Echo Connect: A Telecom Service Company

Milestone 5 – Final Report

COEN 6312 – Model Driven Software Engineering (Section V)

Submitted By
Group 5

Swathi Nagarajan – 40163094

Sourabh Chaturvedi – 40189866

Rishi Murugesan Gopalakrishnan – 40200594

Sai Sivaarchith Samsani – 40231393

Naga Venkatesh Kapalavai – 40231923

Alekhya Bandlamudi – 40240833

Submitted To

Wahab Hamou-Lhadj, PhD, ing.

&

Mohammed Shehab

Department of Electrical and Computer Engineering

Gina Cody School of Engineering and Computer Science

Winter 2024

Table of Contents

List of Figures

List of Tables

Introduction

Across the milestones, our project focuses on designing and implementing Echo Connect – a Telecom Service System, with a strong emphasis on utilizing models throughout the development process. Starting with a thorough domain analysis in Milestone 1, we identified stakeholders, established system features, created a breakdown of requirements, and implemented use case diagrams. Expanding upon this foundation, Milestone 2 included drawing UML class and object diagrams, implementing the class diagram in Java code, and writing test cases using JUnit. In Milestone 3, our focus shifted to implementing complex OCL constraints using Papyrus software, ensuring model validation and adherence to specified rules. Finally, in Milestone 4, we concentrated on implementing behavioral diagrams such as State, Sequence, and Communication diagrams, alongside coding state diagrams and conducting unit tests to verify functionality. Through these milestones, we aim to deliver a robust Telecom Service System that aligns with stakeholder needs and expectations.

Objectives

- Develop the Echo Connect system comprehensively, using models extensively to understand the domain and requirements.
- Ensure accuracy and validity of models by implementing UML diagrams and complex OCL constraints, validating them through rigorous testing.
- Translate designed models into functional Java code, integrating system boundaries, key concepts, and requirements.
- Create and implement behavioral diagrams like State, Sequence, and Communication diagrams to visualize system behavior and interactions.
- Regularly gather feedback, ensuring the system meets required needs, and iterate based on feedback input.

Project Retrospection

In this report, we bring together the progress we've made in building the Echo Connect App across the last three milestones. We've also taken into account the feedback from our TA and applied it to each milestone submission, refining our work along the way. The following are the feedback from TA for each milestone:

- Milestone 1:
  - The non-functional requirements must be as list and you need to fix it as Reliability, Maintainability, etc.
- Milestone 2:
  - Attributes such as PlanID, PlanName, PlanCost, and PlanDescription are redundant. These attributes should be moved to the parent class for reuse in the child classes.
  - There needs to be a relationship between billing and the child classes.
  - In the Association relationship, the object must be declared as a class attribute.
  - You must write attributes as private, protected in some cases.
- Milestone 3:
  - It should be TV.allInstances()->includes(self) in OCL # 8
  - it should be = instead of < > in the post-condition in OCL #10

Problem Statement

Our system serves as a comprehensive Business Support Services platform designed to efficiently manage telecommunications services offered by Echo Connect Telecommunication Service Company. The key functionalities of our system encompass the following processes:

- Billing
- Customer orders
- Subscriptions
- Customer notifications
- Service fulfillment
- Payment management
- Promotions
- Customer relationship management (CRM)

Echo Connect extends a range of services, including Voice, Data, SMS, TV, Wi-Fi, and various Bundle options to its customers. Each customer is assigned a unique account ID to access any service provided by the system, and the account status can be activated or deactivated based on customer requests.

Individual customers are uniquely identified with a user ID and password, granting them access to their profile. A customer can manage multiple lines or services within the same account, accessible by family members or friends. The customer portal allows users to log in or register, update profile details, provide feedback, request support, view audit details, and log off from the application.

Customer acquisition occurs through different methods such as direct, agent, or reseller channels, requiring the recording of procurement details. Customers can place orders and installation requests for Wi-Fi/TV services through the system. Privacy preferences and authentication methods can be set to secure customer accounts.

The Billing module is responsible for generating bills based on the billing cycle, including features like cycle number, start date, and end date. Applicable promotions or discounts are reflected in the credits section. A unique bill ID and the bill amount are generated after analyzing settlements and dues. Notifications are sent to the registered mail ID upon bill generation, with options for paper or email billing preferences.

The Payments module creates a unique payment ID associated with the billing ID, storing payment details such as credit, debit, or cash. Completed payments update the billing module, adjusting the account balance. If an account accumulates more than four unpaid bills, it is transferred to the collections team for resolution.

The Order Management team processes new orders, recording them in the accounts section. Multiple orders can be associated with a single account ID, each having a start date and end date.

The Employee module includes divisions and subdivisions like Marketing, HR management, and support teams. Each division has a manager and a dedicated set of employees. Customer support requests are handled through a ticket management system, with managers assigning issues to their respective employees.

Support services are available through live text, calls, or email. Troubleshooting guidelines and FAQs are provided on the support page to address common problems.

Notifications are set up for invoice generation, data limits, usage warnings, and due date reminders. Users can provide feedback, suggest improvements, and report issues through the system, contributing to continuous enhancement.

System Description

The telecom industry has a wide range of communication services, including voice calls, messaging, data transmission, and multimedia services. Telecom companies provide these services to individuals, businesses, and other organizations, often through a combination of wired and wireless networks. The industry relies on some infrastructural physical components such as towers, cables, switches, routers, satellites, and data centers to provide communication. Services are typically accessed through various devices such as smartphones, tablets, computers, and specialized telecom equipment. Customers interact with telecom providers through multiple channels, including websites, mobile apps, customer service hotlines, and physical stores.

Key functionalities provided by the system include customer registration, account management, billing, service activation, device management, customer support, and feedback collection. The system allows customers to register for telecom services, view and manage their accounts, update profile details, customize billing preferences, activate, or deactivate services, monitor usage, and handle device-related issues. It also provides communication between customers and support representatives, enabling users to request assistance, raise tickets for issue resolution, and provide feedback on their experiences. Internal stakeholders such as employees, managers, and HR personnel utilize the system for various tasks including employee management, task assignment, leave management, hiring, training, performance evaluation, and feedback analysis. The system incorporates features to streamline operations, enhance customer experience, improve service delivery and continuous improvement within the telecom organization.

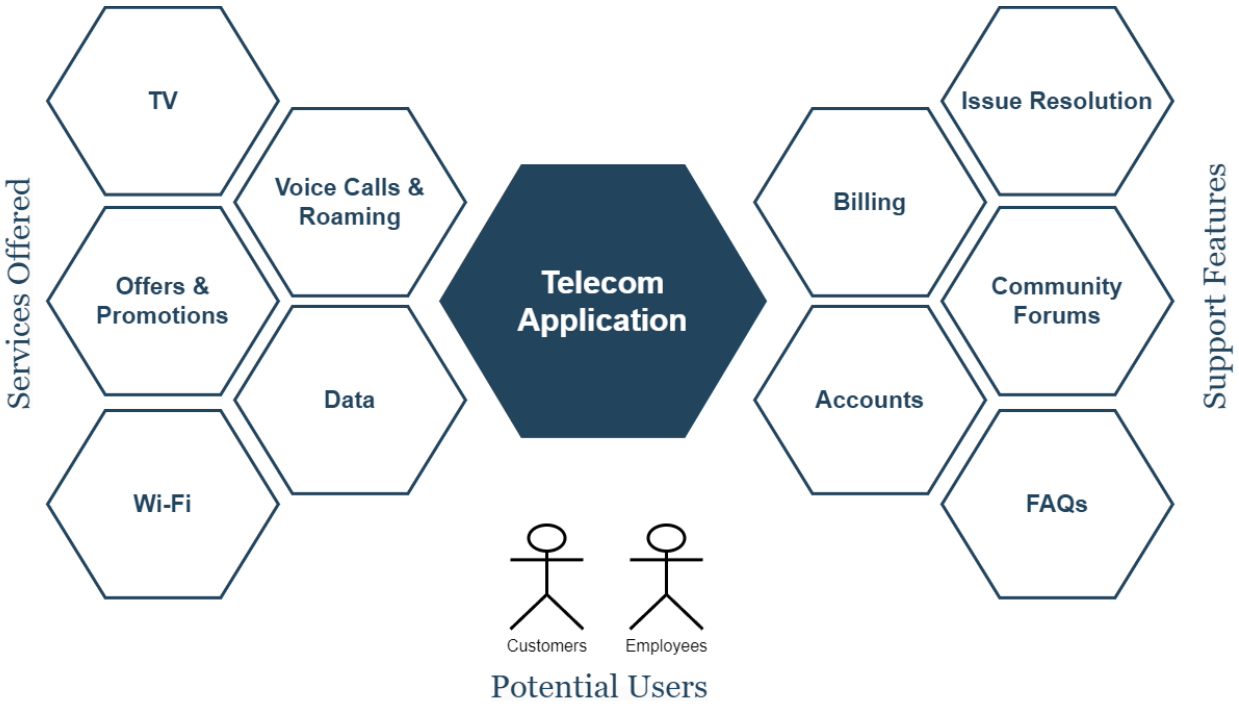

Figure 1 System Description

System Features

1. **Account Creation/Authentication:**
    I. **New Customer Registration:** This feature will allow new users to create an account by using Google authentication or by creating a username and password to access the application.
    II. **New Employee Registration:** New Employees will be assigned with username and password by Managers.
    III. **User login:** Registered users can log successfully into the application by entering a valid username and password or by using Google authentication.
    IV. **User logoff:** Registered users can log out from the application after using the application.
    V. **Multiple Device Login Support:** Registered customer should be able to login to the account from multiple devices.
2. **Account Management:**
    I. **Access the profile page:** If a user logs into the application for the first time, it will allow the user to create a profile. If the user is already registered, they can view or delete their profile information.
    II. **Update profile details:** This will allow users to update their profile information through any modifications.
    III. **View notifications:** Customers will be notified on bill generations, due date remainders, data limits, usage warnings, roaming, and resolution status.
    IV. **Manage notifications:** Customers can change this notification setting by changing the remainder dates or limiting notifications to services.
    V. **Refer a friend:** The customer should refer a friend to share the benefits of the telecom service.
    VI. **Auditing and Logs:** The customer should be able to view and download detailed call logs for auditing and record-keeping purposes.
3. **Service Customization:**
    I. **View and Manage Available Services:** Customers can see all the available services related to mobility plans, Wi-Fi and TV packages, devices, and home phones.
    II. **Activate/Deactivate telecom services:** Customers can select a new service or modify any existing service.
    III. **Manage Subscription Plan:** With this feature, Customers can create, modify or remove any subscription plans as per their requirement.
    IV. **Order and Installation:** This allows Customers to place new order for any service or request any installation request for their Wi-Fi/TV.
    V. **Promotions:** The customer should be able to view or request for any valid promotions applicable on his/her account.
4. **Employee Module:**
    I. **Role/Task Assignment:** The Manager assigns roles and tasks to the employees based on the department.
    II. **Task tracking:** The division manager can view all the tasks with the help of this task tracking feature, and these are enabled with priority, so the highest priority is to be completed first.

III. **Manage Plans and Promotions:** Employees with the necessary permissions can add any plans or promotions to a customer's account. The applied changes should accurately reflect in the customer's billing and promotional sections.
IV. **Access Employee Resources:** Employees can search and access employee resources to answer customer's basic queries without connecting them to a different department

5. **Billing Module**
   I. **Invoice generation:** Generates invoices differently based on the service, and also it provides a single invoice (one bill) for family/friend's account.
   II. **View/download bills:** Customers can view or download the bills.
   III. **Dispute bills:** Customer should be able to easily dispute or inquire about any discrepancies or unfamiliar charges on the bill.
   IV. **View Bill Status:** Customers can view the bill status if it's paid or outstanding.
   V. **Make payments:** This feature allows Customers to navigate to the secure payment page from where users are allowed to pay the bills.
   VI. **Payment information:** Customers can save their bank information, debit/credit cards, and other wallets for faster payment processing. They can also update or delete saved payment methods.
   VII. **Pre-Authorized payments:** For faster payment processing, Customers can opt for the pre-authorized payment or autopay option.
   VIII. **Due date remainder:** Customers will be notified via email for payment remainders, or any overcharges applied to the account.
   IX. **Download Payment Receipt:** Customer should be able to download a payment receipt upon successful payment.
6. **Privacy and Security**
   I. **Privacy preferences:** Customers can restrict access to other family or friends to view or edit their profile settings.
   II. **Account recovery:** If users forget their login credentials, they can recover by sending an Account recovery process to their registered mailing address.
   III. **Two-Factor Authentication:** Users are provided with an enhanced security option while logging into their account.
7. **Support and Help Center**
   I. **Customer support:** Users can connect to any of the customer support Employees via live text, call, or by sending an email. The issues that are raised by the users are handled by the ticket managing system.
   II. **Ticket assignment:** Employees will assign these tickets to their respective departments for resolving them.
   III. **View/Track existing tickets:** The users can view or track raised tickets.
   IV. **Troubleshooting guidelines:** Users have access to guides and tools for troubleshooting common problems faced while using the application.
   V. **FAQs:** Users are provided with an FAQ section to address frequently asked questions allowing users to limit the use of live chat.

8. **Feedback**
    I. **Customer feedback:** Customers can share their experiences, improvements, and issues while using the application.
    II. **Employee feedback:** HR Managers can be able to rate the employees based on their performance with the help of this feature. And Employees can submit their experience about their workplace.

System Context Diagram

The System Context Diagram shows the Telecom Service System as a single high-level process and then shows the relationship that the system has with other external entities (Customer, Employees, etc.).Some of the benefits of a Context Diagram is that it clearly defines system scope and boundaries, illustrates interfaces with external entities, easily understood without technical expertise, simple to draw and amend and can benefit a wide audience including stakeholders, business analyst, data analysts, developers.

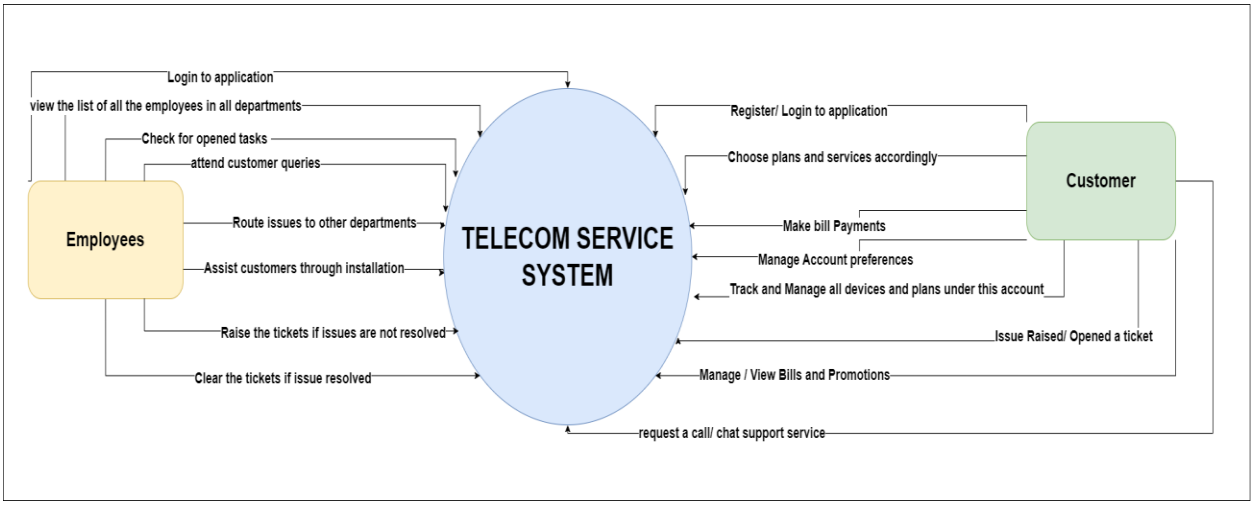

Figure 2 System Context Diagram

Tech-Stack

- **Programming Language**: Java
- **IDE**: Eclipse
- **MDE:** Papyrus Eclipse
- **Version Control**: GitHub

User Stories

Table 1 User Stories

Feature	Parental User ID	Parent User Story	Child User Story Feature	Child User Story	Child User Story ID
Access Application	P1	As a User, I should be able to access the application so that I can utilize or access the telecom services.	New Customer registration	As a new customer, I want to register for telecom services so that I can start using communication services.	C1
			Customer Login	As a customer, I want to login to my account so that I can access the web application.	C2
			User Log Off	As a user, I want to logoff from the application so that I can close the web application.	C3
			Employee Login	As an employee, I should be able to log in to the application so that I can look into the details.	C4
			Multiple device login support	As a customer with multiple devices, I should be able to login from my all the devices simultaneously so that I can access my account conveniently.	C5
Account Management	P2	As a user, I want to be able to access my account so that I can utilize its features and functionalities.	Create the Profile Page	As a new customer, I want to create my profile, providing necessary information and preferences.	C6
			View/ Delete the Profile Page	As a customer, I want to view or delete my profile information.	C7
			Update the Profile Page	As a customer, I want to update my profile details so that I can provide the latest information.	C8
Service Management	P3	As a Customer, I want to manage my services effectively so that I can customize my telecom plan to meet my specific needs.	Activate/Deactivate Services	As a customer, I want to activate or deactivate specific telecom services so that I can customize my plan based on my current needs.	C9
			Cancel all my service	As a customer, I want to cancel my service so that I can discontinue my telecom services when needed.	C10
			View available subscription plan	As a customer I should be able to see my active subscription plan	C11
			Upgrade/Downgrade subscription plan	As a customer I should be able to upgrade/downgrade my subscription plan	C12
			View all the subscription plan	As a Customer I should be able to view available subscription plans and offers	C13
			Request for a new plan activation	As a new Customer I should be able to request for new plan activation	C14
			Bundling of plans for customer needs	As a new Customer I should be able to select from multiple available plans and request for bundling them	C15

Subscription Plan Management	P4	As a manager I should be able to manage the subscription plans so that I can keep it up to date.	Creating a new subscription plan	As a manager I should be able to create a new subscription plan so that I can add new plans into the system.	C16
			Modify an existing subscription plan	As a manager I should be able to modify existing subscription plan so that I can update the details on the plan.	C17
			Remove an existing subscription plan	As a manager I should be able to remove an existing subscription plan	C18
Billing Management	P5	As a Customer, I want a centralized bill management system providing an effective and transparent billing experience.	Viewing Bills	As a Customer, I want to be able to view my bills so that I can understand the specific costs associated with the services.	C19
			Download Bills	As a Customer, I want to download my current or previous bill/s whenever needed.	C20
			Dispute Bills	As a Customer, I want to easily dispute or inquire about any discrepancies or unfamiliar charges on my bill to ensure accurate billing.	C21
			One Bill	As a Customer with multiple plans, I should get only one consolidated bill so that I can easily manage and track my overall expenses.	C22
			View Bill Status	As a Customer, I should be able to see if the current bill is paid or outstanding so that I can track my payments.	C23
Payment System	P6	As a Customer, I want a user-friendly and secure payment management system that enables me to make a hassle-free payment.	Autopay Method	As a Customer, I want to set up and manage autopay for my bills so that payments are automatically deducted from my chosen payment method.	C24
			Available Payment Options	As a Customer, I want to pay my bills with either my credit/debit cards or using wallet for easy and smooth payment option	C25
			Save Payment Methods	As a Customer, I want to save my payment information for faster payment processing.	C26
			Modify Saved Payment Methods	As a Customer, I want to modify my saved payments either by updating or deleting them to keep them updated.	C27
			Secured Payment Page	As a Customer, I want to be redirected to a secure landing page when paying the bill to have a seamless payment process.	C28
			Download Payment receipt	As a customer, I want to download my payment receipt so that I can keep it for any disputes.	C29
Promotions and Discounts	P7	As a user, I want to view or update or manage the	View all Promotions and offers	As a Customer, I should know the available promotions and offers which can be applicable on my account so that I can redeem them.	C30

		promotions, so that I can redeem them.	Update new promotion and offers	As a Marketing manager, I want to keep updating all the current available offers to the customers so that I can keep them updated.	C31
			Apply Promotions and offers	As a Employee, I should be able to able to apply valid promotions on the customer's account so that I can fulfill their request.	C32
Privacy and Security	P8	As a Customer, I want to set my privacy features and have security options so that I have enhanced security for my account.	Account Recovery	As a customer, I should be able to use the forget password option so that I can retrieve my login details.	C33
			Setting privacy preferences in user profile	As a user I want the option to set the privacy preferences in my profile so that I can control the visibility of certain information.	C34
			Two-Factor Authentication	As a customer, I should be able to set two factor authentications so that I can have enhanced security option.	C35
Support Ticket Management	P9	As an Employee, I want to efficiently manage customer issues through a comprehensive Support Management System so that we can provide timely and effective assistance to our customers.	Create a Ticket	As an employee, I should be able to create a ticket if the issue is not solved in a request so that the relevant department can solve it later.	C36
			Raise a Ticket	As a customer, I should be able to raise a ticket so that I can get help regarding my issue.	C37
			Assign customer-generated tickets	As an employee, I should be able to assign customer-generated tickets to the relevant department, so that the customer issue is addressed.	C38
			Set and Modify Priority Levels	As an employee, I should be able to set and modify priority levels on tickets so that tickets with critical issues get addressed first.	C39
			Reassign a Ticket	As an employee, I should be able to reassign a ticket if resolving takes too long so that the ticket will be resolved faster	C40
			View All Tickets in the System	As an employee, I should be able to view all tickets in the system so that I can use it for future reference.	C41
			Filter Tickets in the System	As an employee, I should be able to filter tickets in the system so that I can filter out important or relevant tickets.	C42
			View All Logs	As an employee, I should be able to view all logs so that I can use them for future reference and calculate metrics	C43
Notification System	P10	As an existing user, I want to receive timely notifications related to my subscription,	Reminders and Notification to the customers about Plan and usage	As an existing customer, I should receive reminders and notification related to my subscription and usage	C44
			Notification about new bill	As a Customer, I want to receive a notification when a new bill is generated	C45

		including alerts for new bill generation, approaching due dates, and updates on the resolution status of tickets I raised.		every time so that I can stay informed.	
			Notification about due date	As a Customer, I want to receive a notification when the due date is approaching so that I can avoid overdue fees	C46
			Notification about ticket resolution status	As a user, I should be able to get notifications about the resolution status of the tickets I raised so that I can know the progress of my ticket.	C47
Auditing and Logs	P11	As a customer I want to view my call logs to know my usage.	View Logs	As a customer, I want to view and download detailed call logs for auditing and record-keeping purposes.	C48
Referral	P12	As a customer I want to use the referral program offered by the application to get added benefits.	Refer a friend	As a customer, I want to refer a friend so that I can share the benefits of the telecom service and possibly receive rewards for successful referrals.	C49
Resources	P13	As a User I should be able to access the necessary materials from the application so that I can know how to resolve my queries.	Access and Search FAQ	As a customer, I should be able to access and search the FAQ so that I can try to solve the issue on my own	C50
			Access and Search Employee Resources	As an employee, I should be able to access and search the employee resources so that I can answer customer's basic queries without connecting them to a different department.	C51
Feedback	P14	As a User, I should be able to provide feedback in order to support the improvements.	Customer Feedback	As a customer, I want to provide feedback so that I can share my experiences and contribute to service improvements.	C52
			Employee Feedback	As an Employee, I should be able to provide feedback about the workplace so that improvements can be done	C53

