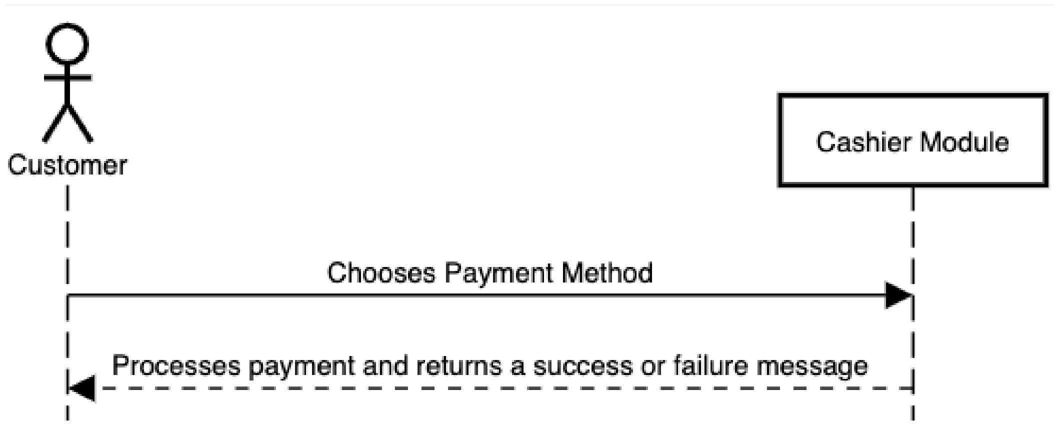
- Identify Costumer and apply Loyalty Points


Customer(throughCashier)
System
Provides Loyalty Identification to the cashier who puts it in the system
The Cashier Module sends the Costumer's ID to the Loyalty Points System,which calculates points and sends the info back

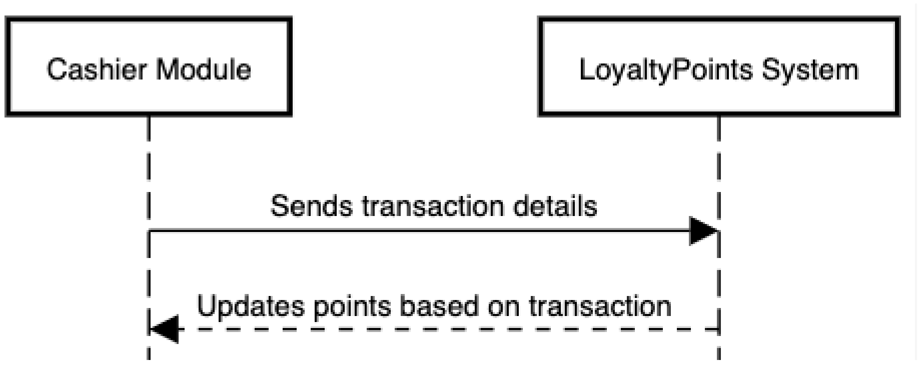
- Finalize Total


Cashier Module
System
Calculates the final total
Displays the final amount to the Cashier

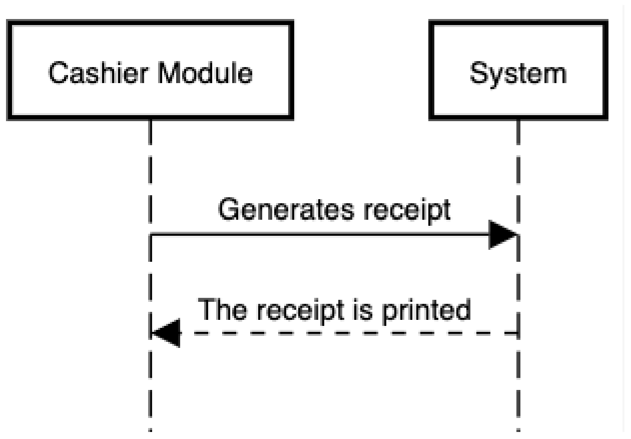
- Process Payment



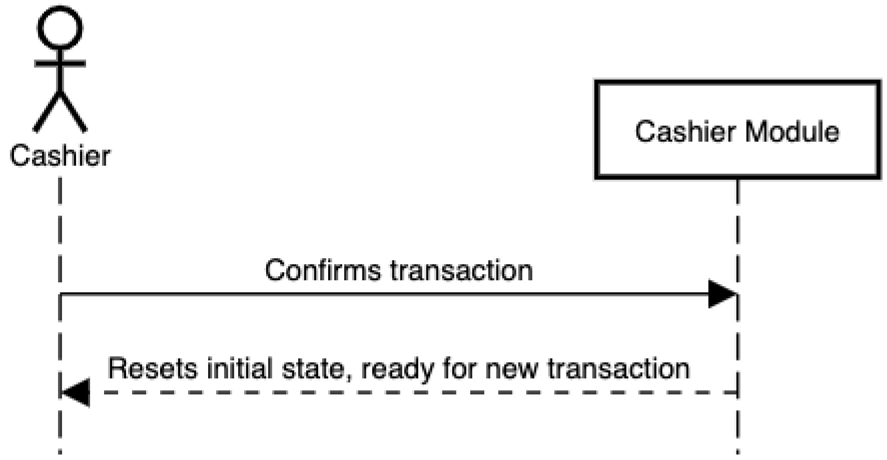
- Update Loyalty Points


Cashier Module
LoyaltyPoints System
Sends transaction details
Updates points based on transaction

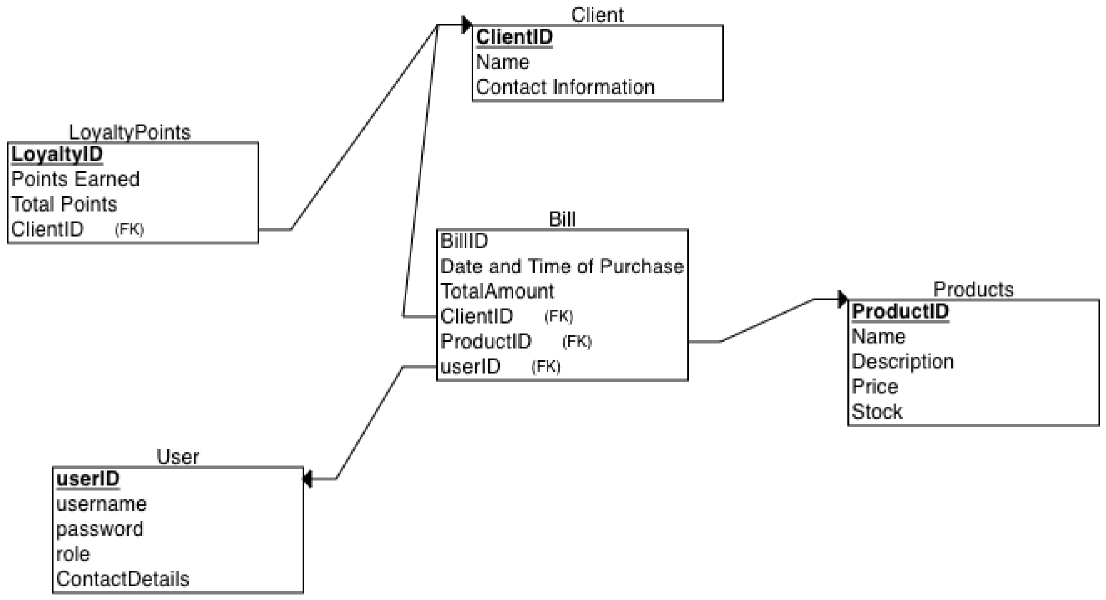
- Generate and Print Receipts



- End Transaction



Database design:



Clients to Bills: One to Many
Each client can have multiple bills, but one bill is linked to one client.

Bills to Products: Many to Many
Each bill has many products and each product can be in many different bills.

Users to Bills: One to Many
Each user can create multiple bills, but each bill in connected to only one user.

Client to Loyalty Points: One to One
Each client has their own loyalty points.