Use Case Diagram

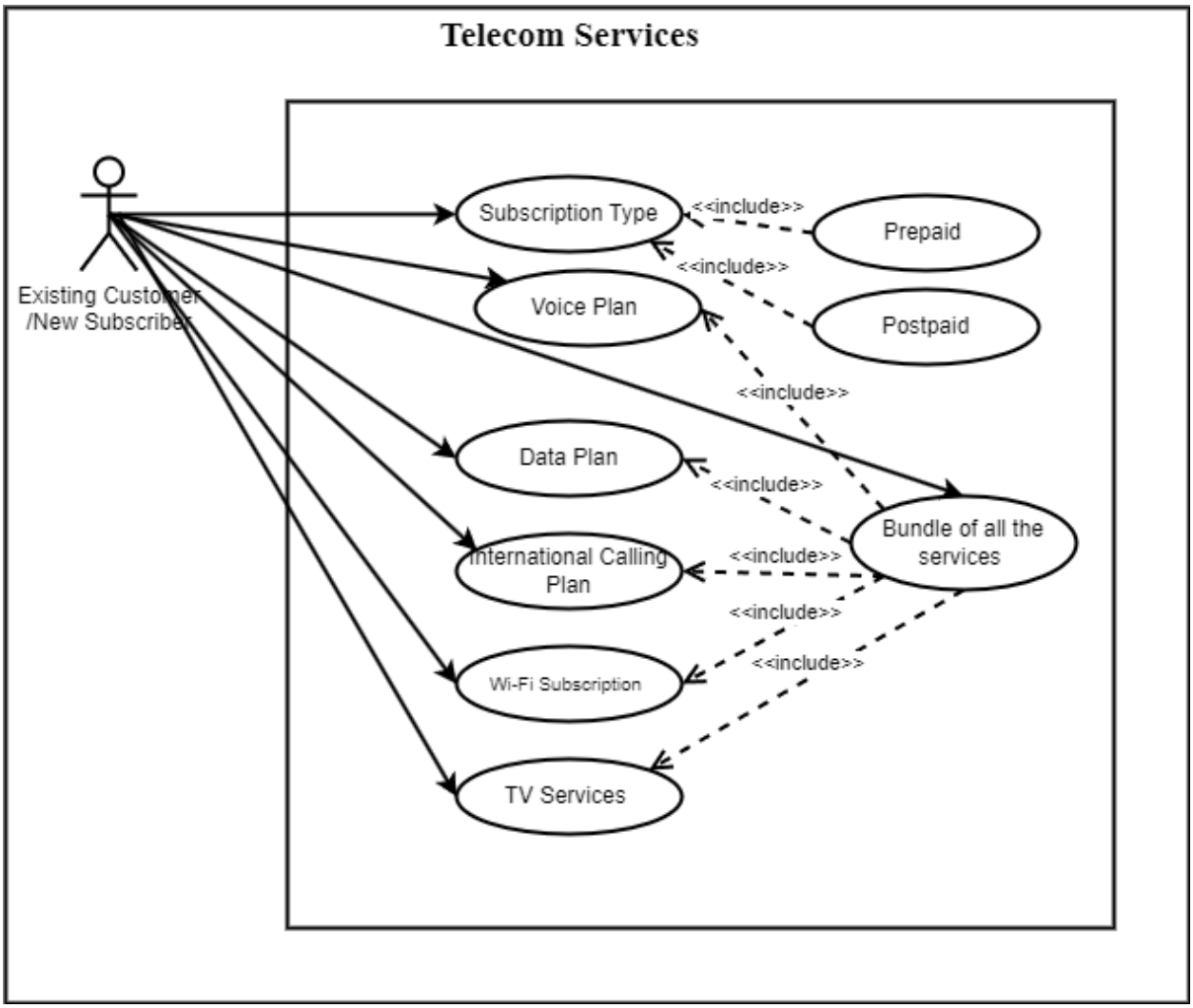

Figure 3 Telecom Services Use Case Diagram

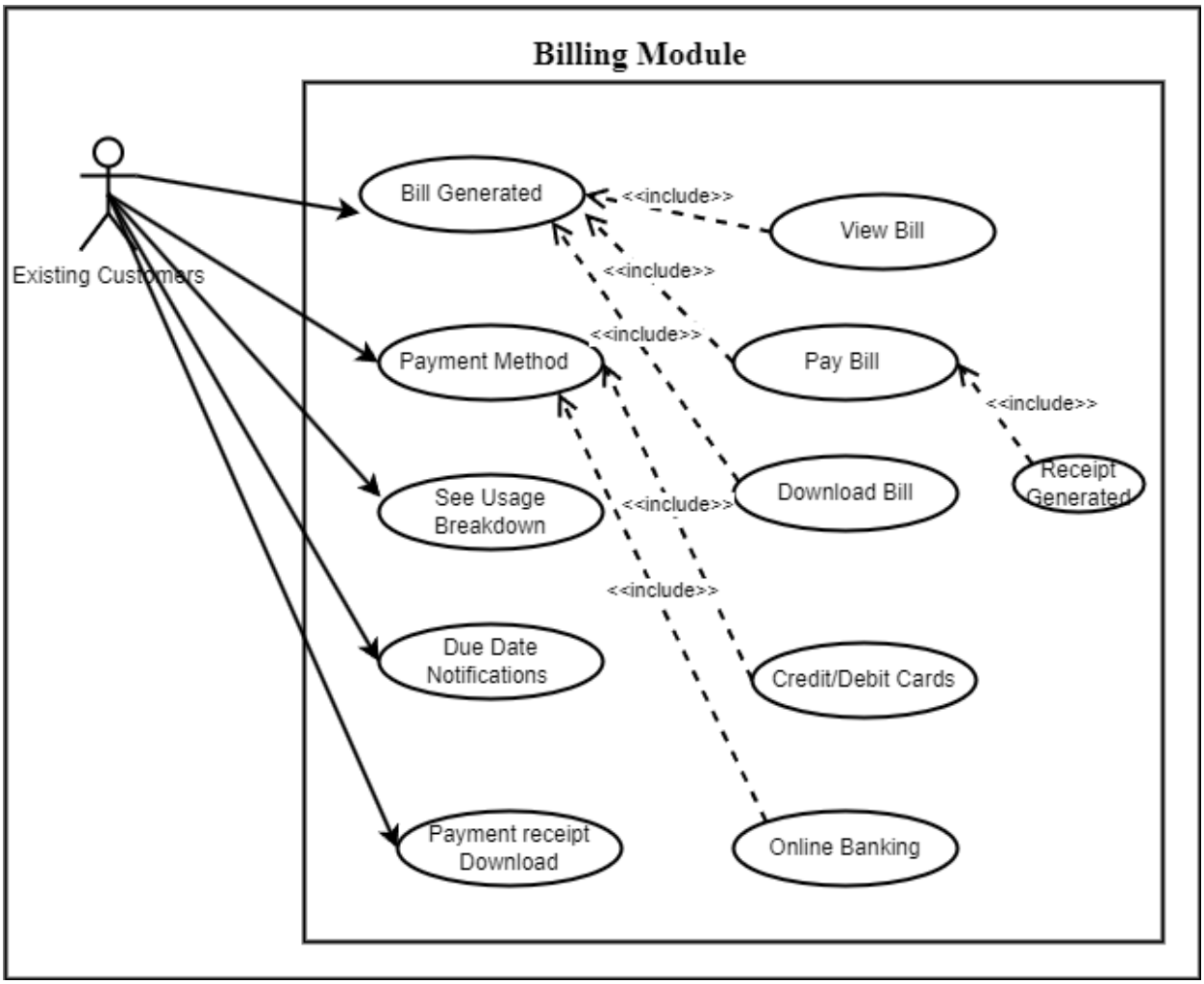

Figure 4 Billing Use Case Diagram

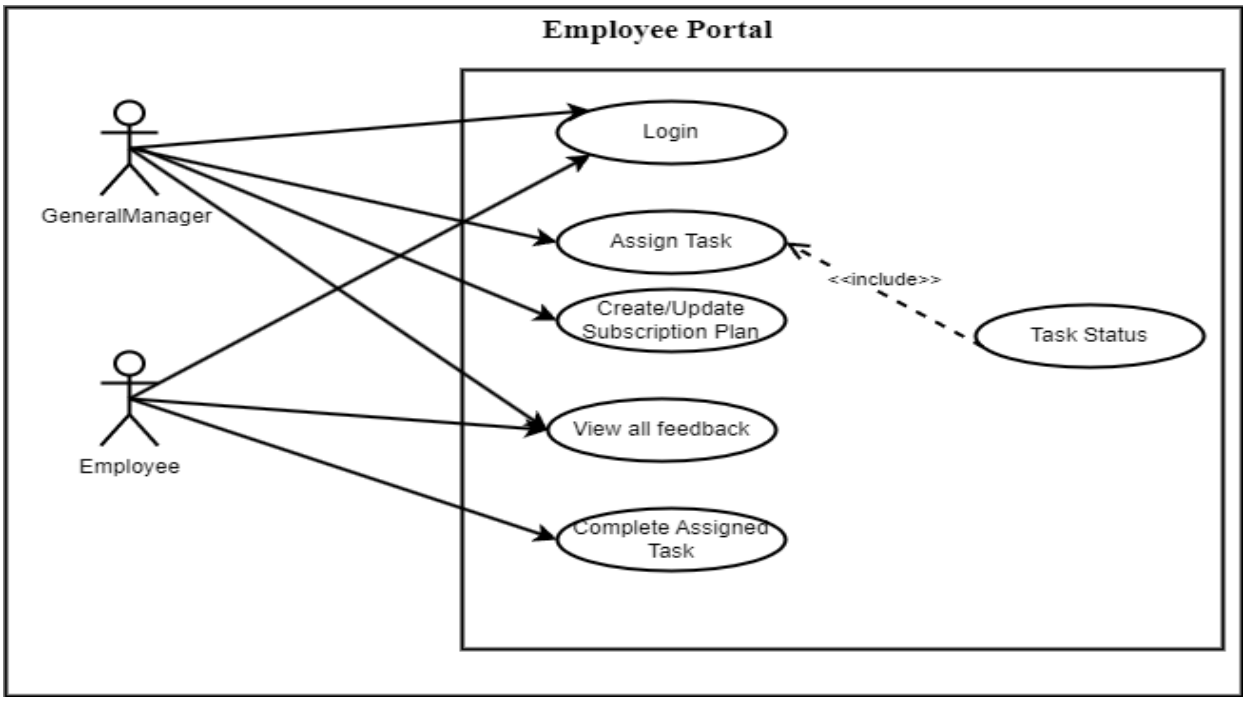

Figure 5 Employee Use Case Diagram

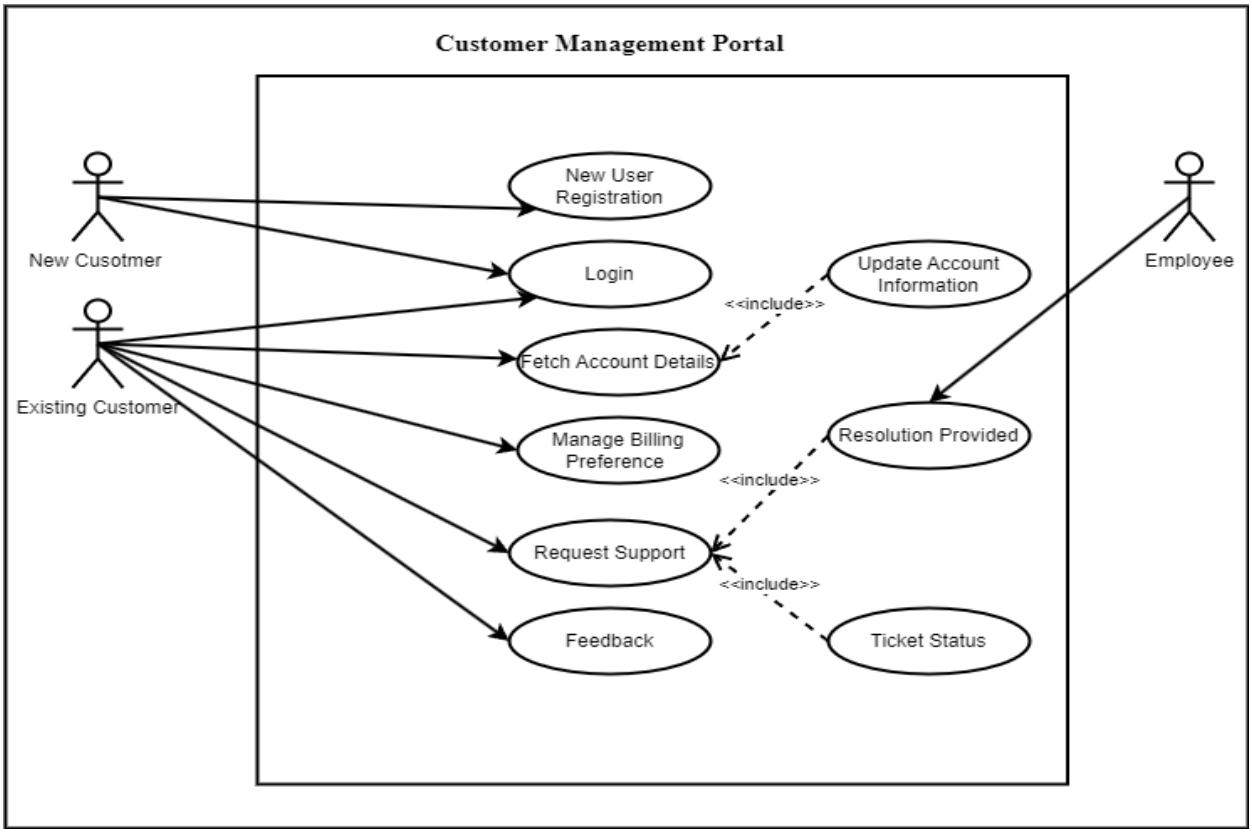

Figure 6 Customers Use Case Diagram

Functional Requirements Mapping

Table 2 Functional Requirements

Child User Story ID	Pre-Condition	Post-Condition	Requirements	Target Users
C1	Users must be new customers accessing the application for the first time.	Customers have a registered account for accessing telecom services.	The application must have a user-friendly registration form. The registration process should include mandatory fields for essential information. The system should validate and store the entered information securely.	New Customer
C2	Customers must have a registered account.	Customers are securely logged into their account, gaining access to the web application.	The web application must have a clearly visible and accessible login option. The login page should provide secure entry fields for username/email and password. The system must validate user credentials for successful login. In case of unsuccessful login attempts, the system should display appropriate error messages.	Existing Customer
C3	Customers must be logged into the application.	Customers are securely logged off from the application.	The application must have a clear and easily accessible logoff option. Logging off should clear session data and ensure security.	Existing Customer
C4	Employees must have a valid username and password. The employee account must be active and not locked.	After a successful login, the employee gains access to the application's features. In case of an unsuccessful login, the system provides clear error messages. After multiple unsuccessful login attempts, the system may temporarily lock the account for security.	Secure login page with encryption for data transmission. User authentication system with validation against a secure database. Account management system for handling active, inactive, and locked accounts. Error handling mechanism for displaying appropriate messages on failed login attempts. Security measures, such as account lockout, to protect against unauthorized access.	Employee
C5	Users must be registered and logged into the application.	User can change account information from any device he logged into	The application must support multiple session and store the user credentials for easy authentication	Existing Customer
C6	Users must be new customers registering for telecom services.	Customers have a completed profile associated with their account.	The registration process should smoothly incorporate the creation of a profile. The profile must encompass sections for both personal details and preferences.	Existing Customer
C7	Customers must be logged into the application.	Customers can view or delete specific	The application must have a dedicated section for viewing and managing profile information.	Existing Customer

		profile information as needed.	Users should be able to access and review their profile details.	
C8	Customers must be logged into the application.	Updated profile details are saved and reflected in the system.	The application should provide a user-friendly profile modification interface. All profile fields should be editable.	Existing Customer
C9	Customers must be registered and logged into the application.	The customer's telecom plan is customized based on the activated or deactivated services.	The application must provide a service customization section. Customers should easily identify and select services to activate or deactivate. The system should adjust the customer's plan and billing accordingly.	Existing Customer
C10	Customers must be registered and logged into the application.	The customer's telecom services are discontinued, and billing adjustments are made.	The application must have a service cancellation feature. Customers should be able to provide a reason for cancellation (optional). The system should update billing and service status accordingly.	Existing Customer
C11	The customer must be logged into their account. The customer should have an active subscription plan.	The customer successfully views details of their active subscription plan. The system displays accurate and up-to-date subscription information.	User-friendly interface displaying current subscription details.	Existing Customer
C12	The customer must be logged into their account. The customer's account must be in good standing with no outstanding issues.	The customer's subscription plan is successfully upgraded or downgraded. The system updates the customer's account with the new plan details.	System capability to process customer requests for plan changes.	Customer
C13	The customer must be logged into their account. The system must have a list of available subscription plans and current offers.	The customer successfully views the list of available subscription plans and current offers. The system displays accurate and up-to-date plan information.	System should show catalog of available subscription plans with detailed information.	New Potential Customer
C14	The new customer must have created an account on the platform. The system must have available plans for activation.	The new customer's plan activation request is successfully submitted. The system processes the request and activates the chosen plan.	New customers can submit a request for plan activation during the signup process.	New Potential Customer
C15	The new customer must have created an account on the	The new customer's bundle selection is successfully	System should support bundling of the plan	New Potential Customer

	platform. The system must offer multiple plans that can be bundled together.	submitted. The system processes the request and activates the bundled plans.		
C16	The manager must be authenticated with appropriate permissions.	A new subscription plan is added to the system. The plan is immediately accessible to customers for subscription.	A user-friendly interface for managers to create new subscription plans.	Manager
C17	The manager must be authenticated with appropriate permissions. For modifying or removing an existing plan, the plan must already exist in the system.	The existing subscription plan details are successfully modified. Customers subscribed to the modified plan see the updated details.	A user-friendly interface for managers to modify existing subscription plans.	Manager
C18	The manager must be authenticated with appropriate permissions. For modifying or removing an existing plan, the plan must already exist in the system.	The existing subscription plan is removed from the system. Customers subscribed to the removed plan receive notifications about the plan removal. Customers are provided with alternative subscription options.	System functionality to safely remove subscription plans without disrupting customer subscriptions.	Manager
C19	The customer must have an active account with associated service plans.	The customer must have an active account with associated service plans.	A user-friendly interface displaying detailed bill information. Accessibility to current and previous bills.	Existing Customer
C20	The customer must have an active account with billing history.	Customers have the ability to store or print bills for record-keeping.	Download feature accessible through the customer's account. Bills available for download in PDF or similar format.	Existing Customer
C21	The customer must identify discrepancies or unfamiliar charges on their bill.	Discrepancies are addressed, ensuring accurate billing.	User-friendly interface for dispute initiation. Timely response from customer support.	Existing Customer
C22	The customer must have multiple active plans.	Customers receive a consolidated bill that simplifies expense tracking.	Billing system capable of recognizing and aggregating charges from multiple plans.	Existing Customer

C23	The customer must have an active billing cycle.	Customers can easily track the payment status of their current bill.	Real-time update of payment status on the customer's account dashboard.	Existing Customer
C24	The customer must have a valid and active payment method on file.	Payments are automatically deducted, providing a hassle-free experience.	Autopay option available in customer account settings. Secure storage and retrieval of payment method information.	Existing Customer
C25	The customer must have an outstanding bill.	Payments are successfully processed using the chosen payment method.	Integration with secure payment gateways. User-friendly payment interface with clear options.	Existing Customer
C26	The customer must have initiated a payment with the option to save payment information.	Saved payment information expedites future payment processes.	Secure storage of payment information with customer consent.	Existing Customer
C27	The customer must have saved payment information.	Customers can manage and maintain updated payment details.	User-friendly interface for modifying saved payment information.	Existing Customer
C28	The customer must initiate the bill payment process.	Customers experience an ideal and secure payment process.	Integration with secure and reputable payment gateways.	Existing Customer
C29	The customer must have completed a successful payment.	Customers have a downloadable payment receipt for record-keeping.	The receipt should be available in a standard, easily readable format.	Existing Customer
C30	The customer must have an active account.	Customers are aware of promotions available for redemption.	Integration with a promotions module that tracks available offers.	Existing Customer
C31	The marketing manager must have access to the promotions management interface.	Customers receive timely updates about new promotions.	Integration with a promotions management system. Notification system to alert customers about new offers.	Marketing Manager
C32	The employee must have the appropriate authorization to apply promotions.	Customers see the applied promotions in their account and billing details.	Employee access controls to ensure only authorized personnel can apply promotions. Integration with the billing and promotions modules for real-time updates.	Employee
C33	Customers must have a registered account but have forgotten or need login details.	Customers receive their login details securely.	The application must include a "Forgot Password" or "Request Login Details" feature. Customers should be able to provide necessary verification information for security.	Existing Customer