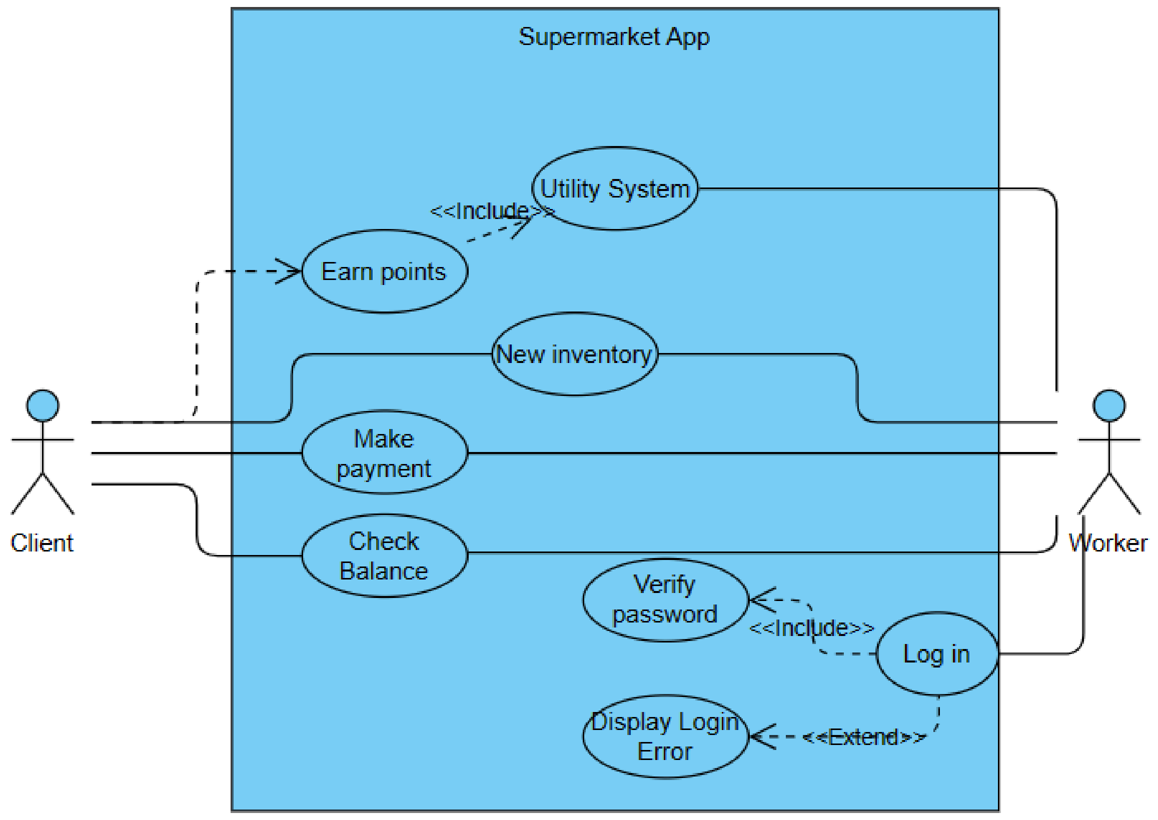
Modeling
Use Case Diagram:

Actors:
Workers
Cashier
Manager

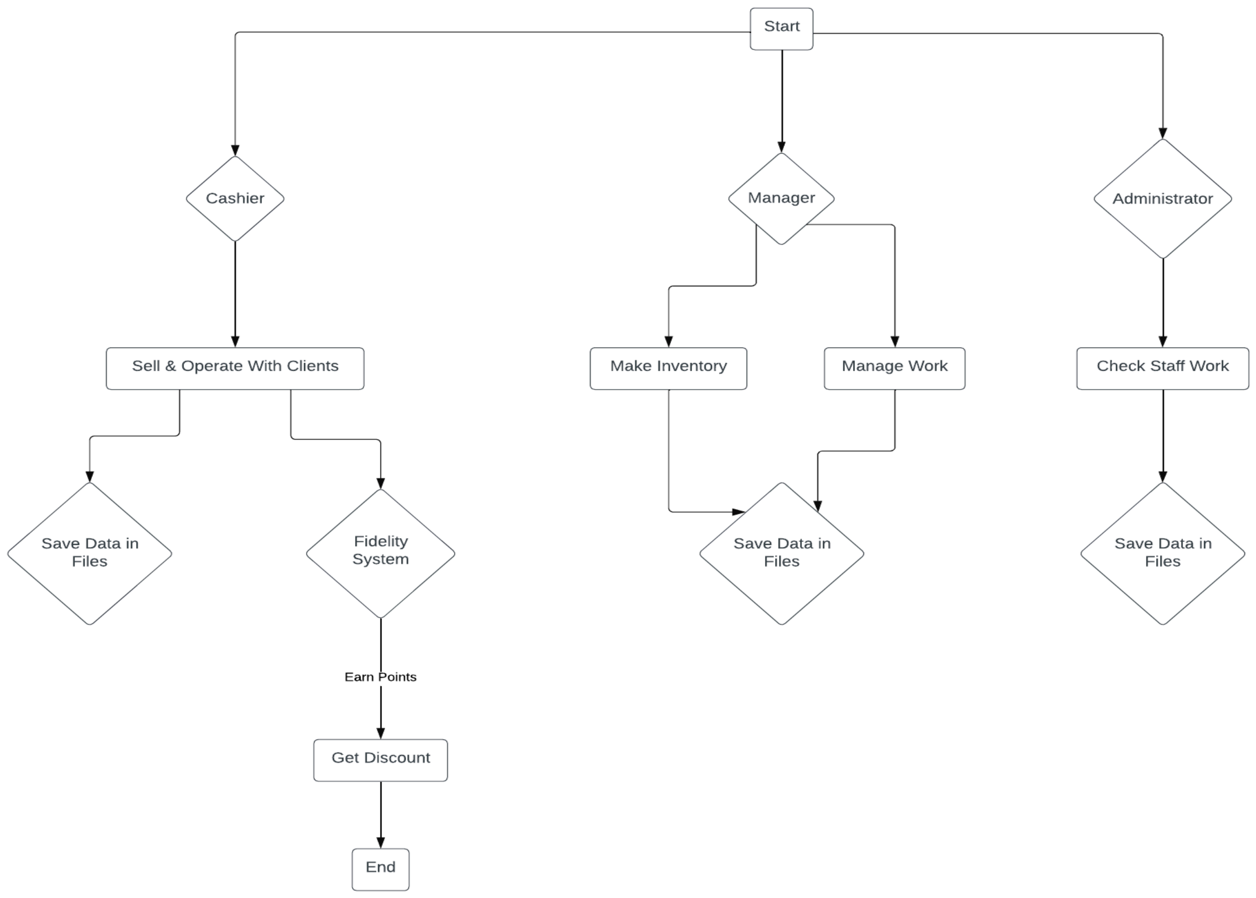
Administrator
Client

Use Cases:
Cashier: Sells products, Process payments, Manage client operations
Manager: Manage inventory, Manage staff work
Administrator: Monitor staff work, Access system logs

Use Case Diagram:

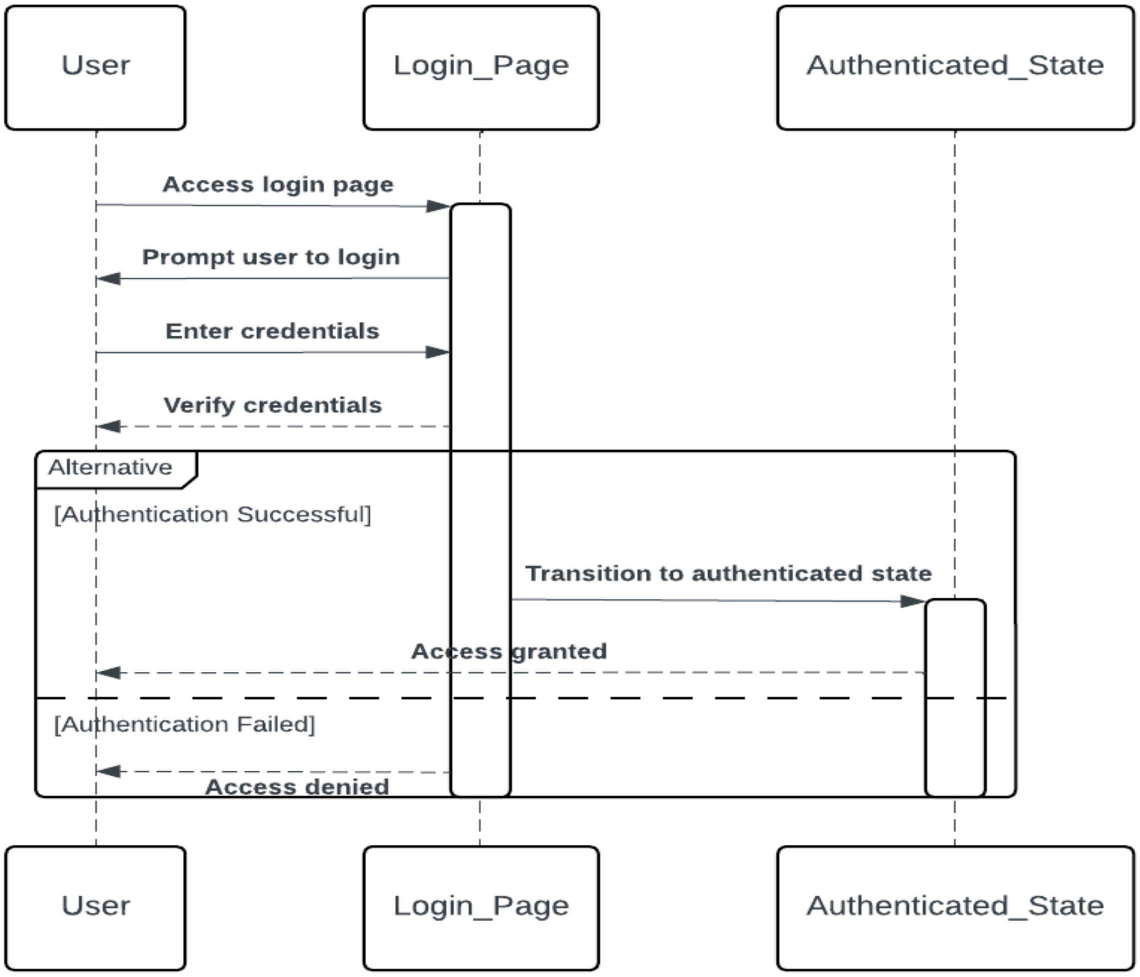


Activity Diagram :

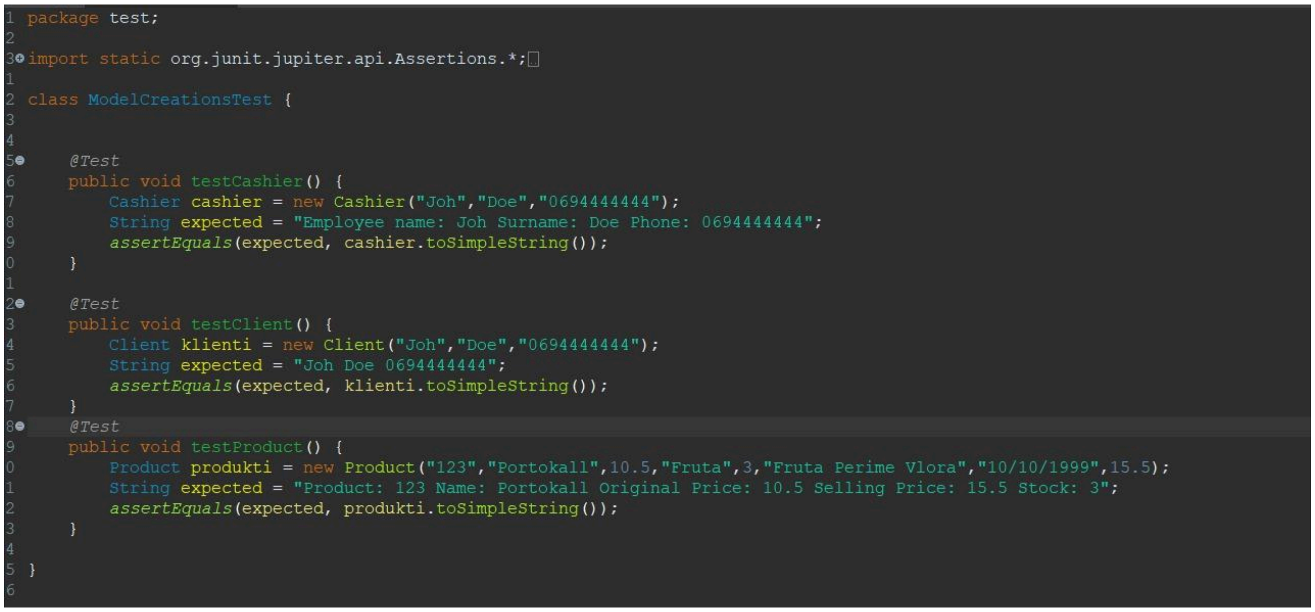


State Diagram:

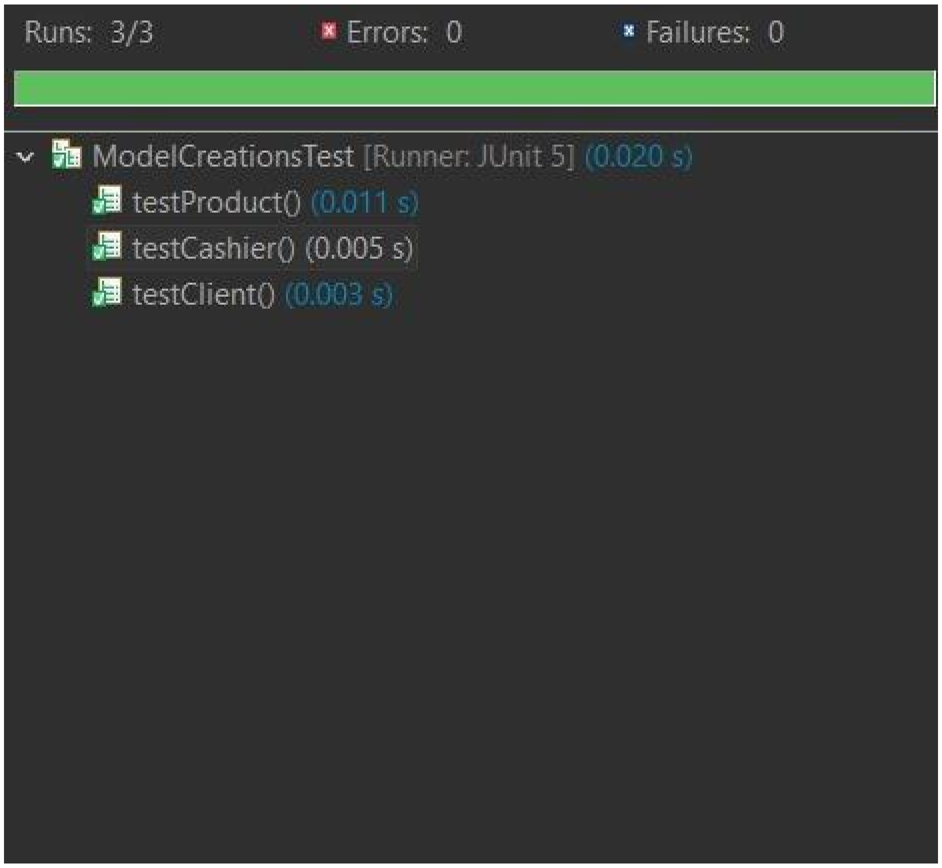
States:
Login State: Represents the initial state when a user attempts to log in.
Authenticated State: Represents the state when a user is successfully authenticated and logged into the system.



- CHAPTER 4



```
package test;

import static org.junit.jupiter.api.Assertions.*;

class ModelCreationsTest {


    @Test
    public void testCashier() {
        Cashier cashier = new Cashier("Joh","Doe","0694444444");
        String expected = "Employee name: Joh Surname: Doe Phone: 0694444444";
        assertEquals(expected, cashier.toSimpleString());
    }

    @Test
    public void testClient() {
        Client klienti = new Client("Joh","Doe","0694444444");
        String expected = "Joh Doe 0694444444";
        assertEquals(expected, klienti.toSimpleString());
    }
    @Test
    public void testProduct() {
        Product produkti = new Product("123","Portokall",10.5,"Fruta",3,"Fruta Perime Vlora","10/10/1999",15.5);
        String expected = "Product: 123 Name: Portokall Original Price: 10.5 Selling Price: 15.5 Stock: 3";
        assertEquals(expected, produkti.toSimpleString());
    }

}
```



```
Runs: 3/3        Errors: 0        Failures: 0

ModelCreationsTest [Runner: JUnit 5] (0.020 s)
    testProduct() (0.011 s)
    testCashier() (0.005 s)
    testClient() (0.003 s)
```

TEST CASE #1 (Dejvi Kocilja)

1. Introduction to Testing:
Software testing is a methodical procedure that is essential to assessing the Bill class and making sure it performs as planned. We check the code for flaws, faults, and defects by running the methods of the Bill class with different inputs and scenarios. Maintaining the accuracy, consistency, and caliber of the software application's billing functionality requires testing the Bill class.

2. Purpose of Testing:
Finding and fixing bugs early in the development cycle is the main goal of testing the Bill class. Early problem detection reduces the amount of money and time needed to repair problems. Furthermore, testing makes sure the Bill class works as intended—that is, that totals are calculated correctly, discounts are applied, and different billing circumstances are handled well. This eventually results in a better user experience by preserving the software's dependability and effectiveness.

3. Focus on Testing a Single Component:
Focusing on testing the Bill class allows us to thoroughly evaluate its role, complexity, and impact on the overall software system. Billing is a component with critical functionality, for which rigorous testing is necessary to ensure it is reliable and correct. Potential flaws or problems unique to billing computations and discount application can cause errors in calculating the profits or losses of the supermarket, which would in turn disrupt its overall activities. That is why it is of greatest importance to test the Bill class which handles the functions mentioned till now.

4. Preparing Test Cases:
Test cases for the Bill class should encompass a wide range of scenarios, including normal inputs (valid bill data), edge cases (empty bill, zero prices), and invalid inputs (non-numeric quantities, negative prices). Each test case should specify the inputs, actions, and expected outcomes, ensuring comprehensive coverage of the Bill class's functionality.

5. Choosing Testing Frameworks:
For testing the Bill class in Java, JUnit is a suitable testing framework. JUnit provides a structured approach to writing, organizing, and executing tests, offering features such as assertion libraries and test runners. By utilizing JUnit, we can efficiently set up and execute test cases for the Bill class, facilitating thorough testing of its methods and functionalities.