C34	Users must be registered and logged into the application.	User can set privacy preferences based on his device profile	The system must support the saved profile preferences simultaneously across the devices	Existing Customer
C35	The customer must have an existing account. The customer must have a valid and accessible second factor (e.g., mobile device, authenticator app).	Two-factor authentication is successfully enabled for the customer's account. The customer receives a confirmation message or email about the successful setup.	Account settings section with an option to enable 2FA. Clear and user-friendly guidance for the 2FA setup process. Support for multiple 2FA methods (e.g., SMS, authenticator app). Secure and reliable communication for sending verification codes. User education materials on the importance and usage of 2FA.	Existing Customer
C36	The employee must be authenticated and have appropriate permissions. The issue reported by the customer must remain unresolved after the initial request.	A new ticket is created in the system. The employee receives confirmation of the successful ticket creation.	Employees can create tickets for unresolved issues through a dedicated form. The form includes details about the issue and allows for the selection of priority levels. A confirmation message is displayed upon successful ticket creation.	Employee(Customer Support Representative)
C37	The customer must be authenticated.	A new ticket is created and linked to the customer account.	Customers can raise tickets for ongoing or unresolved issues through a ticket creation interface. An automated acknowledgment email is sent upon receiving a customer-raised ticket.	Existing Customer
C38	The employee must be authenticated and have appropriate permissions. Relevant departments and employees must be predefined in the system.	The ticket is assigned to the specified department. Notification is sent to the relevant department.	Ticket assignment functionality. Integration with departmental information.	Manager
C39	The employee must be authenticated and have appropriate permissions.	The priority level of the ticket is updated. The system stores the modification history.	Employees can set and modify priority levels for tickets through a priority selection interface. Priority options are clearly described, and modifications are allowed based on issue severity.	Employee(Customer Support Representative)
C40	The employee must be authenticated and have appropriate permissions. The ticket resolution process exceeds a	The ticket is reassigned to a new employee or department. The system logs the reassignment event.	Employees can reassign tickets if resolution takes too long through a reassignment option. Notifications are sent to the newly assigned employee and department.	Employee(Customer Support Representative)

	predefined timeframe.			
C41	The employee must be authenticated and have appropriate permissions.	The employee has a comprehensive view of all tickets.	Ticket listing functionality. User-friendly interface for viewing tickets.	Employee(Customer Support Representative)
C42	The employee must be authenticated and have appropriate permissions.	The system displays a filtered list of tickets.	Ticket filtering options based on various criteria. Responsive and efficient filtering mechanism.	Employee(Customer Support Representative)
C43	The employee must be authenticated and have appropriate permissions.	The employee has access to a detailed log of system activities.	Log storage and retrieval functionality. Clear presentation of system logs.	Employee(Customer Support Representative)
C44	The user must have an existing account with a valid subscription. The user must have an active and valid mobile number or email address associated with their account for receiving notifications.	Users receive timely notifications related to their subscription and usage. Users can take appropriate actions based on the received notifications, such as making timely payments or adjusting their usage.	System integration with user database for subscription and usage data. User preference settings for notification types and channels. Notification content clarity and relevance. Prompt notifications for bill generation, due dates, and ticket updates. Option for users to mark notifications as read or dismiss. Integration with preferred communication channels (e.g., email, SMS, app).	Existing Customer
C45	The user is an existing subscriber with an active account.	The user receives a notification for each new bill generated.	Integration with billing system to trigger notifications. User settings to control notification preferences. Secure handling of billing information.	Existing Customer
C46	The user has an upcoming bill with an approaching due date.	The user receives a timely notification before the due date.	Integration with billing system to calculate and trigger notifications. User settings to configure notification timing. User interface for snoozing or dismissing notifications.	Existing Customer
C47	The user has raised one or more tickets.	The user is informed about the progress and resolution of their tickets through notifications.	Integration with the ticketing system to fetch and update ticket status. User settings to manage ticket notification preferences. Clear communication of ticket status in the notifications.	Existing User(both Customer and Employees)
C48	Users must be registered and logged into the application.	Users can review and download detailed call logs for auditing purposes.	The application must include a call log section. Users should be able to filter and search for specific call details. The system should support the download of call logs in a user-friendly format.	Existing Customer
C49	Users must be registered and	Successful referrals are tracked, and users	The application must have a referral feature. Users should be able to send referral invitations through various channels.	Existing Customer

	logged into the application.	may receive rewards as applicable.	The system should track successful referrals and manage rewards.	
C50	The customer must have access to the company's website or app. The customer should be experiencing an issue and seeking a resolution.	The customer finds a solution to the issue through the FAQ. If the issue persists, the customer can proceed to other support channels.	FAQ section on the company's website or app. Search functionality with relevant keyword matching. Categorized FAQ topics for easy navigation. Regularly updated FAQ content.	Existing user
C51	The employee must be authenticated and have the necessary permissions. The employee should be attempting to address a customer's basic query.	The employee successfully finds information to answer the customer's query. If necessary, the employee can escalate the query to a specialized department.	Employee resource section accessible through company systems. Search functionality with efficient keyword matching. Diverse and comprehensive employee resources. Regular updates to ensure the relevance of information.	Existing user
C52	Users must be registered and logged into the application.	Feedback is recorded for analysis and potential service improvements.	The application must include a feedback submission feature. Users should be able to provide detailed feedback, including text and optional attachments. The system should categorize feedback for analysis.	Existing Customer
C53	Employees must be authenticated to access the feedback system. The feedback system should be accessible through company-provided devices or platforms.	Feedback is submitted and stored in a secure database. An acknowledgment or confirmation is provided to the employee upon successful feedback submission.	Feedback system with authentication. Anonymity option in the feedback form. Secure storage and handling of feedback data. Periodic analysis and reporting tools for HR/Management.	Employees

Non – Functional Requirements

- **Performance**:
  - The system must efficiently handle a substantial number of concurrent logins and service requests.
  - User interactions, such as viewing bills or updating profiles, should exhibit prompt response times, ideally within 3 seconds.

- **Availability**:
  - The application must be always accessible, 24/7, with minimal downtime reserved for maintenance.
  - A failover mechanism should be in place to ensure uninterrupted service in the event of server failures.

- **Reliability**:
  - The system must accurately process and store customer data, maintaining a high level of precision without errors.
  - Robust backup and recovery mechanisms must be implemented to prevent any loss of data.

- **Security**:
  - Secure user authentication and authorization methods must be employed, incorporating secure password storage and transmission.
  - Two-factor authentication should be implemented to enhance overall security.
  - Customer data, encompassing personal and billing details, must undergo encryption both in storage and during transmission.

- **Scalability**:
  - The system must be scalable to effectively accommodate an increasing number of customers and services.
  - Scalability should be achievable through both horizontal (adding more servers) and vertical (upgrading server resources) means.

- **Usability**:
  - The user interface must be designed for ease of use, providing an intuitive experience that allows customers to navigate effortlessly and execute tasks.
  - Accessibility features must be integrated to ensure the application's usability for individuals with disabilities.

- **Auditability**:
  - The system must maintain comprehensive logs of user activities, encompassing usage details, service requests, and modifications to customer profiles.

❖ **Notification System**:
- Notifications must be consistently delivered in a reliable and timely manner.
- The notification system must support various channels (e.g., email, SMS) based on customer preferences.

❖ **Integration**:
- The system must possess the capability to seamlessly integrate with external systems for billing, customer relationship management, and other relevant functionalities.

Class Diagram

- **EchoConnectApp**
  - The **EchoConnectApp** class serves as the central application for our telecommunication system.
  - It establishes a **one-to-one association** with the **Person** class through the user attribute of type **Person**. This association implies that each instance of **EchoConnectApp** is linked to precisely one individual user.
- **Person**
  - The **Person** class represents the users within the system.
  - The **One-to-One association** is reinforced by the **app** attribute in the **person** class, linking each person to exactly one instance of the **EchoConnectApp**.
  - The **Customer** and **Employee** classes are inherited from the common base class, **Person**.
- **Customer**
  - The **Customer** class represents individual users of our telecommunication.
  - **Each customer can be association with multiple services (One-to-Many Association)**. The association is facilitated through the **MyServices** attribute which is a list containing instances of the Services class.
- **Services**
  - The **Services** class manages various telecommunication services.
  - The **one-to-many** association from **customer** class is established using the **customerID** attribute. This bidirectional relationship enables customers to view, subscribe to, and manage their services while allowing each service to be uniquely identified by its association with a customer.
- **Plan**
  - The **Plan** class represents the fundamental blueprint for service plans within our telecommunication application.
  - It encapsulates essential attributes that provide a standardized template for defining various service plans.
  - The **association** between the **Plan** class and the **Service** class is established as a **composition relationship**, indicating that **each service contains one or more instances of the Plan class**.
  - This allows the management of service of plans within the context of individual services.
  - Furthermore, **subclasses derived from the 'Plan' class** extend its functionality to cater to specific service types such as **Mobility**, **WiFi**, **Tv** and **Bundle**, each inheriting the common plan attributes from the Plan class while introducing additional attributes and functionalities tailored to their respective service offerings.
- **Billing**
  - The **Billing** class serves as a representative of billing information within our telecommunication application, and it encapsulates essential attributes that facilitate the management and tracking of billing details for services.

- The **association** between **Services** class and **Billing** class is established as a simple association. **Each service may be associated with zero to many billing instances, representing the billing transactions or record related to that service**.
- The association List name bills of type Bills in the services class and service instance in Bills class facilitates this relationship.

❖ **Payment**

- The **Payment** class functions as a representation of payment transactions within our telecommunication application.
- The **one-to-one association** established between **Payment** class and **Billing** class, ensures that each payment is directly linked to a single billing record.
- The **BillID** attribute in payment class which references the identifier of the associated billing record facilitates this association.

❖ **Ticket**

- The **Ticket** class serves as a representation of support tickets within our telecommunication application.
- Apart from many essential attributes, it includes **CustomerID** attribute from Customer class, to link each ticket to its associated customer.
- This **association** allows for a **one-to-many association** between **Ticket** class and the **Customer** class, enabling each customer to have multiple associated tickets.

❖ **Feedback**

- The **Feedback** class represents feedback provided by customers within our telecommunication system.
- The **customerID** attribute that references from **Customer** class serves as an association attribute linking each feedback to its associated customer.
- This association establishes a **one-to-many relationship** between the **Feedback** class and the **Customer** Class, allowing **each customer to provide multiple feedback**.

Due to the size of our class diagram, we have placed it on the next page for better clarity and ease of reference. Kindly refer to it for a comprehensive overview of the system's structure and relationships.

Figure 7 Class Diagram

Below are some of the **object diagrams** representing our system:

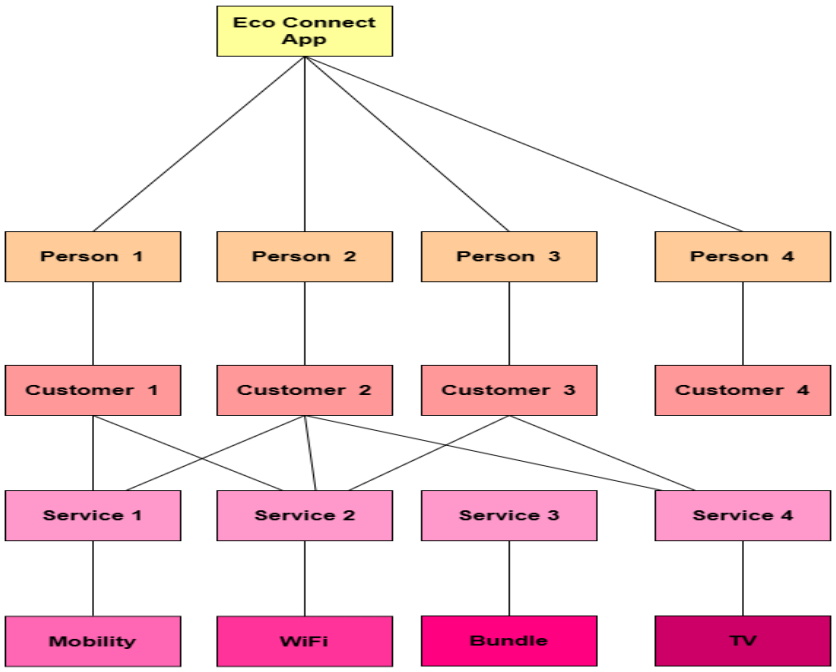

Figure 8 Object Diagram: Customer to Services

This object diagram 1 shows the snapshot of the interaction of the system services with the customers. The main class “Echo Connect App” has one to many relationships to the person class. This indicates that the single App object can be accessed by multiple person instantiations. The Person class is a Superclass for Customer and Employee. Here we have shown that Person class creates four instantiations of the customer class. Every customer should be able to access the services. This object diagram adheres to our class diagram and verifies that a customer can take zero services or many services at a time and there can be a service that will not be taken by any Customers like service object 3, or many customers can access the same services at the same time as service object 2. Moreover, the Service class acts as Superclass to different services like Mobility, Wi-Fi, Tv, Bundle, so the service class will create different instantiations of the sub classes.

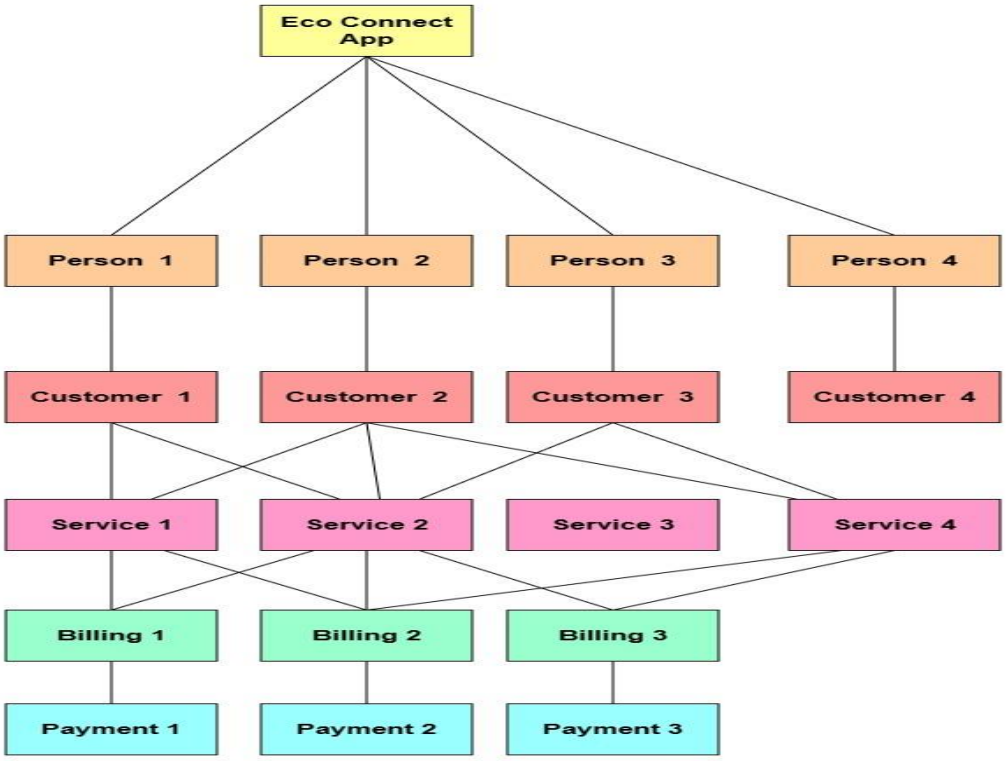

Figure 9 Object Diagram: Customer to Payment

This object diagram illustrates the interactions between the service, billing, and payment classes. It represents that if a service is used by the customer, then he/she should receive bills for all the services taken. In this scenario, Customer 4 has not availed any services, so he/she will not receive any bills. Additionally, for every bill generated, the customer must make a payment.

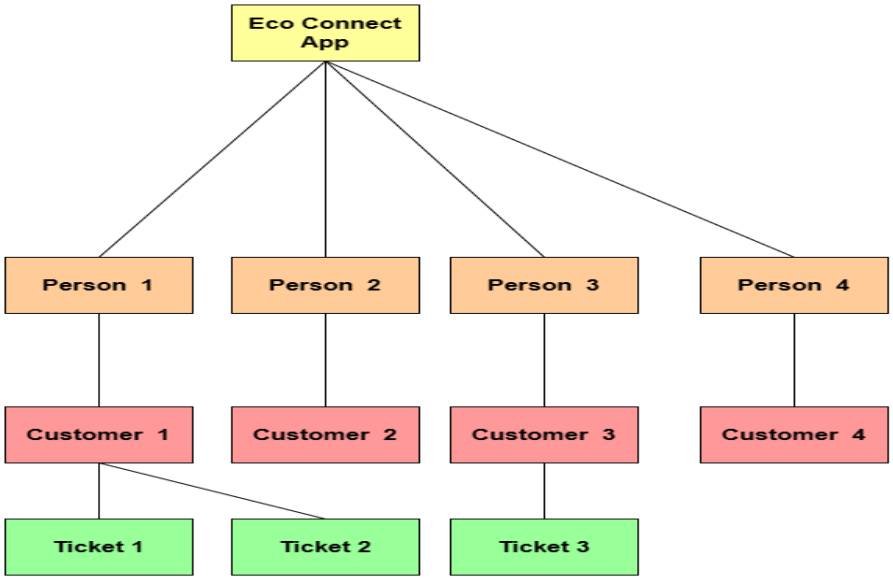

Figure 10 Object Diagram: Customer to Ticket

This object diagram shows the association between Customer and Ticket Classes. Here according to our design, a customer can raise zero or many tickets and a ticket should belong to only one customer. Every ticket will have a unique ticket id which can be referenced by the customer.

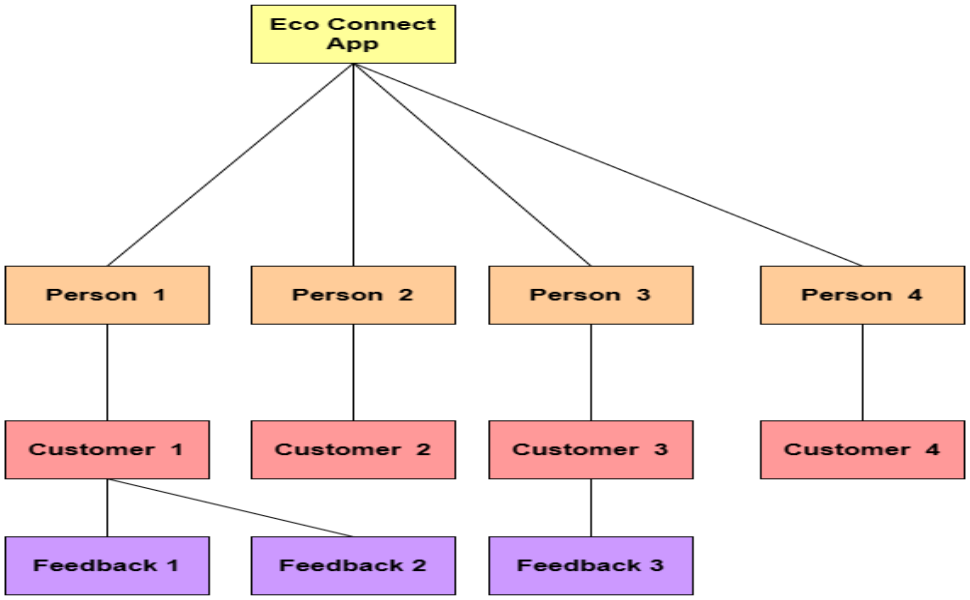

Figure 11 Object Diagram: Customer to Feedback

This object diagram shows the interaction between Customer and Feedback Classes. The feedback class is associated with the customer class. Here according to our work, a customer can give zero or much feedback and feedback should belong to only one customer.

Further we also verified, the relationships between employees and the support system, as well as the dependencies on resources, through object instances.

This verification step helped us to validate and modify the design aligning with the functionalities of our system.

Implementation of Class Diagram with Papyrus Eclipse

Papyrus is a widely used MDE tool integrated into the Eclipse IDE, offering support for designing UML diagrams, including class diagrams. Papyrus supports Model-Driven Development (MDD) by allowing developers to create UML diagrams, including class diagrams, to visualize and design their systems.

We created a new Papyrus project within the Eclipse and created a new class diagram. We started adding classes, attributes, methods, associations, and other UML elements using the Papyrus editor. Later we defined the properties of classes, specified relationships, and set multiplicities as per our system requirements.

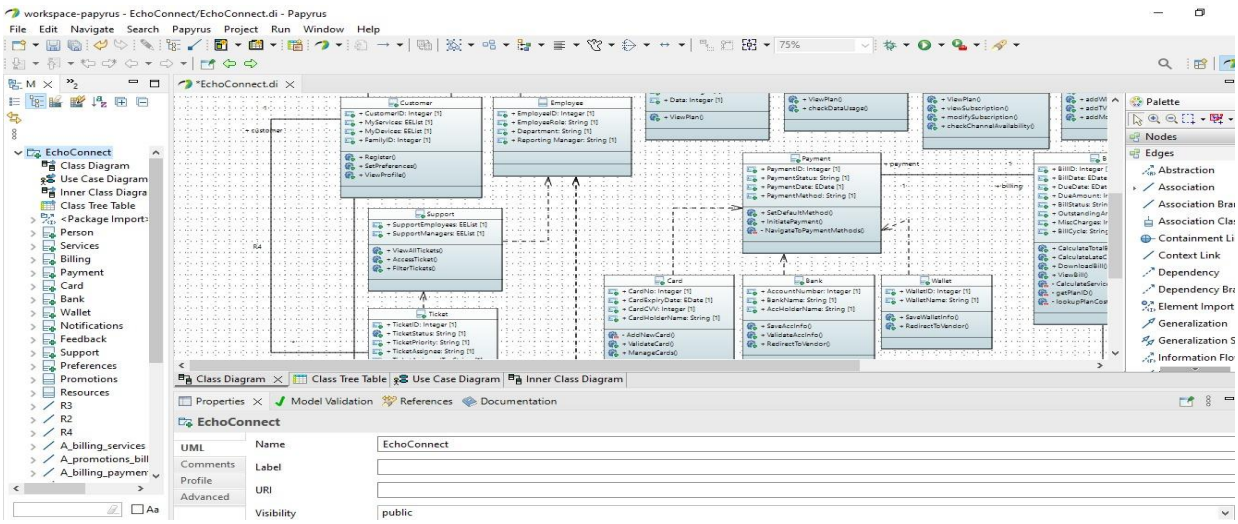

Figure 12 Papyrus Window

Later using this design, we generated the code for the system. Papyrus does not inherently generate full code, but it can be integrated with code generation plugins or tools. But we used the code as starting point and developed our functionalities.

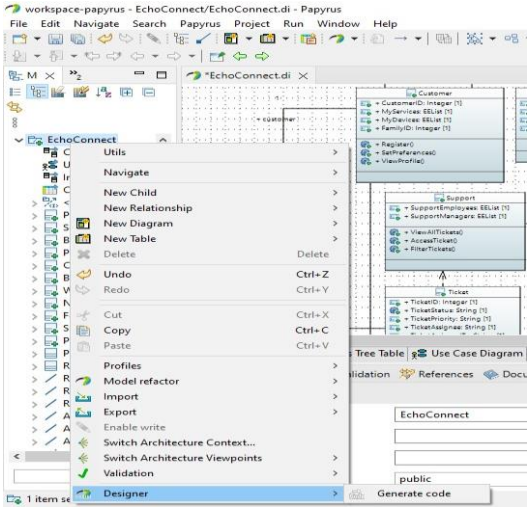

Figure 13 Code Generation Step from Class Diagram

Implementation of Test Cases and Results

Unit testing is a crucial aspect of software development that involves testing individual units or components of a software system in isolation to ensure if the functionalities align with the requirements and the system works as expected.