6. Writing Test Code:

Writing test code for the Bill class involves creating test methods to exercise different functionalities, such as calculating total bill amounts and applying discounts. In the piece of code below, I have first imported all the necessary libraries and created a function to initialize objects or perform necessary setup steps before executing individual test methods for the Bill class. Then function testTotalBillWithoutDiscount verifies that the total bill amount calculated by the totalBill method of the Bill class matches the expected value without applying any discount. testTotalBillWithDiscount: Checks whether the total bill amount calculated by the totalBill method of the Bill class, after applying a 10% discount for a specific client, matches the expected discounted value.

7. Running Tests:
Executing tests for the Bill class involves running the test code against its methods and analyzing the results. Tests may pass, fail, or encounter errors, indicating the correctness of the Bill class's implementation. By interpreting test results, we can identify any defects or deviations from expected behavior and take corrective actions as necessary.

8. Test Coverage:
In order to thoroughly test the Bill class and make sure that the test suite is appropriately exercising all of its methods and features, achieving high test coverage is essential. Enough test coverage makes it easier to spot untested code paths and guarantees that the Bill class operates correctly and consistently across a range of billing scenarios. A more stable and dependable application can be produced by having a high test coverage level, which can aid in the early detection and correction of any possible flaws or difficulties. In summary, aiming for high test coverage in the Bill class is critical to preserving the billing system's overall integrity and quality.

TEST CASE #2 (Ifigjenia Sopiqoti)

1. Introduction to Testing:
Software testing is a fundamental process within the software development lifecycle aimed at identifying defects or bugs in software applications. It involves systematically executing the software to ensure it behaves as expected and fulfills specified requirements. Testing plays a pivotal role in software development by ensuring the reliability, correctness, and overall quality of the software product. By rigorously testing software components, developers can identify and address issues early in the development process, ultimately reducing costs and time spent on rework.

2. Purpose of Testing:
The primary purpose of testing is to detect defects early in the development process, minimizing the cost and effort required for rectifying them. Additionally, testing verifies that software components, such as the Inventory class, perform as intended and meet user needs. By testing software components thoroughly, developers can ensure that the final product is robust, reliable, and meets specified requirements.

3. Focus on Testing a Single Component:
The Inventory class plays a crucial role in managing the inventory of products within a software system. Testing this component is essential to ensure accurate inventory management and proper handling of product data. By focusing on testing the Inventory class, developers can identify potential defects or errors specific to inventory management functionalities, thereby ensuring the reliability and functionality of the overall system.

4. Preparing Test Cases:
Test cases serve as detailed specifications for testing specific functionalities or scenarios of the Inventory class. These cases should cover a range of scenarios, including normal operations, edge cases such as empty inventory or missing products, and invalid inputs such as non-existent products or negative quantities. By preparing comprehensive test cases, developers can ensure thorough testing of the Inventory class and identify potential issues across various scenarios.

5. Choosing Testing Frameworks:
In Java development, JUnit stands as a suitable testing framework for writing and executing tests, organizing test cases, and reporting results. Setting up the testing environment with JUnit involves adding the JUnit dependency to the project's build configuration. By leveraging JUnit, developers can streamline the testing process and ensure effective testing of the Inventory class.

6. Writing Test Code:
Writing test methods for the Inventory class involves creating tests to cover different functionalities such as adding products, updating stock, and checking for missing products. These test methods should be well-structured, readable, and maintainable, utilizing assertions to validate expected outcomes against actual results. By writing effective test code, developers can ensure thorough testing and verify the correctness of the Inventory class's behavior.

8. Test Coverage:
High test coverage is essential to ensure thorough testing of the Inventory class, covering all its methods, edge cases, and interactions with product data. By achieving high test coverage, developers can identify areas of the code that are not adequately tested and ensure that critical

functionalities are covered by tests. Comprehensive test coverage contributes to the overall reliability and quality of the Inventory class and the software system as a whole.

TEST CASE #3 (Keisi Breshanaj)

1. Introduction to Testing:
Software testing is a pivotal process integral to the software development lifecycle, encompassing the systematic evaluation of software to pinpoint defects or bugs. It serves as a critical safeguard, ensuring the reliability and correctness of software systems by subjecting them to various scenarios and inputs. The importance of testing cannot be overstated, as it significantly contributes to the overall quality of the software product. By identifying and rectifying defects early in the development process, testing minimizes the likelihood of issues affecting the software's performance or functionality, ultimately enhancing user satisfaction and trust.