The project is tested using Junit testing. The test cases have been built and tested successfully without any errors or failures and the results are shown below:

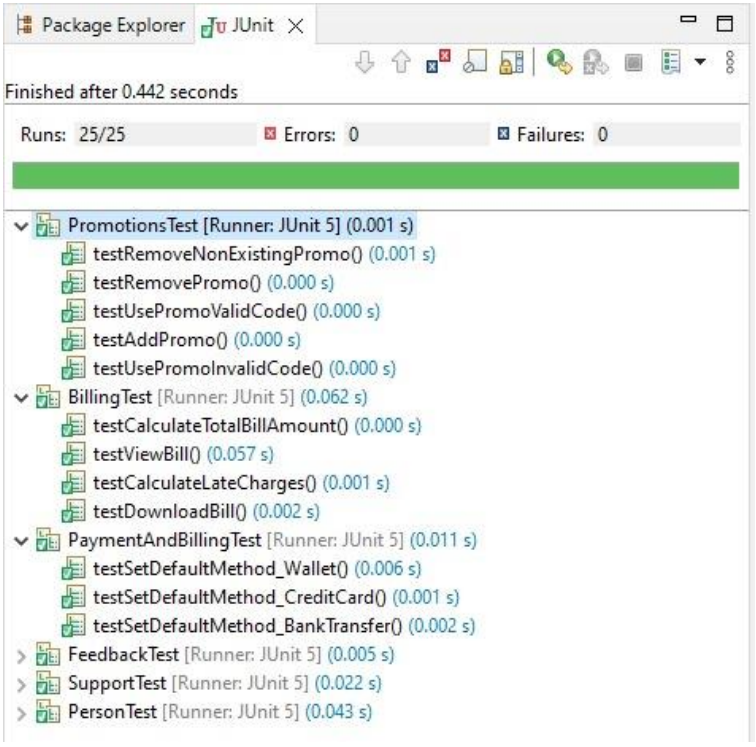

Figure 14 Test Cases Execution

The test cases should cover the code functionality for validating requirements, identifying bugs, preventing regressions, and facilitating code refactoring. This will improve code quality and boost collaboration, ultimately ensuring a stable and reliable software system. The below snapshot shows the sample of code coverage for the Billing Functionality:

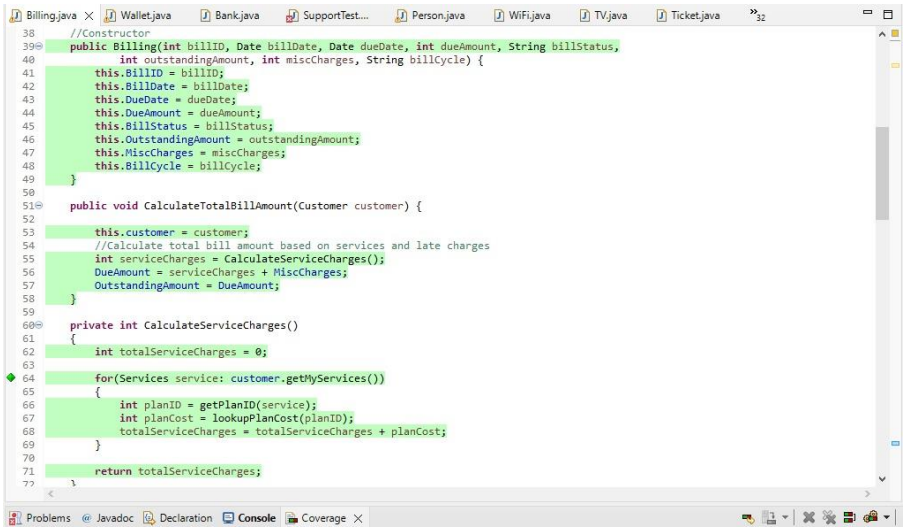

Figure 15 Code Coverage

GitHub Link

https://github.com/Rishi-M-G/Echo-Connect

Relationship Table

The following are the associations and relationships between classes in our telecommunication system.

Table 3 Relationship Table

Relationship Identifier	Classes Involved	Relationship Type	Multiplicity
R1	EchoConnectApp and Person	Association	One-to-One
R2	Customer and Services	Association	Many-to-Many
R3	Person and Preferences	Association	One-to-One
R4	Customer and Ticket	Association	One-to-Many
R5	Customer and Feedback	Association	One-to-Many
R6	Promotions and Billing	Association	One-to-Many
R7	Payment and Billing	Association	One-to-One
R8	Billing and Notifications	Association	One-to-Many
R9	Billing and Services	Association	Many-to-Many
R10	Services and Plan	Composite Aggregation	One-to-Many
R11	Customer and Billing	Association	One-to-Many
R12	Services and Feedback	Association	Many-to-Many
R13	Customer and Notifications	Association	One-to-Many
R14	Customer and Payment	Association	One-to-Many
R15	Payment and Notifications	Association	Many-to-Many

We have covered almost every relationship in the OCL constraints that we have created.

OCL Constraints

We have covered every relationships and incorporated multiple classes in the following complex OCL constraints.

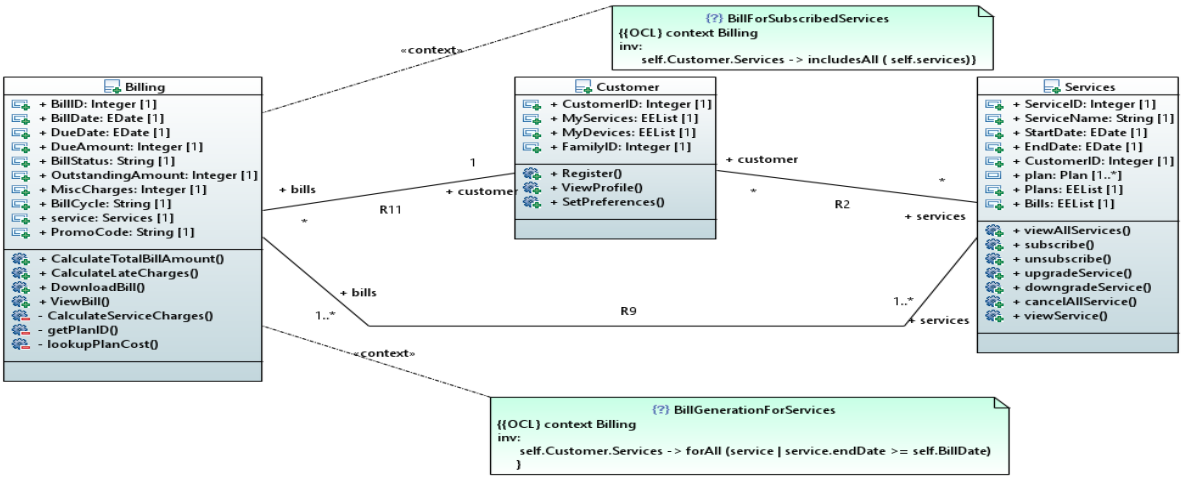

Figure 16 OCLs related to Billing Customer and Services Classes

1. **Bill should not be generated for services that have passed the end date**

context Billing
inv:
self.Customer.Services -> forAll (service | service.endDate >= self.BillDate)

Explanation:

- This constraint applies to instances of billing class
- Self.customer.Services retrieves lists of service instances used by the customer.
- For each service in the list, the end date must be greater than or equal to billing date and this is asserted using the second expression.

2. **Customers should get billed for all services they have subscribed to**

context Billing
inv:
self.Customer.Services -> includesAll (self.services)

Explanation:

- This constraint applies to instances of Billing class
- Self.customer.services retrieves the list of service instances to which customer associated with billing is subscribed and self.services retrives the list of services in billing.

- The OCL expression asserts that all services included in the billing are also part of the customer's subscriptions.

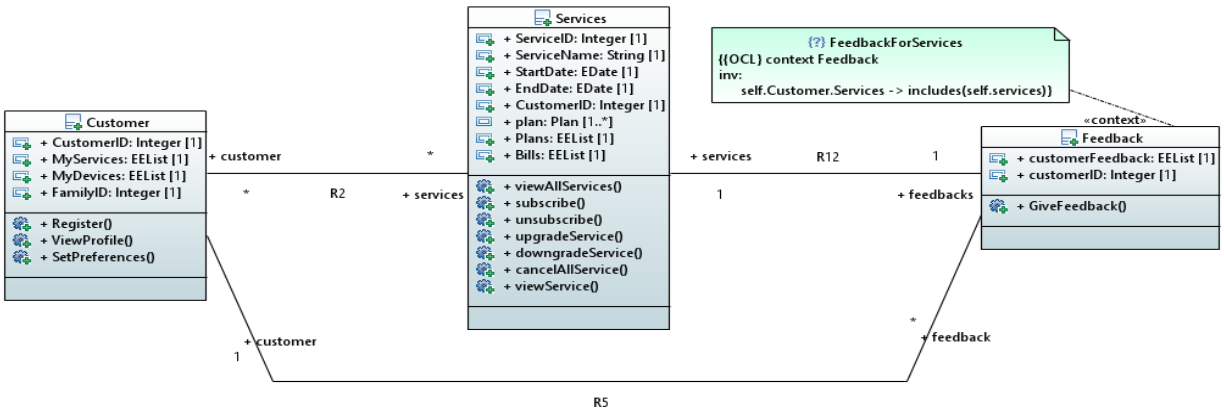

Figure 17 OCLs for Customer, Services and Feedback classes

3. **<u>Customers should not be able to give feedback for services for which he is not subscribed to.</u>**

context Feedback
inv:
self.Customer.Services -> includes(self.services)

<u>Explanation</u>:

- This OCL constraint ensures that for each feedback instance, the service associated with feedback is included in the list of services to which the customer is subscribed.

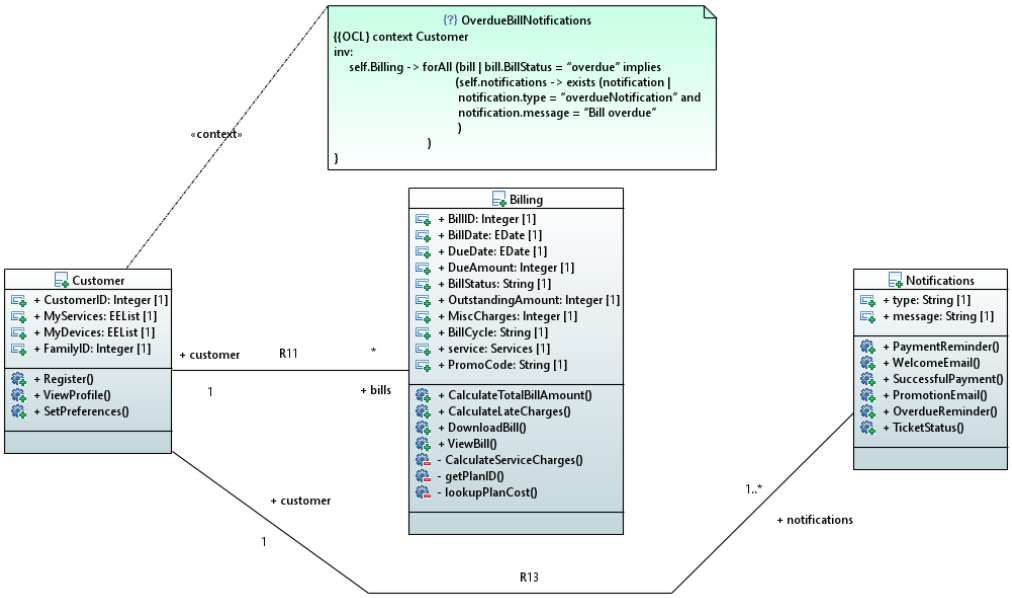

Figure 18 OCLs for Customer, Billing and Notifications Classes

4. **Customers receive notifications for overdue bills.**

context Customer
inv:
self.Billing -> forAll (bill | bill.BillStatus = "overdue" implies
(self.notifications -> exists (notification |
notification.type = "overdueNotification" and
notification.message = "Bill overdue"
)
)

Explanation:

- This OCL constraint ensures that when a bill status changes to overdue, the customer receives an overdue notification from the telecommunication system.
- The change in bill status attribute will trigger the overdueReminder function in Notifications class.

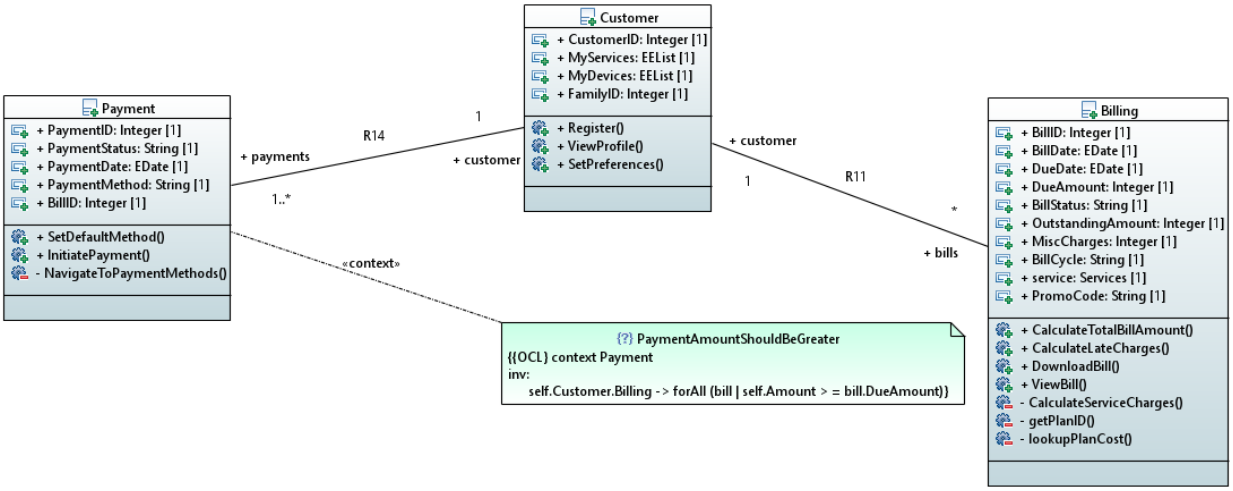

Figure 19 OCLs for Customer, Payment and Billing Classes

5. **The Payment amount must be greater than or equal to the total amount of each billing statement associated with the customer.**

context Payment
inv:
self.Customer.Billing -> forAll (bill | self.Amount > = bill.DueAmount)

Explanation:

- This OCL constraint ensures that for each payment instance, the payment amount is greater than or equal to the total amount of each bill associated with the customer.
- This ensures that the payments paid by the customer covers the total amount owed for each bill.

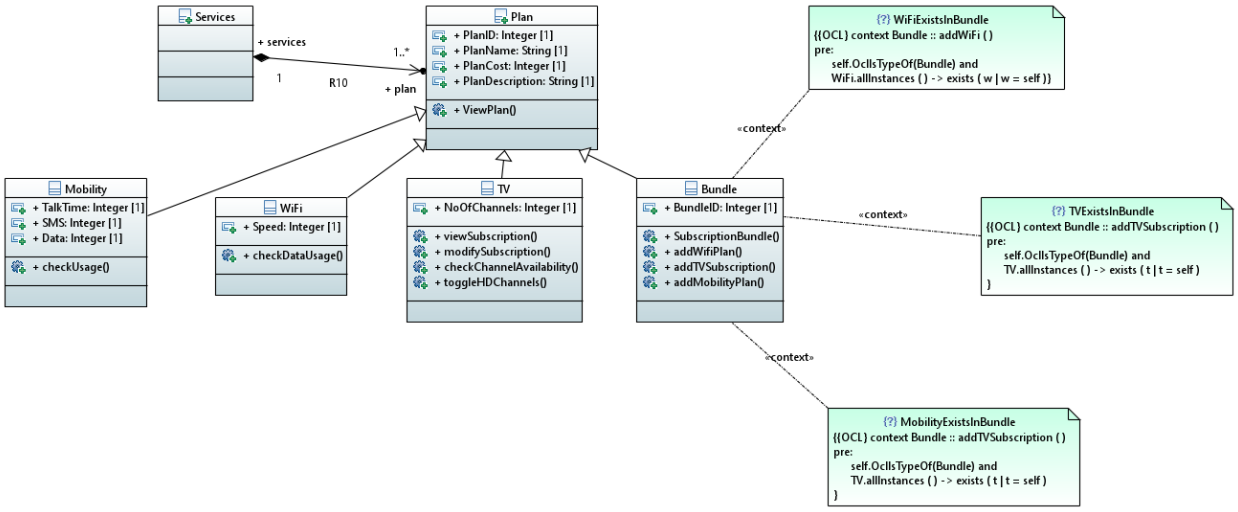

Figure 20 OCLs for Bundle Class

6. **If a customer wants to add a WiFi plan to a Bundle, that WiFi plan should already be existing.**

context Bundle :: addWiFi ( )
pre:
self.OclIsTypeOf(Bundle) and
WiFi.allInstances ( ) -> exists (w | w = self)

Explanation:

- This constraint ensures that before adding a WiFi plan to Bundle, the respective plan must already be existing in the telecommunication system.
- The exists ( ) function checks if there exists any instances of the WiFi class that is equal to the current instance in Bundle class.

7. **If a customer wants to add a TV plan to a Bundle, that TV plan should already be existing.**

context Bundle :: addTVSubscription ( )
pre:
self.OclIsTypeOf(Bundle) and
TV.allInstances ( ) -> exists (t | t = self)

Explanation:

- This constraint ensures that before adding a TV plan to Bundle, the respective plan must already be existing in the telecommunication system.
- The exists ( ) function checks if there exists any instances of the TV class that is equal to the current instance in Bundle class.

8. **If a customer wants to add a TV plan to a Bundle, that Mobility plan should already be existing.**

context Bundle :: addTVSubscription ( )
pre:
self.OclIsTypeOf(Bundle) and
TV.allInstances ( ) -> includes(self)

Explanation:

- This constraint ensures that before adding a TV plan to Bundle, the respective plan must already be existing in the telecommunication system.
- The exists ( ) function checks if there exists any instances of the TV class that is equal to the current instance in Bundle class.

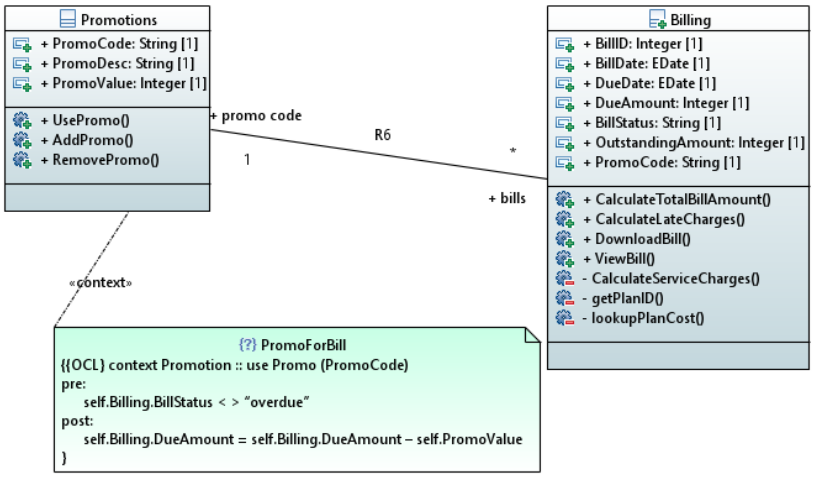

Figure 21 OCLs for Promotions and Billing Classes

9. **Promo code will reduce due amount if and only if bill status is not overdue**

context Promotion :: use Promo (PromoCode)
pre:
self.Billing.BillStatus < > "overdue"
post:
self.Billing.DueAmount = self.Billing.DueAmount – self.PromoValue

Explanation:

- This constraint ensures that the usePromo ( ) function can only be invoked if the billing status is not overdue.
- If the billing status is overdue, the promotion will not be applied.
- The post condition ensures that after the function is invoked, the promotion value is subtracted from the due amount of the associated billing record.

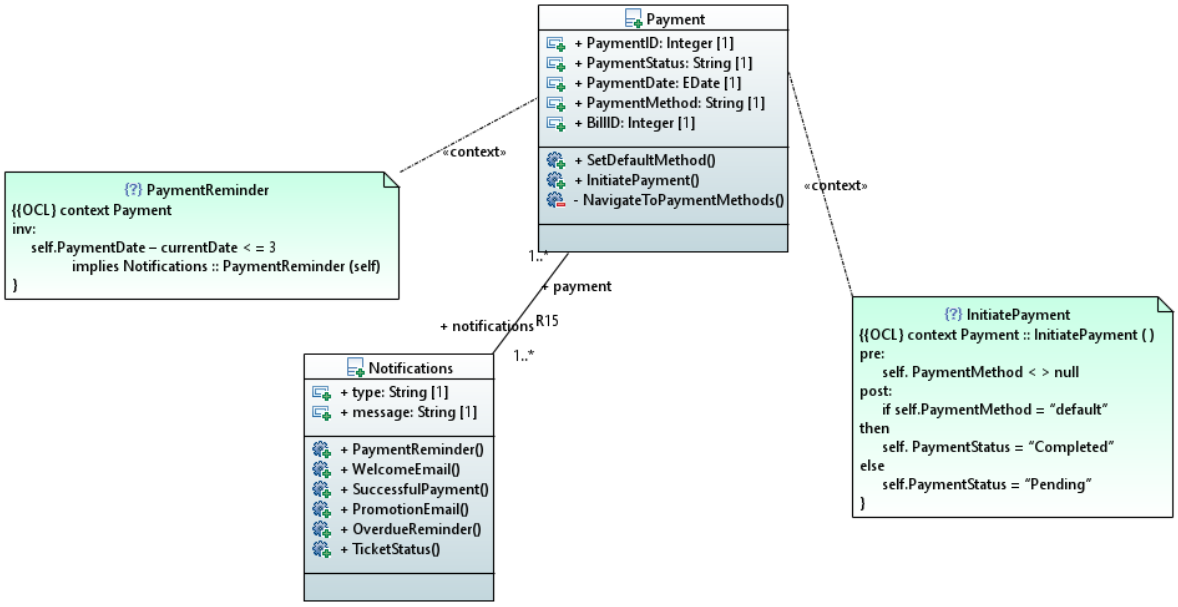

Figure 22 OCLs for Payment and Notifications Classes

10. **<u>Ensure that when initiating a payment using a default payment method, the payment status is set to Completed. However, if navigating to individual methods, the payment status should be pending until the payment is successful.</u>**

context Payment :: InitiatePayment ( )
pre:
self. PaymentMethod < > null
post:
if self.PaymentMethod = "default"
then
self. PaymentStatus = "Completed"
else
self.PaymentStatus = "Pending"

<u>Explanation</u>:

- This OCL ensures that when initiating payment, if the default payment method is chosen, the Payment Status is automatically set to "Completed" after the payment process.
- If navigating to individual payment methods, the Payment Status remains Pending until a specific payment method is chosen and successfully processed.
- Pre-condition is used to ensure that the Payment Method is not null before initiating the payment process.
- Post-condition assigns the appropriate Payment Status based on the chosen payment method.

11. <u>Payment reminder function should be triggered when current date is three days before payment date.</u>

context Payment
inv:
self.PaymentDate – currentDate < = 3
implies Notifications :: PaymentReminder (self)

<u>Explanation</u>:

- This OCL rule is applied to the instances of Payment class
- It checks if the difference between the payment date and the current date is less than or equal to 3 days.
- If the condition is met, it implies that the payment date is approaching thus it triggers the PaymentReminder function from the Notifications class.

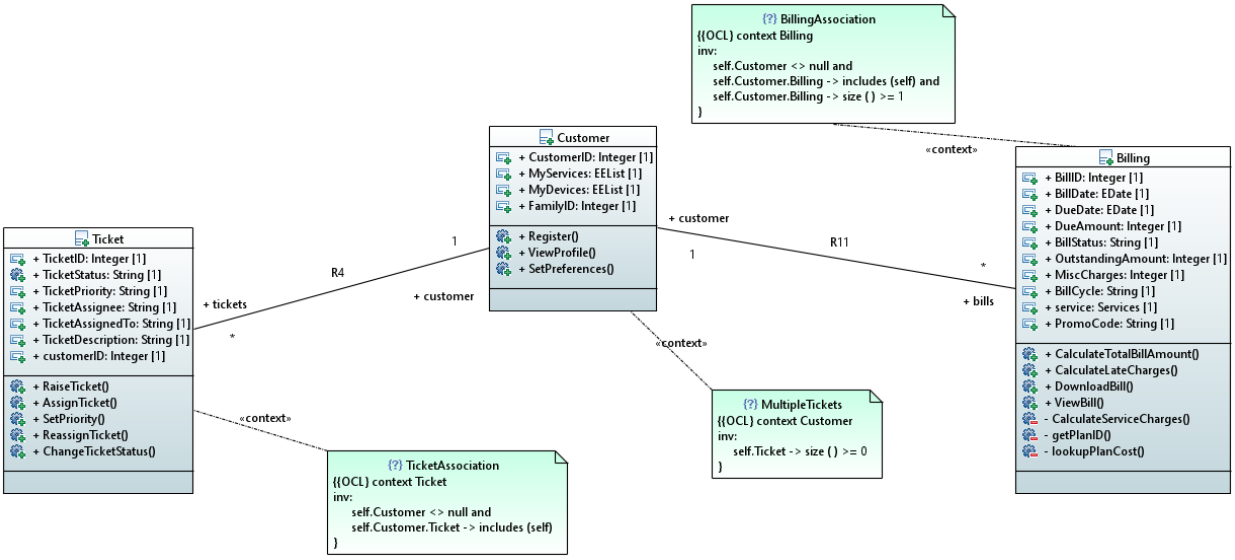

Figure 23 OCLs for Ticket, Customer and Billing Classes

12. <u>Ensure that each billing record is associated with exactly one customer.</u>

context Billing
inv:
self.Customer <> null and
self.Customer.Billing -> includes (self) and
self.Customer.Billing -> size ( ) >= 1

Explanation**:**

- This OCL rule applies to instances of the Customer class.
- The invariant self.billing -> forAll(b | b.customer = self) ensures that for all billing associated with the customer, the customer reference in each billing (b.customer) points back to the same customer (self).
- The self.billing -> size() >= 1 ensures that the number of billing associated with the customer is greater than or equal to one, indicating that the customer must have at least one billing associated with them.

13. Enforce that each ticket is associated with exactly one customer.

context Ticket
inv:
self.Customer <> null and
self.Customer.Ticket -> includes (self)

Explanation:

- The context Ticket specifies that this OCL rule applies to instances of the Ticket class.
- The invariant self.CustomerId -> size() = 1 ensures that each ticket has exactly one customer ID associated with it, meaning each ticket is associated with only one customer.
- The self.tickets -> forAll(t | t.CustomerId = self.CustomerId implies t = self) ensures that each ticket ID is unique among the tickets.

14. Ensure that each customer can have multiple associated tickets.

context Customer
inv:
self.Ticket -> size ( ) >= 0

Explanation:

- This OCL rule applies to instances of the Customer class.
- The invariant self.tickets -> forAll(t | t.customer = self) ensures that for all tickets associated with the customer, the customer reference in each ticket (t.customer) points back to the same customer (self).
- The self.tickets -> size() >= 0 ensures that the number of tickets associated with the customer is greater than or equal to zero, indicating that the customer can have zero or more tickets.

15. Every person should be associated with the plan from the same city and same country.

context Person
inv:
self.Services -> notEmpty ( ) and
self.Services.Plan <> null and
self.Services.Plan.city = self.city and
self.services.Plan.country = self.country

Explanation:

- This OCL rule applies to instances of the Person class.
- The invariant self.Services -> notEmpty() ensures that the person has at least one service associated with them.
- The self.Services.Plan <> null ensures that the plan associated with the person's service is not null.
- The self.Services.Plan.city = self.city and self.Services.Plan.country = self.country checks that the city and country of the plan associated with the person's service matches the person's city and country.

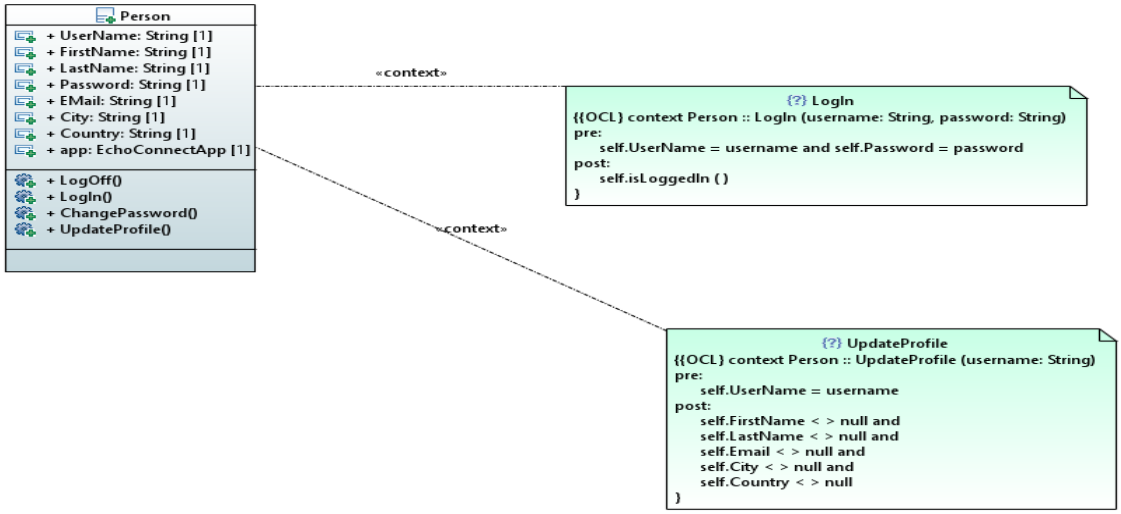

Figure 24 OCLs for Person Class

16. A person should be able to login with a valid username and password.

context Person :: LogIn (username: String, password: String)
pre:
self.UserName = username and self.Password = password
post:
self.isLoggedIn ( )

Explanation:

- This OCL rule is applies to the Login functionality of the Person class, taking a username and password as parameters.
- The precondition self.UserName = username and self.Password = password ensures that the login attempt matches the stored username and password in the Person instance.
- The postcondition self.isLoggedIn() states that if the login is successful (isLoggedIn() returns true).

17. A person should be able to update his/her profile details

context Person :: UpdateProfile (username: String)
pre:
self.UserName = username
post:
self.FirstName < > null and
self.LastName < > null and
self.Email < > null and
self.City < > null and
self.Country < > null

Explanation:

- This OCL rule applies to the UpdateProfile operation of the Person class, taking a username as a parameter.
- The precondition self.UserName = username ensures that the update operation is performed by the person whose username matches the provided username.
- The postcondition self.FirstName <> null, self.LastName <> null, self.Email <> null, self.City <> null, and self.Country <> null ensure that after the update, these attributes are not null, indicating that the profile update includes valid information for these fields.

18. Cancel All Services

context services :: cancelAllServices ( )
inv:
pre: self.R2 -> notempty ( )
post: self.R2 -> empty ()

Explanation:

- This OCL rule applies to the cancelAllServices operation in the services class.

- The precondition is self.R2 -> notEmpty() ensures that the collection R2(association between services and person) is not empty before the operation is executed indicating there should be atleast one service in the Myservices List.
- The postcondition self.R2 -> isEmpty() specifies that after executing the operation, the collection R2 should be empty, indicating that all services have been canceled.

19. Subscribe service function

context services :: subscribe (ServiceID)
inv:
pre: self.R2 -> excludes (serviceID)
post: self.R2 -> includes (serviceID)

Explanation**:**

- This OCL rule applies to the subscribe operation in the services context, taking a serviceID as a parameter.
- The precondition is self.R2 -> excludes(serviceID) ensures that the serviceID is not already included in the collection R2 before the operation is executed.
- The postcondition is self.R2 -> includes(serviceID) specifies that after executing the operation, the serviceID should be included in the collection R2, indicating that the subscription has been successfully processed.

20. Unsubscribe service function.

context services :: unsubscribe (ServiceID)
inv:
pre: self.R2 -> includes (ServiceID)
pos: self.R2 -> excludes (ServiceID)

Explanation**:**

- The context services::unsubscribe(serviceID) specifies that this OCL rule applies to the unsubscribe operation in the services context, taking a serviceID as a parameter.
- The precondition is self.R2 -> includes(serviceID) ensures that the serviceID is included in the collection R2 before the operation is executed.
- The postcondition self.R2 -> excludes(serviceID) specifies that after executing the operation, the serviceID should be excluded from the collection R2, indicating that the unsubscription has been successfully processed.

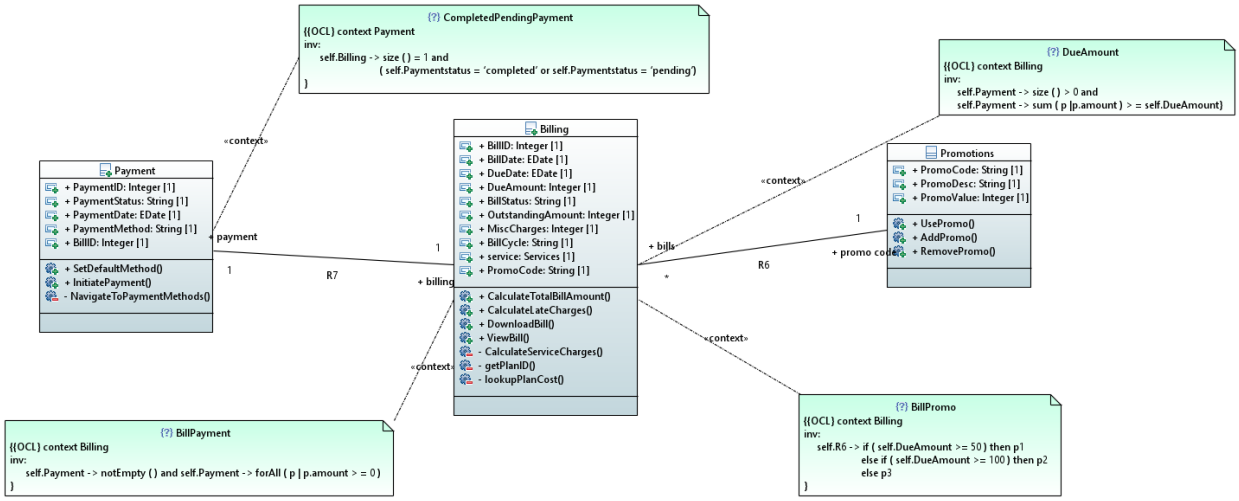

Figure 25 OCLs for Payment, billing and Promotions Classes

21. Billing and Promo

context Billing
inv:
self.R6 -> if (self.DueAmount >= 50) then p1
else if (self.DueAmount >= 100) then p2
else p3

Explanation:

- This OCL rule is applied to the instances of Billing class
- The invariant self.R6 which is the promo code will be assigned according to the due amount in the billing class:
- If the due amount is greater than or equal to 50 but less than 100, p1 is applied.
- If the due amount is greater than or equal to 100, p2 is applied. Otherwise, p3 is applied.

22. Ensure that each billing record is associated with at least one payment and the payment amount is greater than and equal to zero.

context Billing
inv:
self.Payment -> notEmpty ( ) and self.Payment -> forAll (p | p.amount > = 0)

Explanation:

- This OCL constraint applies to instances of the Billing class.

- self.payment->notEmpty() ensures that the Billing class has at least one associated Payment. self.payment refers to the collection of associated Payment instances for the current BillingRecord, and ->notEmpty() checks that this collection is not empty.
- self.payment->forAll(p | p.amount >= 0): This part of the constraint uses forAll to iterate over all Payment instances associated with the current BillingRecord. It checks that the amount attribute of each Payment is greater than or equal to zero.

23. Make sure that each payment is associated with exactly one billing record and the payment status is 'completed/pending'.

context Payment
inv:
self.Billing -> size ( ) = 1 and
(self.Paymentstatus = 'completed' or self.Paymentstatus = 'pending')

Explanation:

- This OCL constraint applies to instances of the Payment class.
- self.billingRecord->size() = 1: Ensures that each Payment is associated with exactly one BillingRecord. self.billingRecord refers to the associated BillingRecord instance, and ->size() = 1 checks that there is exactly one such instance.
- (self.status = 'completed' or self.status = 'pending'): Checks that the status attribute of the Payment is either 'completed' or 'pending'.

24. Enforce that each billing record can have multiple associated payments, and the total payment amount is equal to or greater than the billing amount.

context Billing
inv:
self.Payment -> size ( ) > 0 and
self.Payment -> sum (p |p.amount) > = self.DueAmount

Explanation:

- This OCL constraint applies to instances of the Billing class
- self.payment->size() > 0: Ensures that there are multiple associated Payment instances for the BillingRecord.
- self.payment->sum(p | p.amount) >= self.amount: This part calculates the sum of all payment amounts (p.amount) associated with the BillingRecord and checks if it's greater than or equal to the amount attribute of the BillingRecord itself. This ensures that the total payment amount covers the billing amount.

25. Ensure that each support interaction is associated with at least one employee, and the interaction type is either 'phone' or 'email'.

context Support
inv:
self.Employee -> notEmpty ( ) and (self. type = 'phone' or self.type = 'email)

Explanation:

- This OCL constraint applies to instances of the Support class
- self.Employee->notEmpty(): Ensures that there is at least one associated Employee for the Support Operation.
- (self.type = 'phone' or self.type = 'email'): Checks that the type attribute of the Support instance is either 'phone' or 'email'.

26. Make sure that each billing record is associated for each and every service.

context Services
inv:
allInstances -> forAll (service | Billing.allInstances ( ). Bills -> exists (bill | bill.services = service)

Explanation:

- This OCL constraint applies to all instances of the Service class.
- self.services: This refers to the collection of services associated with a particular Bill instance
- size(): This navigates to the collection of services and retrieves its size.

27. Enforce that each service can have multiple associated billing records.

context Service
inv:
self.Billing -> notEmpty ( )

Explanation:

- This OCL constraint applies to all instances of the Service class.
- self.bills: This refers to the collection of bills associated with a particular Service instance.
- notEmpty(): This navigates to the collection of bills and checks if it's not empty.

28. Overdue Payment Notification Sent Only After Due Date

context Billing
inv:
self.Notifications -> select (n | n.type = ‘ OverdueNotification’)
-> forAll (self.DueDate < self.CurrentDate)

Explanation:

- This OCL constraint applies to all instances of the Billing class.
- The provided OCL (Object Constraint Language) expression is designed to select all notifications of type 'OverdueNotification' from a collection of notifications (self.Notifications) and then check if the due date of each notification is earlier than the current date (self.CurrentDate).
- This is essentially a way to filter out notifications that are overdue, based on their due date being earlier than the current date.

29. This constraint ensures a payment can only be made using a valid card.

context Payment
inv:
self.Card -> notEmpty ( ) and self.Card.isValid ( )

Explanation:

- This OCL constraint applies to all instances of the Billing class.
- self.card->notEmpty(): Checks if the payment has an associated card.
- self.card.isValid(): Navigates to the card and calls an assumed isValid() method to validate the card information.

30. Constraint ensuring unique serviceIDs within the Services class.

context Services
inv:
allInstances -> forAll (s1, s2 | s1 < > s2 implies s1.serviceID < > s2.serviceID)

Explanation:

- This OCL constraint applies to all instances of the Services class.
- This OCL condition is condition being checked for each pair of instances. It reads as "if s1 is not equal to s2, then s1's serviceID must not be equal to s2's serviceID."

31. **Constraint ensuring that the plan associated with a service exists in the plans list in Services.**

context Services
inv:
allInstances -> forAll (service | self.plans -> invludes (service.plan))

Explanation:

- This OCL constraint applies to all instances of the Services class.
- self.services: Refers to the collection of instances of the Services class.
- forAll(service | ...): This iterates over each service instance in the collection.
- self.plans->includes(service.plan): This is the condition being checked for each service. It reads as "the plans list of Services must include the plan associated with the service."

32. **Constraint ensuring that the startdate is before the enddate for each service in Services.**

context Services
inv:
allInstances -> forAll (service | service.StartDate < service.EndDate)

Explanation:

- This OCL constraint applies to all instances of the Services class.
- self.services: Refers to the collection of instances of the Services class.
- forAll(service | ...): This iterates over each service instance in the collection.
- service.startdate < service.enddate: This is the condition being checked for each service. It ensures that the start date of the service is before the end date.

33. **Constraint ensuring that the paymentremainder() method is only called for bills with a billstatus of "Unpaid".**

context Notifications :: paymentRemainder ( )
pre:
self.Billing.BillStatus = 'Unpaid'

Explanation:

- This OCL constraint applies to the paymentremainder() method within the Notifications class.

- pre: This keyword indicates a precondition, which is a condition that must be true before the execution of the method.
- self.bill.billstatus = 'Unpaid': This is the precondition being specified. It reads as "the bill status of the associated bill must be 'Unpaid'.

34. Post condition ensuring that the successfulpayment() method updates the Bill Status to "Paid" and sets the Due Amount to zero

context Notifications :: successfulpayment ( )
post:
self.Billing.BillStatus = 'Paid' and self.Billing.DueAmount = 0

Explanation:

- This OCL constraint applies to the successfulpayment () method within the Notifications class.
- post: This keyword indicates a postcondition, which is a condition that must be true after the execution of the method.
- self.bill.BillStatus = 'Paid' and self.bill.DueAmount = 0: This is the postcondition being specified. It reads as "after the successful payment, the bill status must be 'Paid' and the due amount must be 0."

35. Make sure that each preference is associated with exactly one customer.

context Customer
inv:
allInstances -> forAll (p1, p2 | p1 < > p2 implies p1.Customer < > p2. Customer)

Explanation:

- pairs of preferences (p1, p2) associated with the customer, if they are different (p1 <> p2), then their associated customers (p1.customer and p2.customer) must also be different, ensuring that each preference is associated with exactly one customer.
- This constraint ensures that no two preferences associated with the same customer can have the same customer associated with them.

36. Constraint ensuring unique Username within the Person class.

context Person
inv:
allInstances -> forAll (p1, p2 | p1 < > p2 implies p1.UserName < > p2. UserName)

Explanation:

- This OCL constraint applies to all instances of the Person class.
- This OCL condition is condition being checked for each pair of instances. It reads as "if p1 is not equal to p2, then p1's UserName must not be equal to p2's UserName."

37. Constraint ensuring unique PreferenceID within the Preferences class.

context Preferences
inv:
allInstances -> forAll (p1, p2 | p1 < > p2 implies p1. PreferenceID < > p2.PreferenceID)

Explanation:

- This OCL constraint applies to all instances of the Preferences class.
- This OCL condition is condition being checked for each pair of instances. It reads as "if p1 is not equal to p2, then p1's PreferenceID must not be equal to p2's PreferenceID."

38. Constraint ensuring unique EmployeeID within the Employee class

context Employee
inv:
allInstances -> forAll (e1, e2 | e1 < > e2 implies e1.EmployeeID < > e2.EmployeeID)

Explanation:

- This OCL constraint applies to all instances of the Employee class.
- This OCL condition is condition being checked for each pair of instances. It reads as "if e1 is not equal to e2, then e1's EmployeeID must not be equal to e2's EmployeeID."

39. Constraint ensuring unique CustomerID within the Customer class

context Customer
inv:
allInstances -> forAll (c1, c2 | c1 < > c2 implies c1.CustomerID < > c2. CustomerID)

Explanation:

- This OCL constraint applies to all instances of the Customer class.
- This OCL condition is condition being checked for each pair of instances. It reads as "if c1 is not equal to c2, then c1's CustomerID must not be equal to c2's CustomerID."

40. Constraint ensuring unique BillID within the Billing class

context Billing
inv:
allInstances -> forAll (b1, b2 | b1 < > b2 implies b1.BillID < > b2. BillID)

Explanation:

- This OCL constraint applies to all instances of the Billing class.
- This OCL condition is condition being checked for each pair of instances. It reads as "if b1 is not equal to b2, then b1's BillID must not be equal to b2's BillID."

41. Constraint ensuring unique PaymentID within the Payment class

context Payment
inv:
allInstances -> forAll (p1, p2 | p1 < > p2 implies p1. PaymentID < > p2. PaymentID)

Explanation:

- This OCL constraint applies to all instances of the Payment class.
- This OCL condition is condition being checked for each pair of instances. It reads as "if p1 is not equal to p2, then p1's PaymentID must not be equal to p2's PaymentID."