2. Purpose of Testing:
In the context of the Cashier class, testing ensures that cashier-related functionalities, such as user credential generation and employee data management, operate seamlessly and adhere to specified requirements. By detecting defects early, testing mitigates the risk of issues proliferating throughout the software system, thereby reducing the cost and effort required for remediation.

3. Focus on Testing a Single Component:
The Cashier class serves an important function in the software system for managing cashier information, which emphasizes the significance of thorough testing. This part must perform accurately because it is in charge of important functions including processing employee data and creating user credentials. Developers can verify that the Cashier class behaves as planned and complies with business rules controlling cashier management by testing it in a variety of scenarios.

4. Preparing Test Cases:
Comprehensive test cases are essential for thorough testing of the Cashier class. These cases should encompass a spectrum of scenarios, ranging from normal inputs to edge cases and invalid inputs. For instance, test cases may include validating the generation of user credentials with valid input data, handling empty phone numbers or default salary values as edge cases, and verifying the class's behavior when confronted with invalid inputs, such as negative salary values. By preparing these test cases, developers can ensure exhaustive coverage of the Cashier class's functionalities and behaviors.

5. Choosing Testing Frameworks:
Choosing a good testing framework such as JUnit makes testing more organized and productive when it comes to Java programming. JUnit is a great option for verifying the functionality of the Cashier class because it offers powerful facilities for creating, running, and maintaining unit tests. In order to set up the JUnit testing environment, the framework must be included into the project's build settings. This will streamline the testing procedure and increase its overall effectiveness.

6. Writing Test Code:
Writing test code for the Cashier class entails creating test methods that exercise its various functionalities. These methods should be crafted to cover different scenarios and inputs, ensuring comprehensive testing coverage. Employing assertions within test methods enables developers to validate expected outcomes against actual results, effectively verifying the correctness and reliability of the Cashier class's behavior.

8. Test Coverage:
These test cases cover various functionalities of the Cashier class, including getters and setters for name, surname, phone, and salary, as well as the generation of user credentials and the toString() method. Each test method verifies that the actual output matches the expected output for the respective functionality. This not only helps identify areas of the code that require further testing but also enhances the overall reliability and quality of the software product, instilling confidence in its performance and functionality.

TEST CASE #4 (Loard Bejko)

1. Introduction to Testing:
Software testing is the systematic process of evaluating software to identify defects or bugs. In the context of the Product class, testing involves analyzing its functionality, attributes, and validation mechanisms to ensure that it behaves as expected and meets specified requirements. The importance of testing in software development for ensuring reliability and correctness cannot be overstated.
2. Purpose of Testing:
The goal of testing is to find flaws early in the development process so that they can be fixed quickly and don't affect other processes. Testing helps preserve the integrity of the system and improves its reliability by confirming that the Product class and its related features operate as intended. This ultimately helps to produce high-quality software solutions.

3. Focus on Testing a Single Component:
Testing the Product class is essential due to its central role in managing product data within the system. With its multifaceted responsibilities and complex attributes, including validation logic,

defects within this class can have significant repercussions on inventory management, order processing, and customer experience. Thorough testing helps identify and rectify potential issues early, mitigating risks, improving software quality, and ensuring the reliability and stability of the system, ultimately enhancing overall performance and user satisfaction.

4. Preparing Test Cases:
Test cases for the Product class should cover a range of scenarios, including valid inputs, boundary cases, and invalid inputs. For instance, test cases may validate the setting and retrieval of product attributes, as well as the behavior of the isValid method for input validation.

5. Choosing Testing Frameworks:
To execute tests for the Product class, you would typically run the ProductTest class using a testing framework like JUnit. This framework automatically initializes the necessary objects and executes the test methods defined within the class.
6. Writing Test Code:
Test code for the Product class involves creating test methods to verify its functionalities. These methods should test various aspects of the class, such as attribute setting and retrieval, validation of input data, and the toString method for generating product information.
7. Running Tests:
During the execution of tests, passing scenarios occur when the actual behavior matches the expected behavior defined within the test cases. For example, in the testSetName method, if the setName method successfully sets the name of the product and the assertion assertEquals("New Name", product.getName()) passes, the test case is marked as passing. Failing scenarios occur when the actual behavior deviates from the expected behavior defined within the test cases. This could happen due to issues such as incorrect implementation or unexpected behavior within the code being tested.
8. Test Coverage:
Achieving high test coverage is essential for thorough testing of the Product class. It ensures that all aspects of the class, including methods and edge cases, are correct and the ERP system will work without any defects. High test coverage contributes to the overall reliability and quality of the software product.