42. Constraint ensuring unique TicketID within the Ticket class

context Ticket
inv:
allInstances -> forAll (t1, t2 | t1 < > t2 implies t1. TicketID < > t2. TicketID)

Explanation:

- This OCL constraint applies to all instances of the Ticket class.
- This OCL condition is condition being checked for each pair of instances. It reads as "if t1 is not equal to t2, then t1's TicketID must not be equal to t2's TicketID."

43. Constraint ensuring unique WalletID within the Wallet class

context Wallet
inv:
allInstances -> forAll (w1, w2 | w1 < > w2 implies w1. WalletID < > w2. WalletID)

Explanation:

- This OCL constraint applies to all instances of the Wallet class.
- This OCL condition is condition being checked for each pair of instances. It reads as "if w1 is not equal to w2, then w1's WalletID must not be equal to w2's WalletID."

44. Constraint ensuring unique PromoCode within the Promotions class

context Promotions
inv:
allInstances -> forAll (p1, p2 | p1 < > p2 implies p1. PromoCode < > p2. PromoCode)

Explanation:

- This OCL constraint applies to all instances of the Promotion class.
- This OCL condition is condition being checked for each pair of instances. It reads as "if p1 is not equal to p2, then p1's PromoCode must not be equal to p2's PromoCode."

45. Constraint ensuring unique CardNo within the Card class

context Card
inv:
allInstances -> forAll (c1, c2 | c1 < > c2 implies c1. CardNo < > c2. CardNo)

Explanation:

- This OCL constraint applies to all instances of the Card class.
- This OCL condition is condition being checked for each pair of instances. It reads as "if c1 is not equal to c2, then c1's CardNo must not be equal to c2's CardNo."

Implementation of Class Diagram and OCL Constraints with Papyrus Eclipse

We updated the EchoConnect Papyrus project within the Eclipse and created new OCL diagrams. We have added OCL constraints using Papyrus editor.

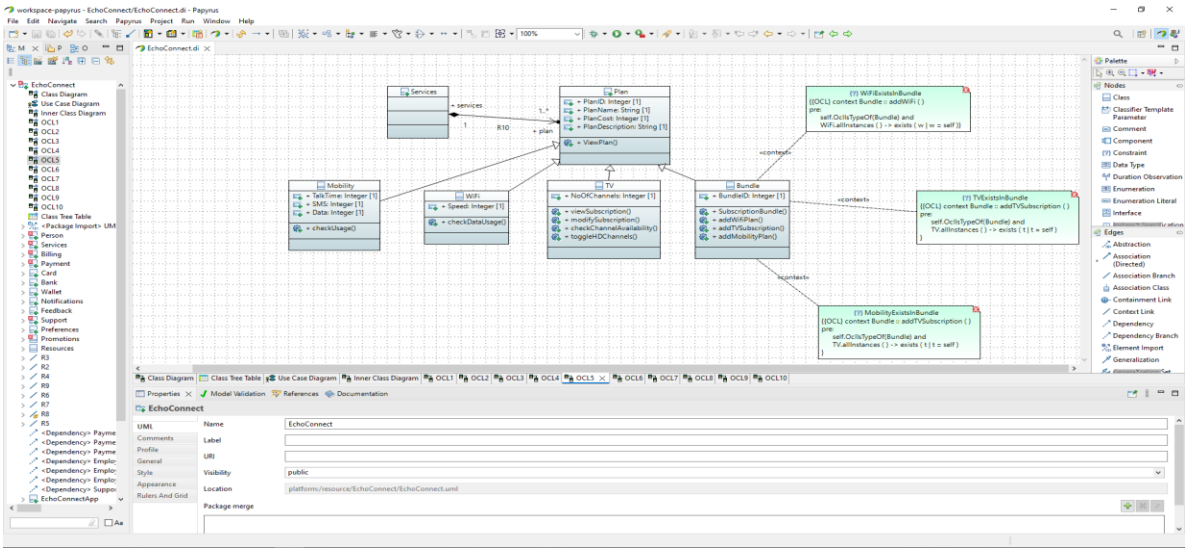

Figure 26 Papyrus Window for OCL Constraints

Later using the updated class diagram and OCL constraints, we generated code for the system. Papyrus does not inherently generate full code, but it can be integrated with code generation plugins or tools. But we used the code as starting point and developed our functionalities.

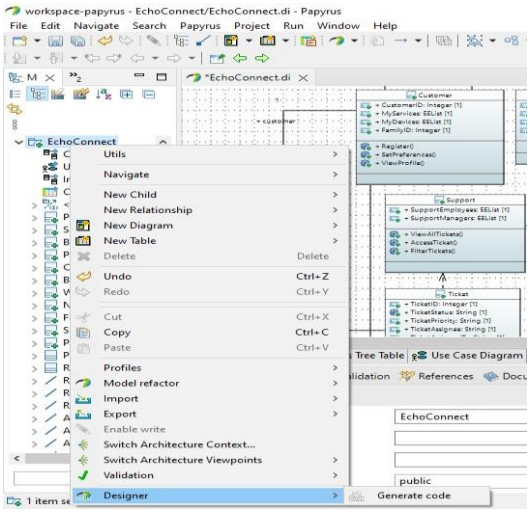

Figure 27 Code Generation

Implementation of Test Cases and Results

Unit testing is a crucial aspect of software development that involves testing individual units or components of a software system in isolation to ensure if the functionalities align with the requirements and the system works as expected.

The project is tested using Junit testing. The test cases have been built and tested successfully without any errors or failures and the results are shown below:

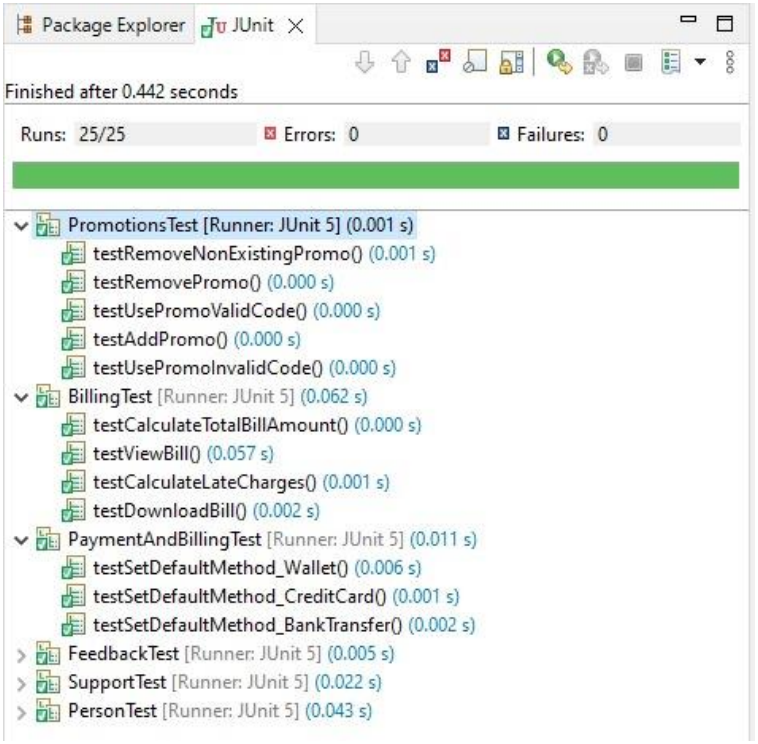

Figure 28 Test Cases Execution

The test cases should cover the code functionality for validating requirements, identifying bugs, preventing regressions, and facilitating code refactoring. This will improve code quality and boost collaboration, ultimately ensuring a stable and reliable software system.

State Diagrams

A state diagram is used to represent the state of a system or part of a system at a finite point in time. It's a behavioral diagram and uses finite state transitions to show how things behave. The state of an object describes a set of values for its attributes at a given point in time. An object remains in a state until it receives events; it then transitions to another state if necessary. A transition represents a change in the state of an object in response to a signal (or an event). A state transition table is a tabular representation of the possible transitions between states in a system that is the result of various events or actions. Each row of the table corresponds to a state, and each column of the table represents an event or an action. The cells of the table indicate the possible transitions or the effects that each event will have on the state of the system.

Here we have tried to put together and represent different states and transitions in our application.

Each diagram is represented along with a state transition table and code implementation snapshot that shows the implementation of functionalities and state transitions in the Java Code.

The state diagrams and code snapshots here may not be visible clearly since they are compressed to fit, Please Zoom to inspect the diagram clearly.

Tickets State Diagram:

This state diagram represents a ticket management system where users can perform various actions related to ticket management. Here's the explanation:

LoggedIn: This is the initial state after a user logs in. From here, they can view their profile, raise a ticket, view all tickets, access a specific ticket, assign tickets, set priorities, change ticket status, or reassign tickets.

RaiseTicket: In this state, users can enter ticket details, update details, select priority, and wait for the manager's action. The process allows for entering and managing new tickets.

ViewAll: Users can view all tickets, view details of specific tickets, and go back to the main logged-in state.

TicketStatus: This state represents the lifecycle of a ticket, starting from "New" to "Assigned," then "In Progress," "On Hold/Pending," "Resolved/Closed," and potentially "Reopened" or "Cancelled/Invalid." There's also an option to change the status of a ticket.

Access: Users can access a specific ticket, filter tickets, and go back to the main logged-in state.

Filter: This state allows users to filter tickets based on criteria and navigate back to the access state.

The transitions between states represent the flow of actions a user can take within the ticket management system, from creating and updating tickets to managing their status and accessing specific ticket details.

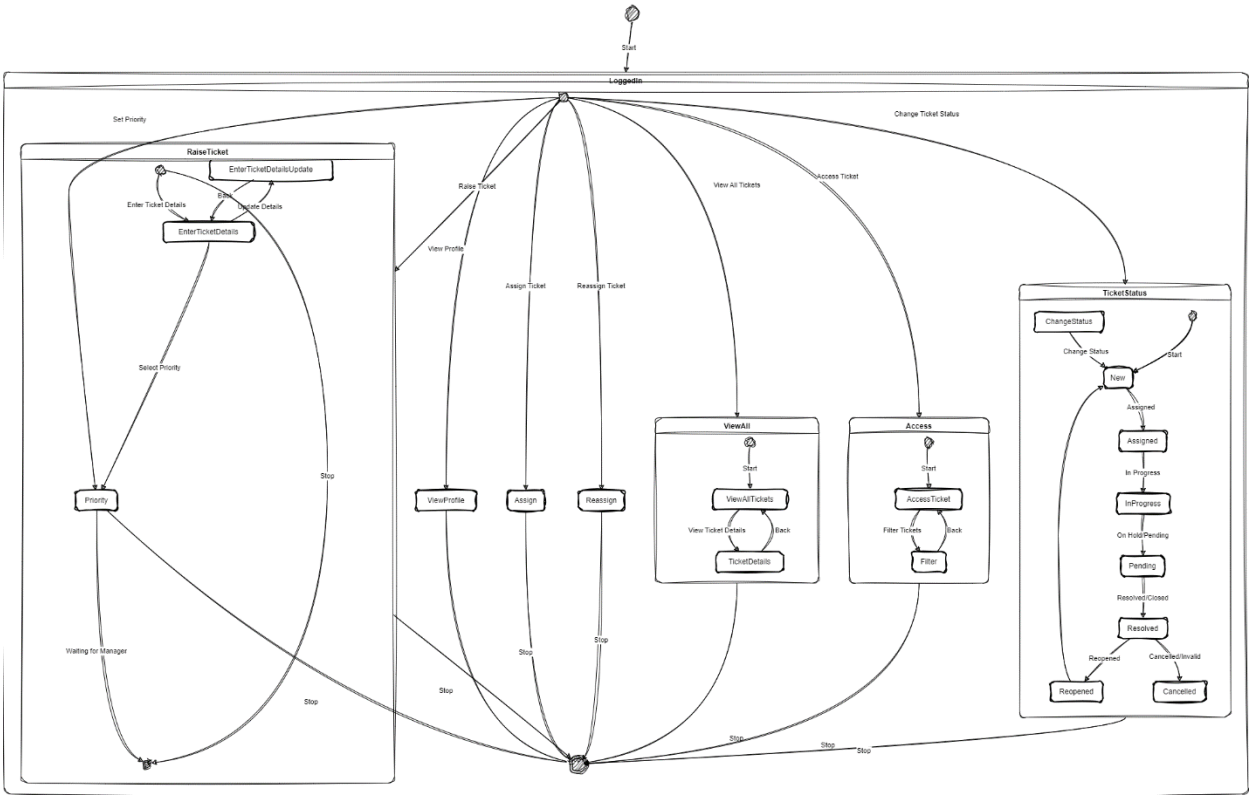

Figure 29 Ticket Status State Diagram

```java
// Code generated by Papyrus Java

package EchoConnect;

public class Ticket {
    public int TicketID;
    public String TicketStatus;
    public String TicketPriority;
    public String TicketAssignee;
    public String TicketAssignedTo;
    public String TicketDescription;

    public Ticket(int ticketID, String ticketStatus, String ticketPriority,
            String ticketAssignee, String ticketAssignedTo, String ticketDescription) {
        this.TicketID = ticketID;
        this.TicketStatus = ticketStatus;
        this.TicketPriority = ticketPriority;
        this.TicketAssignee = ticketAssignee;
        this.TicketAssignedTo = ticketAssignedTo;
        this.TicketDescription = ticketDescription;
}
    /**
     * Getters and Setters
     */
    // Getters and Setters
    public int getTicketID() {
        return ticketID;
    }

    public void setTicketID(int ticketID) {
        this.ticketID = ticketID;
    }

    public String getTicketStatus() {
        return ticketStatus;
    }

    public void setTicketStatus(String ticketStatus) {
        this.ticketStatus = ticketStatus;
    }

    public String getTicketPriority() {
        return ticketPriority;
    }

    public void setTicketPriority(String ticketPriority) {
        this.ticketPriority = ticketPriority;
    }

    public String getTicketAssignee() {
        return ticketAssignee;
    }

    public void setTicketAssignee(String ticketAssignee) {
        this.ticketAssignee = ticketAssignee;
    }

    public String getTicketAssignedTo() {
        return ticketAssignedTo;
    }

    public void setTicketAssignedTo(String ticketAssignedTo) {
        this.ticketAssignedTo = ticketAssignedTo;
    }

    public String getTicketDescription() {
        return ticketDescription;
    }

    public void setTicketDescription(String ticketDescription) {
        this.ticketDescription = ticketDescription;
    }

    /**
     * Functionalities for Ticket State Diagram
     */
    public void raiseTicket(int ticketID, String ticketPriority, String ticketDescription) {
        this.ticketID = ticketID;
        this.ticketPriority = ticketPriority;
        this.ticketDescription = ticketDescription;
        this.ticketStatus = "New";
        System.out.println("New support ticket raised!");
    }

    public void assignTicket(String assignee) {
        this.ticketAssignee = assignee;
        this.ticketStatus = "Assigned";
        System.out.println("Ticket assigned to: " + assignee);
    }

    public void changeTicketStatus(String newStatus) {
        this.ticketStatus = newStatus;
        System.out.println("Ticket status changed to: " + newStatus);
    }

    public static void viewallTickets(List<Ticket> tickets) {
        for (Ticket ticket : tickets) {
            System.out.println("Ticket ID: " + ticket.getTicketID());
            System.out.println("Status: " + ticket.getTicketStatus());
            System.out.println("Priority: " + ticket.getTicketPriority());
            System.out.println("Assignee: " + ticket.getTicketAssignee());
            System.out.println("Assigned To: " + ticket.getTicketAssignedTo());
            System.out.println("Description: " + ticket.getTicketDescription());
            System.out.println("----------------------------------------");
        }
    }

    public void setPriority(String priority) {
        this.ticketPriority = priority;
        System.out.println("Priority set to: " + priority);
    }

    public void startProgress() {
        if (ticketStatus.equals("Assigned")) {
            ticketStatus = "InProgress";
            System.out.println("Ticket " + ticketID + " is now in progress.");
        } else {
            System.out.println("Ticket must be assigned before starting progress.");
        }
    }

    public void markPending() {
        if (ticketStatus.equals("InProgress")) {
            ticketStatus = "Pending";
            System.out.println("Ticket " + ticketID + " marked as pending.");
        } else {
            System.out.println("Ticket must be in progress before marking as pending.");
        }
    }

    public void resolveTicket() {
        if (ticketStatus.equals("InProgress") || ticketStatus.equals("Pending")) {
            ticketStatus = "Resolved";
            System.out.println("Ticket " + ticketID + " resolved.");
        } else {
            System.out.println("Ticket must be in progress or pending to be resolved.");
        }
    }

    public void cancelTicket() {
        if (!ticketStatus.equals("Resolved") && !ticketStatus.equals("Cancelled")) {
            ticketStatus = "Cancelled";
            System.out.println("Ticket " + ticketID + " cancelled.");
        } else {
            System.out.println("Ticket is already resolved or cancelled.");
        }
    }
}
```

Figure 30 Code Snapshot of Ticket Status Implementation

Table 4 Ticket State Transition Table

State/Event	Assign	Inprogress	OnHold	Resolve	Reopen	Cancel
New	Assigned	New	New	New	New	New
Assigned	Assigned	Inprogress	Assigned	Assigned	Assigned	Assigned
Inprogress	Inprogress	Inprogress	Pending	Inprogress	Inprogress	Inprogress
Pending	Pending	Pending	Pending	Resolved	Pending	Pending
Resolved	Resolved	Resolved	Resolved	Resolved	Reopened	Cancelled

Service/ Plan State Diagram:

This state diagram represents a customer's interaction with a subscription service platform. Since different classes are involved, we have shown the state diagram with abstraction. Here's a description of the states and transitions:

Logged In: This is the initial and final state of the interaction, representing a logged-in customer on the platform.

ViewProfile: Allows the customer to view their profile information.

Service: Provides details about a specific service, such as viewing service details, subscribing, unsubscribing, upgrading, and downgrading.

Plan: Displays details about subscription plans for each service category (TV, WiFi, Bundle, Mobility).

Service Plan Details (TVPlanDetails, WiFiPlanDetails, BundleDetails, MobilityPlanDetails): These states provide specific actions related to each service plan, such as modifying subscriptions, checking channel availability, toggling HD channels, checking data usage, modifying bundles, adding, or removing services, etc.

Subscribe, Unsubscribe, Upgrade, Downgrade: These states allow customers to perform actions related to their subscriptions, such as subscribing to a new service, unsubscribing from a service, upgrading, or downgrading their current service plans.

Cancel All: Provides an option to cancel all services if the customer has active subscriptions. If no services are active, it informs the customer that there are no services to cancel.

Overall, the state diagram outlines the customer's journey in managing their subscriptions, from viewing and modifying services to handling subscriptions across different service categories.

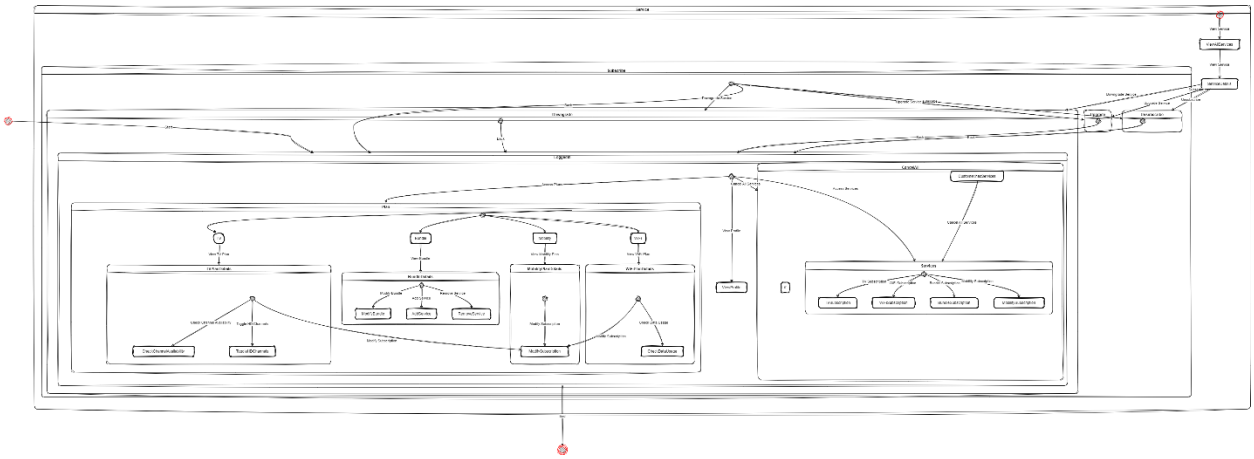

Figure 31 Services State Diagram

```
// ----------------------------------------
// Code generated by Papyrus Java
// ----------------------------------------

package EchoConnect;

import java.util.Date;
import java.util.List;
import EchoConnect.Person.Customer;

public class Services {

    private int serviceID;
    private String serviceName;
    private java.util.Date startDate;
    private java.util.Date endDate;

    public static void viewAllServices() {
        Services mobilityPlan = new Services(1, "MobilityPlan", new java.util.Date(), new java.util.Date());
        Services wifi = new Services(2, "WiFi", new java.util.Date(), new java.util.Date());
        Services tv = new Services(3, "TV", new java.util.Date(), new java.util.Date());
        Services bundle = new Services(4, "Bundle", new java.util.Date(), new java.util.Date());

        System.out.println("All Services:");
        System.out.println("-------------------------");
        printServiceDetails(mobilityPlan);
        printServiceDetails(wifi);
        printServiceDetails(tv);
        printServiceDetails(bundle);
        System.out.println("-------------------------");
    }

    public void subscribe(String serviceName, List<String> myServices) {
        System.out.println("Subscribed to " + serviceName);
        myServices.add(serviceName);
    }

    public void unsubscribe(String serviceName, List<String> myServices) {
        System.out.println("Unsubscribed from " + serviceName);
        myServices.remove(serviceName);

    }
    public static void viewService(String serviceName) {
        System.out.println("Viewing details for service: " + serviceName);
        printServiceDetails(serviceName);
    }
```

```
package EchoConnect;

public class Mobility extends EchoConnect.Plan{
    public int planCost;
    public String planDescription;
    public String planName;
    public int talkTime;
    public int sms;
    public int data;
    public int planID;


    // Function to view plan based on PlanID and PlanName
    public void viewPlan(int inputPlanID, String inputPlanName) {
        if (this.planID == inputPlanID && this.planName.equals(inputPlanName)) {
            System.out.println("Plan ID: " + planID);
            System.out.println("Plan Name: " + planName);
            System.out.println("Plan Cost: " + planCost);
            System.out.println("Plan Description: " + planDescription);
            System.out.println("Talk Time: " + talkTime);
            System.out.println("SMS: " + sms);
            System.out.println("Data: " + data);
        } else {
            System.out.println("Plan not found.");
        }
    }
    public void modifySubscription(int newPlanCost, String newPlanDescription, int newTalkTime, int newSMS, int newData) {
        System.out.println("Modifying Mobility Plan...");
        this.planCost = newPlanCost;
        this.planDescription = newPlanDescription;
        this.talkTime = newTalkTime;
        this.sms = newSMS;
        this.data = newData;
        System.out.println("Subscription modified successfully.");
    }
```

```
package EchoConnect;

import java.util.Date;

public class TV extends EchoConnect.Plan {
    public int planCost;
    public String planDescription;
    public String planName;
    public int planID;
    private int numberOfChannels;
    private boolean hdChannels;
    // Function to view subscription details based on planID and planName
    public void viewSubscription(int inputplanID, String inputplanName) {
        if (this.planID == inputplanID && this.planName.equals(inputplanName)) {
            System.out.println("Plan ID: " + planID);
            System.out.println("Plan Name: " + planName);
            System.out.println("Plan Cost: " + planCost);
            System.out.println("Plan Description: " + planDescription);
            System.out.println("Number of Channels: " + numberOfChannels);
            System.out.println("HD Channels Included: " + (hdChannels ? "Yes" : "No"));
        } else {
            System.out.println("Subscription not found.");
        }
    }
    // Function to modify subscription details
    public void modifySubscription(int newplanCost, String newplanDescription, int newNumberOfChannels, boolean newHdChannels) {
        this.planCost = newplanCost;
        this.planDescription = newplanDescription;
        this.numberOfChannels = newNumberOfChannels;
        this.hdChannels = newHdChannels;
        System.out.println("Subscription details updated successfully.");
    }

    // Function to check channel availability
    public void checkChannelAvailability(String channelName) {
        // For demonstration, let's assume you have a list of channels stored in an array.
        String[] availableChannels = {"CNN", "BBC", "ESPN", "Discovery", "National Geographic", "HBO"};

        // Check if the provided channelName exists in the availableChannels array
        for (String channel : availableChannels) {
            if (channel.equalsIgnoreCase(channelName)) {
                System.out.println("Channel '" + channelName + "' is accessible with this subscription.");
                }
        }

        // If channelName is not found in availableChannels array
        System.out.println("Channel '" + channelName + "' is not accessible with this subscription.");
    }
    // Function to toggle HD channels
    public void toggleHDChannels() {
        this.hdChannels = !this.hdChannels;
        System.out.println("HD Channels toggled: " + (hdChannels ? "On" : "Off"));
    }
```

Figure 32 Code Snapshot of Services Implementation (Part I)

```
package EchoConnect;

public class WiFi extends EchoConnect.Plan
{
    public int planCost;
    public String planDescription;
    public String planName;
    public int planID;
    public int speed;
    public int dataLimit;

    // Function to view plan based on PlanID and PlanName
    public void viewPlan(int inputPlanID, String inputPlanName) {
        if (this.planID == inputPlanID && this.planName.equals(inputPlanName)) {
            System.out.println("Plan ID: " + planID);
            System.out.println("Plan Name: " + planName);
            System.out.println("Plan Cost: " + planCost);
            System.out.println("Plan Description:  " + planDescription);
            System.out.println("Speed: " + speed + " Mbps");
            System.out.println("Data Limit: " + dataLimit + " GB");
        } else {
            System.out.println("Plan not found.");
        }
    }

     // Function to check data usage
    public void checkDataUsage(int usage) {
        if (usage >= this.dataLimit) {
            System.out.println("You have exceeded the data limit. Additional charges may apply.");
        } else {
            System.out.println("Data usage within limits.");
        }
    }
```

```
package EchoConnect;

import java.util.ArrayList;

public class Bundle extends EchoConnect.Plan {
    public Bundle(int serviceID, String serviceName, Date startDate, Date endDate) {
        super(serviceID, serviceName, startDate, endDate);
        // TODO Auto-generated constructor stub
    }
    private List<WiFi> wifiPlans;
    private List<TV> tvSubscriptions;
    private List<Mobility> mobilityPlans;
    public int planCost;

    public void SubscriptionBundle() {
        this.wifiPlans = new ArrayList<>();
        this.tvSubscriptions = new ArrayList<>();
        this.mobilityPlans = new ArrayList<>();
    }
    public void addWifiPlan(WiFi wifiPlan) {
        this.wifiPlans.add(wifiPlan);
    }
    public void addTVSubscription(TV tvSubscription) {
        this.tvSubscriptions.add(tvSubscription);
    }
    public void addMobilityPlan(Mobility mobilityPlan) {
        this.mobilityPlans.add(mobilityPlan);
    }
```

Figure 33 Code Snapshot of Services Implementation (Part II)

Table 5 Services State Transition Table

State\ Event	Subscribe	Unsubscribe	Upgrade	Downgrade	CancelAll
LoggedIn	Subscribed	Subscribed	LoggedIn	LoggedIn	LoggedIn
Subscribed	Subscribed	Not Subscribed	Subscribed	Subscribed	Not Subscribed
Not Subscribed	Subscribed	Not Subscribed	Not Subscribed	Not Subscribed	Not Subscribed

Billing/ Payment/ Notifications State Diagram:

This state diagram represents the flow of actions and events in a system, likely related to a user's interaction with a service or application. Here's an explanation of the different states and transitions in the diagram:

LoggingIn: Upon successful login, the user transitions to the LoggingIn state.

GeneratingBill: If the user chooses to generate a bill, they move to the GeneratingBill state.

BillGenerated: After the bill is successfully generated, the system moves to the BillGenerated state.

CreatingNotification: Upon bill generation, a notification is created. This state represents the creation of that notification.

NotificationSent: The notification is then sent to the user.

NotificationDelivered: Once the notification is successfully delivered, the system moves to this state.

NotificationRead: If the user reads the notification, the system enters the NotificationRead state.

NotificationFailed: In case the notification delivery fails, the system enters the NotificationFailed state.

NotificationRetried: If there's a failure, the system retries sending the notification.

InitiatingPayment: If the user initiates a payment, the system moves to the InitiatingPayment state.

ViewingPaymentDetails: After payment initiation, the user can view payment details.

PaymentSuccess: If the payment is successful, the system enters the PaymentSuccess state.

PaymentFailure: In case of payment failure, the system enters the PaymentFailure state.

ViewingReceipt: After payment, the user can view the receipt.

ActivatingServices: Following payment, the system activates services, moving to the ActivatingServices state.

ViewingBillOverdue: If the bill becomes overdue, the user can view the overdue bill.

ViewingBillingPreferences: The user can also view and edit billing preferences.

EditingBillingPreferences: This state represents the editing of billing preferences.

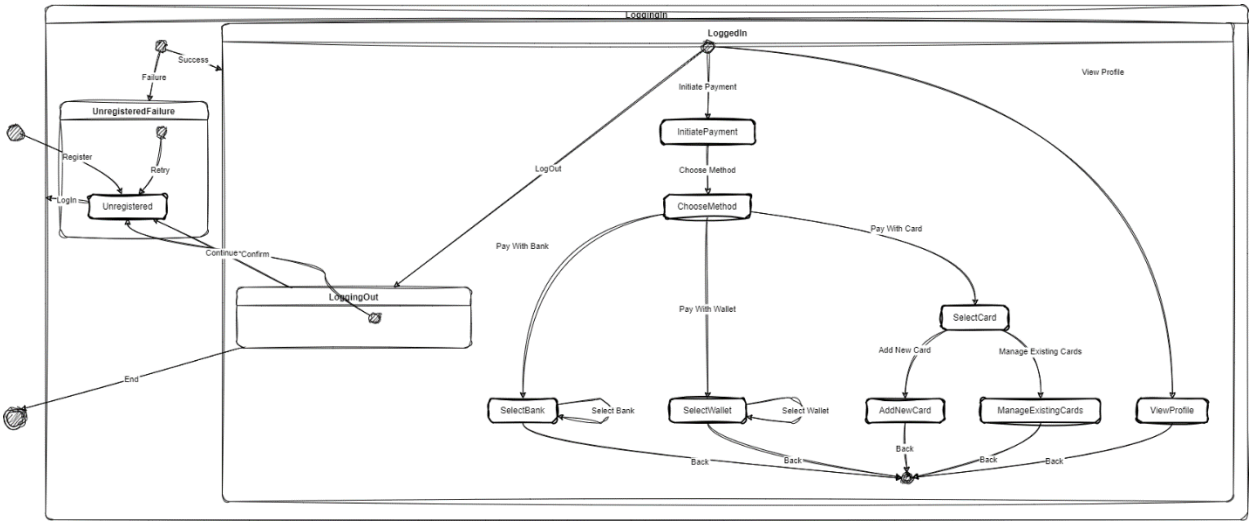

Figure 34 Payment Status State Diagram

```java
// ----------------------------------------------------------
// Code generated by Papyrus Java
// ----------------------------------------------------------

package EchoConnect;

import java.util.Date;
import java.util.Scanner;


public class Payment {
    public int PaymentID;
    public String PaymentStatus;
    public Date PaymentDate;
    public String PaymentMethod;
    public String PaymentPreference;
    /**
     * Setting Default method between card, bank and wallet
     */
    public void SetDefaultMethod() {
        switch (PaymentPreference.toLowerCase())
        {
        case "credit card":
            PaymentMethod = "Credit Card";
            break;
        case "bank transfer":
            PaymentMethod = "Bank Transfer";
            break;
        case "wallet":
            PaymentMethod = "Wallet";
            break;
        default:
            PaymentMethod = "Unknown";
            break;
        }

        System.out.println("Setting default payment method to: "+ PaymentMethod);

    }
```

```java
    /**
     * Initiating Payment
     */
    public void InitiatePayment() {
        //Displaying options to the user
        System.out.println("Choose an option");
        System.out.println("1. Pay Using default payment method");
        System.out.println("2. Go to individual payment methods");

        // Get user input
        Scanner scanner = new Scanner(System.in);
        int userChoice = scanner.nextInt();

        switch(userChoice)
        {
        case 1:
            System.out.println("Paying Using default payment method: "+ PaymentMethod);
            PaymentStatus = "Completed";
            break;
        case 2:
            //Navigate to individual payment methods
            NavigateToPaymentMethods();
            break;
        default:
            System.out.println("Invalid Choice");
        }
    }

    private void NavigateToPaymentMethods() {
        System.out.println("Navigating to individual payment methods menu");
        System.out.println("1. Card");
        System.out.println("2. Wallet");
        System.out.println("3. Bank");

        Scanner scanner = new Scanner(System.in);
        int userChoice = scanner.nextInt();

        switch (userChoice)
        {
        case 1:
            Card.ManageCards();
            break;
        case 2:
            Wallet.RedirectToVendor("WalletVendor");
            PaymentStatus = "Completed";
            break;
        case 3:
            Bank.RedirectToVendor("BankVendor");
            PaymentStatus = "Completed";
            break;
        default:
            System.out.println("Invalid Choice");
        }
```

```java
public class Card {

    public int CardNo;
    public Date CardExpiryDate;
    public int CardCVV;
    public String CardHolderName;

    static void ManageCards() {
        System.out.println("Managing cards and making payments");

        System.out.println("1. Manage Existing Cards");
        System.out.println("2. Add New Card");

        Scanner scanner = new Scanner(System.in);
        int userChoice = scanner.nextInt();

        switch (userChoice) {
        case 1:
            System.out.println("Managing existing cards...");
            break;
        case 2:
            System.out.println("Adding New Card..");
            AddNewCard();
            break;
        default:
            System.out.println("Invalid Choice");
        }
    }

    private static void AddNewCard() {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter Card Number");
        int cardNo = scanner.nextInt();

        System.out.println("Enter Card CVV:");
        int cardCVV = scanner.nextInt();

        System.out.println("Enter Card Holder Name:");
        String cardHolderName = scanner.next();

        System.out.println("New card added successfully!");
    }
```

Figure 35 Payment Status Code Implementation

Table 6 Payment State Transition Diagram

State/ Event	CreatingNotification	PaymentSuccessful	RetryPayment	ViewPaymentDetails
InitiatingPayment	CreateNotification	InitiatingPayment	InitiatingPayment	InitiatingPayment
ViewingPaymentDetails	ViewingPaymentDetails	PaymentSuccess	ViewingPaymentDetails	ViewingPaymentDetails
PaymentSuccess	CreateNotification	PaymentSuccess	PaymentSuccess	PaymentSuccess
PaymentFailure	CreateNotification	PaymentFailure	InitiatingPayment	PaymentFailure
CreateNotification	CreateNotification	CreateNotification	CreateNotification	ViewingPaymentDetails

Table 7 Bill State Transition Table

State/ Event	Login Successful	Bill Generation Complete	Bill Downloaded	Bill Viewed	CreatingNotification	Make a Payment
LoggedIn	Managing Billing	LoggedIn	LoggedIn	LoggedIn	LoggedIn	LoggedIn
ManagingBilling	Managing Billing	BillGenerationComplete	ManagingBilling	ManagingBilling	ManagingBilling	ManagingBilling
GeneratedBill	Generated Bill	GeneratedBill	DownloadBill	GeneratedBill	CreateNotification	InitiatePayment
DownloadBill	Download Bill	DownloadBill	DownloadBill	ViewBill	DownloadBill	DownloadBill

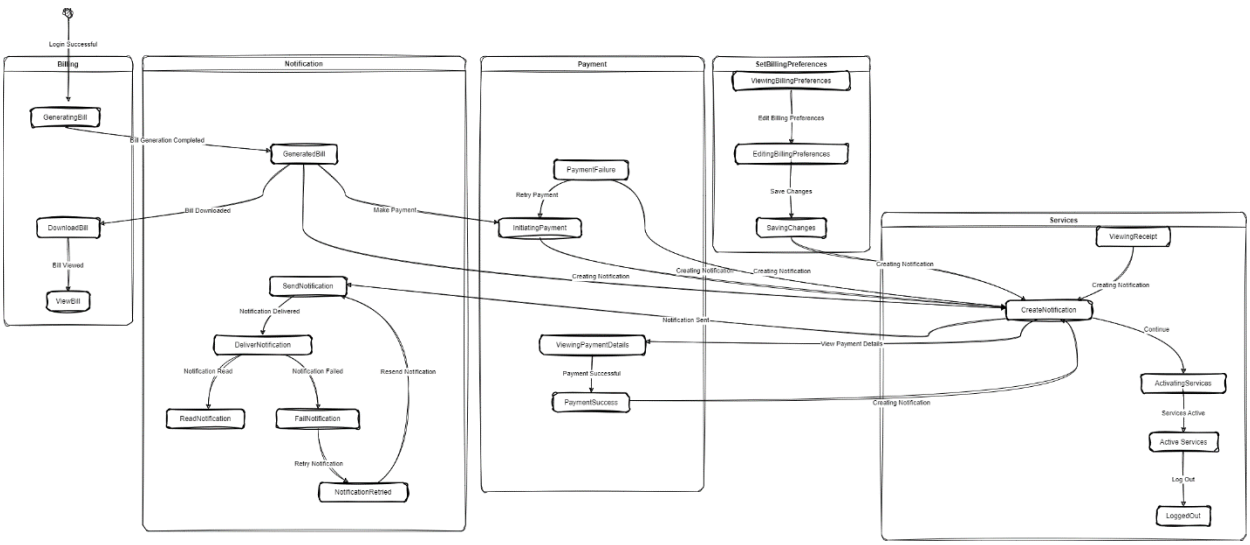

Figure 36 Billing and Payment with Notifications State Diagram

```java
// Code generated by Papyrus Java

package EchoConnect;

import java.time.LocalDate;

/*********************************************************/
/**
 *
 */
public class Notifications {
    private Billing billing;
    private Payment Payment;
    private int lastCustomerId = 0;

    public Notifications(Billing billing, EchoConnect.Payment Payment) {
        this.billing = billing;
        this.Payment = Payment;
    }
    /**
     *
     */
    public void PaymentReminder() {
        LocalDate today = LocalDate.now();
        LocalDate dueDate = billing.getDueDate();
        LocalDate billDate = billing.getBillDate();

            if (billDate.equals(today)) {
                double dueAmount = billing.getDueAmount();
                System.out.println("New bill generation: Your bill of $" + dueAmount + " has been generated.");
            }

            LocalDate remainderDate = dueDate.minusDays(5);
            if (today.equals(remainderDate)) {
                double dueAmount = billing.getDueAmount();
                System.out.println("Payment reminder: Your bill of $" + dueAmount + " is due in 5 days.");
            }
    }

    /**
     *
     */
    public void WelcomeEmail(Customer customer) {
         if (customer != null && customer.getCustomerId() > lastCustomerId) {
                System.out.println("New Customer ID is generated: " + customer.getCustomerId());
                System.out.println("Welcome to Echo Connect: A Telecom Service Company.");
            }
    }

    /**
     *
     */
    public void SuccessfulPayment(Payment payment) {
        if (payment.getPaymentStatus().equals("Success")) {
            System.out.println("Payment Successful,Thank you");
        }
    }
```

Figure 37 Notifications Code Implementation

Table 8 Notifications State Transition Table

State/ Event	Notification Sent	NotificationDelivered	Notification Read	Notification Failed	RetryNotification	ResendNotification
CreateNotification	SendNotification	CreateNotification	CreateNotification	CreateNotification	CreateNotification	CreateNotification
SendNotification	SendNotification	DeliverNotification	SendNotification	SendNotification	SendNotification	SendNotification
DeliverNotification	DeliverNotification	DeliverNotification	ReadNotification	FailNotification	DeliverNotification	DeliverNotification
FailNotification	FailNotification	FailNotification	FailNotification	FailNotification	NotificationRetried	FailNotification
NotificationRetried	NotificationRetried	NotificationRetried	NotificationRetried	NotificationRetried	vNotification Retried	SendNotification

User Profile / Login/Register Module:

OpenApplication: This is the initial state when the user is not logged in.

Login (Login button clicked) leads to the LoggedIn state, indicating a successful login attempt.

Register (Register button clicked) leads to the Registration state for new user registration.

LoggedIn: This state represents a successfully logged-in user.

LoginSuccess (Successful login process) leads to the Home state, indicating a successful login experience.

ViewProfile, EditProfile, SaveChanges, and LogOut represent various actions a logged-in user can perform.

ViewProfileDetails: This state allows the user to view their profile details.

EditProfile (Edit button clicked) leads to the EditProfileDetails state for editing profile information.

EditProfileDetails: In this state, the user can edit their profile details.

SaveChanges (Save button clicked) leads to the SaveProfileChanges state to save the changes made.

SaveProfileChanges: This state represents the process of saving profile changes.

ChangesSaved indicates that the changes have been successfully saved and returns the user to ViewProfileDetails to view the updated information.

Logout: This transition occurs when the user logs out of the system, returning to the initial LoggedOut state.

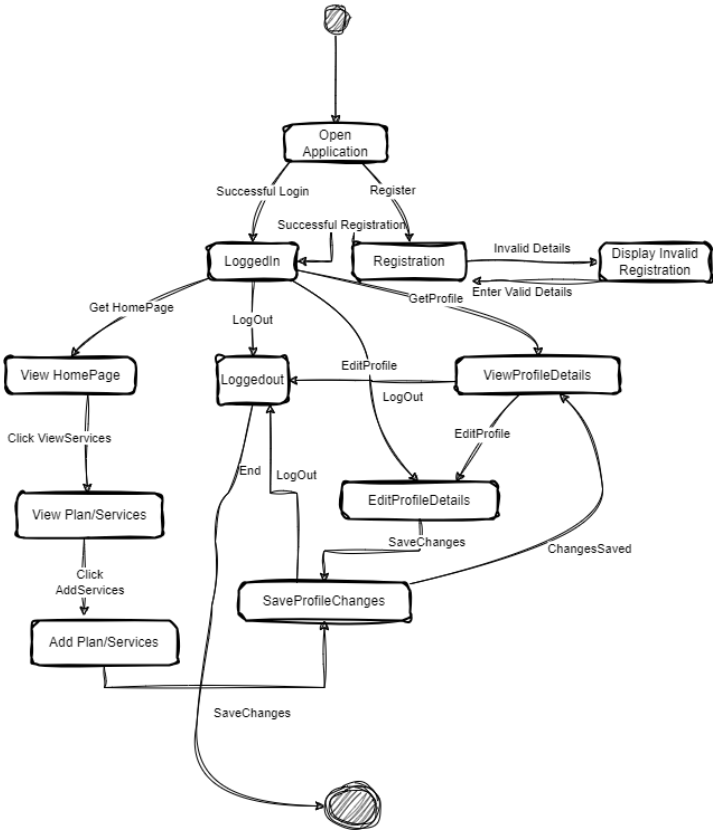

Figure 38 User Profile State Diagram

```
102
103     public boolean LogOff(String username) {
104         if (this.UserName.equals(username)) {
105             System.out.println("Successfully Logged Off!");
106             return true;
107         } else {
108             System.out.println("Invalid username.User logoff failed.");
109             return false;
110         }}
111
112     public boolean LogIn(String username, String password) {
113         if (this.UserName.equals(username)) {
114             if (this.Password.equals(password)) {
115                 System.out.println("Successfully Logged On!");
116                 System.out.println("Welcome " + this.UserName);
117                 return true;
118             } else {
119                 System.out.println("Invalid password. User login failed.");
120                 return false;
121             }
122         } else {
123             System.out.println("Invalid username. User login failed.");
124             return false;
125         }}
126     public void ChangePassword(String username, String currentPassword, String newPassword) {
127         if (this.UserName.equals(username) && this.Password.equals(currentPassword)) {
128             this.Password = newPassword;
129             System.out.println("User password changed successfully!");
130         } else {
131             System.out.println("Invalid username or current password for password change.");
132         }
133     }
134
135     public void UpdateProfile() {
136         if (this.UserName.equals(UserName)) {
137             System.out.println("Current Profile Information:");
138             System.out.println("Username: " + this.UserName);
139             System.out.println("First Name: " + this.FirstName);
140             System.out.println("Last Name: " + this.LastName);
141             System.out.println("Email: " + this.EMail);
142             System.out.println("City: " + this.City);
143             System.out.println("Country: " + this.Country);
144
145             // Get updated information
146             Scanner scanner = new Scanner(System.in);
147             System.out.println("Enter new first name:");
148             this.FirstName = scanner.next();
149             System.out.println("Enter new last name:");
150             this.LastName = scanner.next();
151             System.out.println("Enter new email:");
152             this.EMail = scanner.next();
153             System.out.println("Enter new city:");
154             this.City = scanner.next();
155             System.out.println("Enter new country:");
156             this.Country = scanner.next();
157
158             System.out.println("Profile updated successfully!");
159         } else {
160             System.out.println("Invalid username for profile update.");
161         }
```

Figure 39 User Profile Code Implementation

Table 9 User Login State Transition Table

State/Event	Register	Successful Registration	Invalid Details	Enter valid details	Successful Login
OpenApplication	Registration	OpenApplication	OpenApplication	OpenApplication	LoggedIn
Registration	Registration	LoggedIn	DisplayInvalid Registration	Registration	Registration
DisplayInvalidRegistration	DisplayInvalidRegistration	DisplayInvalidRegistration	DisplayInvalidRegistration	Registration	DisplayInvalidRegistration

Table 10 User Profile State Transition Table

State/Event	GetHomePage	Click ViewServices	Click AddServices	GetProfile	EditProfile	SaveChanges	ChangesSaved	Logout
LoggedIn	ViewHomePage	LoggedIn	LoggedIn	ViewProfileDetails	EditProfileDetails	LoggedIn	LoggedIn	LoggedOut
ViewHomePage	ViewHomePage	ViewPlan/Services	ViewHomePage	ViewHomePage	ViewHomePage	ViewHomePage	ViewHome Page	ViewHome Page
ViewPlan/Services	ViewPlan/Services	ViewPlan/Services	AddPlan/Services	ViewPlan/Services	ViewPlan/Services	ViewPlan/Services	ViewPlan/Services	ViewPlan/Services
AddPlan/Services	AddPlan/Services	AddPlan/Services	AddPlan/Services	AddPlan/Services	AddPlan/Services	SaveProfileChanges	AddPlan/Services	AddPlan/Services
ViewProfileDetails	ViewProfileDetails	ViewProfileDetails	ViewProfileDetails	ViewProfileDetails	EditProfileDetails	ViewProfileDetails	ViewProfile Details	LoggedOut
EditProfileDetails	EditProfileDetails	EditProfileDetails	EditProfileDetails	EditProfileDetails	EditProfileDetails	SaveProfileChanges	EditProfileDetails	EditProfileDetails
SaveProfileChanges	SaveProfileChanges	SaveProfileChanges	SaveProfileChanges	SaveProfileChanges	SaveProfileChanges	SaveProfileChanges	ViewProfile Details	LoggedOut

Promotions State Diagram:

The "PromotionsState" diagram represents the state transitions in a system related to managing promotions. Here's a description of the states and transitions in the diagram:

Idle State: This is the initial state where the system is waiting for an action to be performed.

PromoCodeEntered State: When a user adds a promo code, the system transitions to this state.

PromoDescEntered State: After entering the promo code, the user must input a promo description, leading to this state.

PromoAdded State: Upon successfully adding the promo with a valid code and description, the system transitions to this state.

Invalid Promo Code State: If the entered promo code is invalid, the system transitions to this state from the Idle state.

Invalid Promo Description State: Similarly, if the promo description entered is invalid, the system transitions to this state from the Idle state.

This state diagram helps in visualizing the flow of actions and validations related to adding and managing promotions in the system.

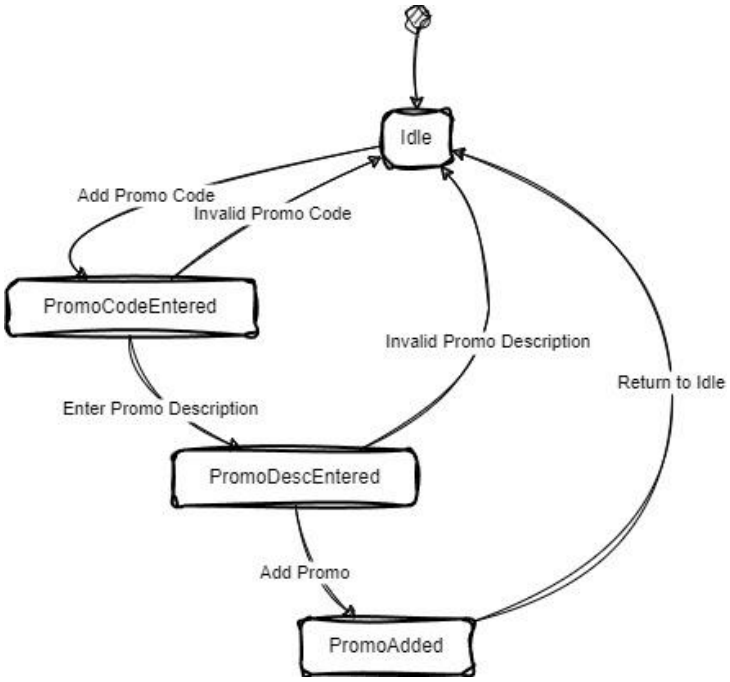

Figure 40 Promotions State Diagram

Table 11 Promotions State Transition Table

Current State	Add Promo Code	Enter Promo Description	Add Promo	Return to Idle	Invalid Promo Code	Invalid Promo Description
Idle	PromoCodeEntered	Idle	Idle	Idle	Idle	Idle
PromoCodeEntered	PromoCodeEntered	PromoDescEntered	PromoCodeEntered	PromoCodeEntered	Idle	PromoCodeEntered
PromoDescEntered	PromoDescEntered	PromoDescEntered	PromoAdded	PromoDescEntered	PromoDescEntered	Idle
PromoAdded	PromoAdded	PromoAdded	PromoAdded	Idle	PromoAdded	PromoAdded
Invalid Promo Code	Invalid Promo Code	Invalid Promo Code	Invalid Promo Code	Invalid Promo Code	Invalid Promo Code	Idle
Invalid Promo Desc	Invalid Promo Desc	Invalid Promo Desc	Invalid Promo Desc	Invalid Promo Desc	Idle	Invalid Promo Desc

```java
public class Promotions {

    private Map<String, String> promoCodes;  // Map to store promo codes and their descriptions
    public Promotions() {
        this.promoCodes = new HashMap<>();
    }
    public String PromoCode;
    public String PromoDesc;
    public void UsePromo(String promoCode) {
        if (promoCodes.containsKey(promoCode)) {
            System.out.println("Promo code applied successfully!");
            System.out.println("Promo Description: " + promoCodes.get(promoCode));
        } else {
            System.out.println("Invalid promo code. Please enter a valid one.");
        }
    }
    public void AddPromo(String promoCode, String promoDesc) {
        if (!promoCodes.containsKey(promoCode)) {
            promoCodes.put(promoCode, promoDesc);
            System.out.println("Promo code added successfully!");
        } else {
            System.out.println("Promo code already exists. Cannot add duplicate promo codes.");
        }
    }
    public void RemovePromo(String promoCode) {
        if (promoCodes.containsKey(promoCode)) {
            promoCodes.remove(promoCode);
            System.out.println("Promo code removed successfully!");
        } else {
            System.out.println("Promo code not found. Cannot remove non-existing promo code.");
        }
    }
}

```

Figure 41 Promotions Code Implementation

Sequence Diagram

Sequence diagram are used to visualize the interactions that takes place between actors and several other modules or classes in a system. The below given sequence diagram depicts several scenarios between the actors and modules within the Echo Connect System.

I. Login

1. **Actors and System involved –**
   Actor: The actor involved is the Customer.
   System: The systems involved are the UI (user interface), Google Oauth Server, and echoConnectApp.
2. **Description**- The sequence diagram depicts the following flow:
   - The Customer initiates the interaction by selecting the "Register with Google Account" option on the UI.
   - The UI then sends a "Register with Google" message to the echoConnectApp.
   - The echoConnectApp transmits a "Request" message to the Google Oauth Server.
   - Google Oauth Server:
   - If the user already has an account, it sends a message back to the echoConnectApp indicating the user already exists.
   - If the user doesn't have an account, it prompts the user to fill out a registration form to create a new profile and credentials.
   - echoConnectApp:
   - In the scenario where the user already has a Google account, it sends a message back to the UI to sign in with the existing credentials.
   - If the user creates a new account, the echoConnectApp receives a "User registered successfully" message from Google Oauth Server.
   - The echoConnectApp relays the "User registered successfully" message to the UI.
   - The UI retrieves the registration form from the echoConnectApp.
   - The Customer fills out the registration form on the UI.
   - The UI sends a "Request for registration" message to the echoConnectApp.
   - The echoConnectApp sends a "Request to register" message to the Google Oauth Server.
   - The Google Oauth Server registers the user and sends a "User registered successfully" message back to the echoConnectApp.
   - The echoConnectApp transmits the "User registered successfully" message to the UI.

II. View or create a plan:

1. **Actors and System involved –**
   Actor: The actors involved are Customers and Employees.
   System: The systems involved are the UI (user interface), echoConnectApp, plan, customer, ticket, and promotion.
2. **Description**- The sequence diagram depicts the following flow:
   - The diagram starts with the customer who can either view all plans or their plans.

- If the customer selects to view all plans, the UI sends a request to echoConnectApp, which checks if there are any available promotions which then retrieves all plans and displays them to the customer.
- If the customer selects to view their plans, the UI sends a request to echoConnectApp to get the customer data. echoConnectApp then retrieves the plans associated with the customer and displays them.
- An employee can also add a new plan. This involves creating a new plan, filling out a new plan form, and submitting the request. Upon submission, the echoConnectApp retrieves the plan creation details and adds a new plan. The UI then confirms that a new plan has been successfully added.

III. Upgrade service:

1. **Actors and System involved –**
   Actor: The actors involved are Customers and Employees.
   System: The systems involved are the UI (user interface), echoConnectApp, plan, billing, ticket.
2. **Description** - The sequence diagram depicts the following flow:
   - The customer initiates the process by selecting to upgrade their service plan through the UI.
   - The UI sends a request to echoConnectApp indicating that the customer wants to upgrade their service plan.
   - echoConnectApp communicates with the Service/Plan system to retrieve information on the customer's current plan.
   - echoConnectApp will raise a ticket to the employee for service/plan upgradation.
   - echoConnectApp then retrieves information on the upgraded plan the customer has selected.
   - Once echoConnectApp has both the current and upgraded plan information, it sends a request to the Service/Plan system to update the customer's plan.
   - The Service/Plan system removes the customer's current plan.
   - The Service/Plan system adds the upgraded plan to the customer's account.
   - The Service/Plan system communicates with the Billing system to update the customer's bill with the new service plan costs.
   - Once the update is complete, echoConnectApp sends a confirmation message back to the UI, indicating that the customer's plan has been successfully upgraded.

IV. Billing and Payments:

1. **Actors and System involved –**
   Actor: The actors involved are Customers and Employees.
   System: The systems involved are the UI (user interface), echoConnectApp, billing, and payment.
2. **Description-**
   - The process begins with the User/Customer who initiates the interaction by selecting “View My Billing” on the UI.
   - The UI then sends a request to the echoConnectApp to request the customer’s bill information.

- Upon receiving the request, the echoConnectApp retrieves the bill data, including previous bills, and sends it back to the UI.
- The UI displays the retrieved bill information to the User/Customer.
- Once the User/Customer has reviewed their bill, they can initiate a payment by selecting the “Payment” option on the UI.
- The UI then sends a payment request to the echoConnectApp.
- After receiving the payment request, the echoConnectApp retrieves the amount to be paid from the bill data.
- The echoConnectApp then sends a payment request to the bank (Payment) for processing.
- Once the payment is processed by the bank, the bank sends a payment completion acknowledgement to the echoConnectApp.

Upon receiving the acknowledgement, the echoConnectApp informs the UI that the payment is complete.

Due to the size of our sequence diagram, we have placed it on the next page for better clarity and ease of reference. Please Zoom to inspect the diagram clearly.

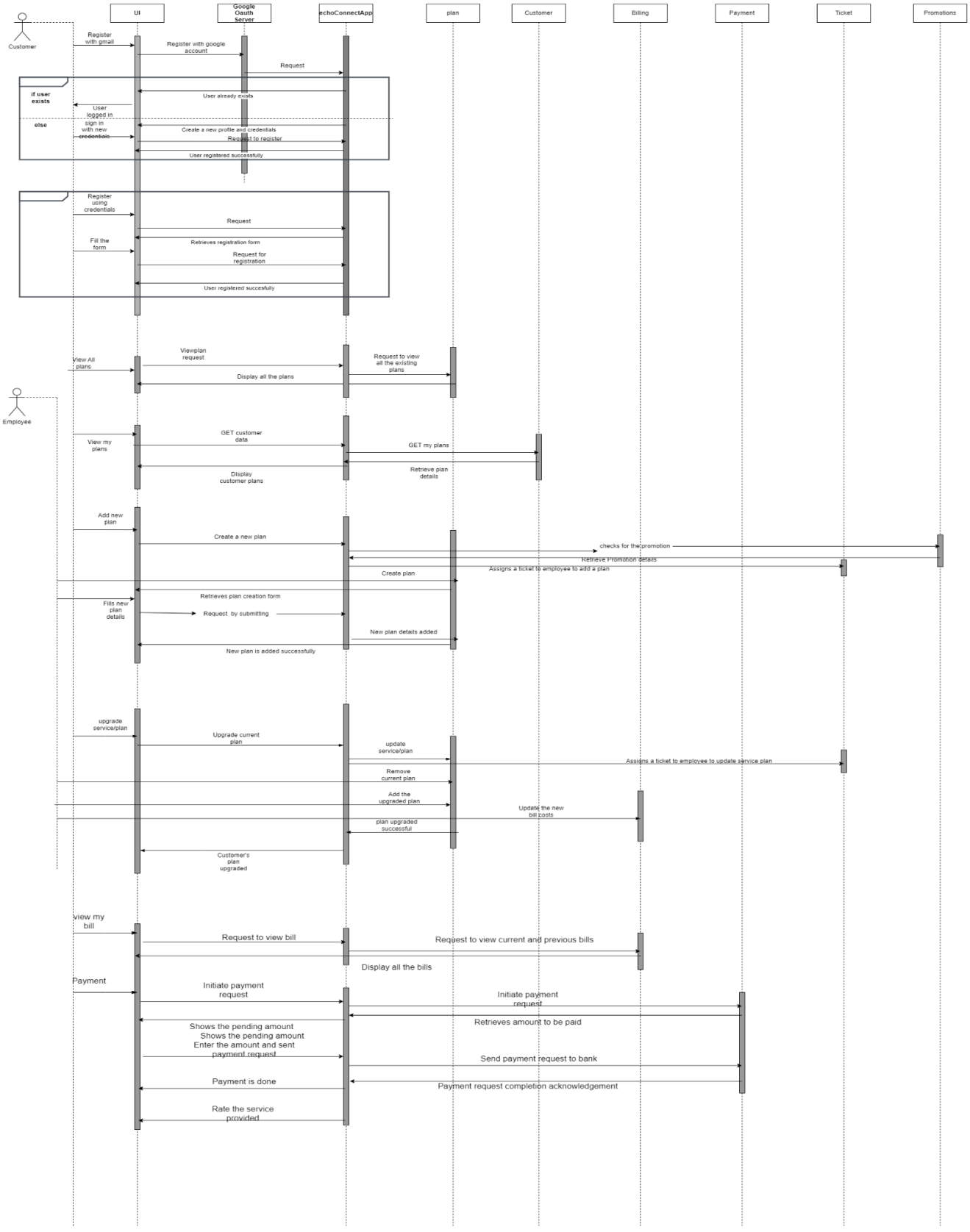

Figure 42 Sequence Diagram

Communication Diagram

A communication diagram, previously known as a collaboration diagram in older UML versions, is an interaction diagram illustrating interactions between objects or roles in a system or software application. It emphasizes the structure of messages exchanged between objects, contrasting with sequence diagrams which focus on message ordering over time. These diagrams are valuable for exploring how objects collaborate, identifying participants, required interfaces, structural changes, and data exchanged during interactions.

The below two communication diagrams illustrates the communication that takes place between the Customer and Employee roles along with several other Functionalities of the Echo Connect System.

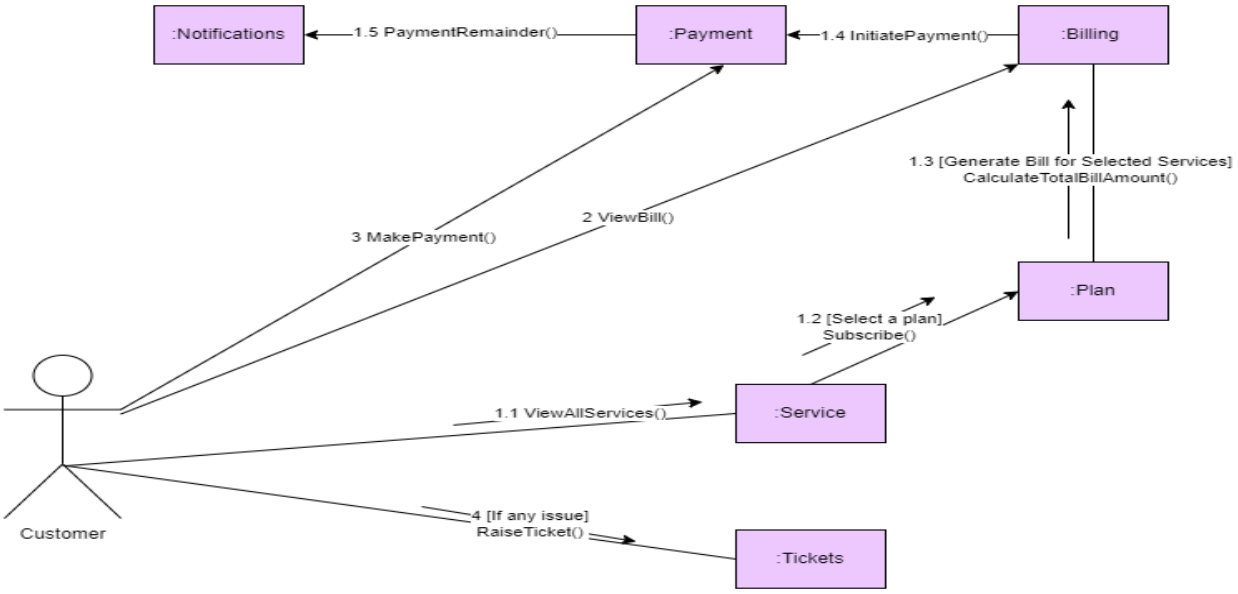

Figure 43 Communication Diagram - Customer

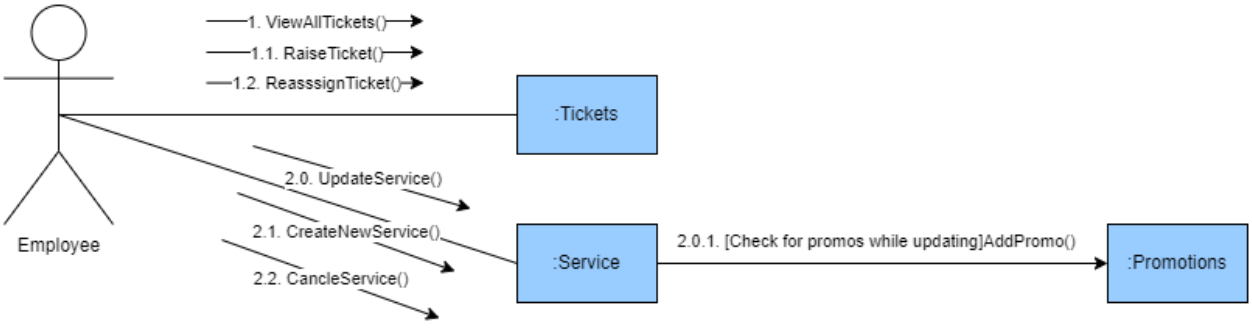

Figure 44 Communication Diagram - Employee

Conclusion

Our journey through the Echo Connect project was focused on designing and implementing a robust telecom service system. To ensure a thorough understanding of the system requirements, we prioritized the use of a series of models ranging from domain analysis to behavioral diagrams. Milestone 1 was foundational, focusing on stakeholder identification, defining system characteristics, and creating detailed use case diagrams. Milestone 2 built on this by translating these models into Java code, implementing UML class and object diagrams, and performing meticulous testing with JUnit. As we progressed to Milestone 3, our focus shifted to implementing complex OCL constraints using Papyrus software to ensure our models remained accurate and validated. Milestone 4 focused on behavioral diagrams, such as state, sequence, and communication diagrams. These diagrams provide a visual representation of the system's behavior and interactions.

Outcomes

Our goals were clear, building Echo Connect extensively, validating models through rigorous testing, translating designs into functional code, and visualizing system behavior effectively. Regular feedback gathering and iterative refinement were integral to the delivery of a system that was responsive to stakeholder needs. Feedback from our TA was instrumental in the refinement of our work and the streamlining of the development process.

References

[1] Kumar, Prasanna. “Understanding Telecom BSS: An End-to-End System Design with Cloud-Native Architecture Approach.” Medium, Medium, 29 Mar. 2023, medium.com/@2pkk/understanding-telecom-bss-an-end-to-end-system-design-with-cloud-native-architecture-approach-d8e69b177abc

[2] “Placeholder.” Telecom BSS Models, mef.dev/bss/. Accessed 20 Feb. 2024

[3] UML Class Diagram Relationships Explained with Examples | Creately. (2022, November 25). https://creately.com/guides/class-diagram-relationships/

[4] Papyrus. (n.d.). https://eclipse.dev/papyrus/

[5] Vpadmin. (2023, September 13). Class diagrams vs Object diagrams in UML - Visual Paradigm Guides. Visual Paradigm Guides. https://guides.visual-paradigm.com/class-diagrams-vs-object-diagrams-in-uml/

[6] Cabot, J. (2022, May 14). The Ultimate Object Constraint Language (OCL) tutorial. Modeling Languages. https://modeling-languages.com/ocl-tutorial/

[7] Help - Eclipse Platform. (n.d.). https://help.eclipse.org/latest/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FOCLExamplesforUML.html

[8] School of Informatics Courses: SDM Lab 6. (n.d.). https://www.inf.ed.ac.uk/teaching/courses/sdm/labs/ocl.html

[9] GfG. (2024, January 15). State Machine Diagrams Unified Modeling Language (UML). GeeksforGeeks. https://www.geeksforgeeks.org/unified-modeling-language-uml-state-diagrams/

[10] IBM Developer. (n.d.). https://developer.ibm.com/articles/the-sequence-diagram/

[11] Diagram maker for developers. (n.d.). https://www.gleek.io/blog/uml-communication-diagram