

# *CREDIT CARD MANAGEMENT SYSTEM*

CS5721 Software Design

*URL : [https://github.com/riyajoe/CS5721\\_Software\\_Design.git](https://github.com/riyajoe/CS5721_Software_Design.git)  
commit Id:69d7dc7*

*Ciara Tully 20213743, Dhavan Shah 20031696, Praveen Talavar  
20089236, Riya Joe 20023693, Yukti Patil 20097786*

*Group 17*

## Table of Contents

<b>1</b>	<b>BUSINESS SCENARIO.....</b>	<b>3</b>
<b>2</b>	<b>SOFTWARE LIFECYCLE MODEL.....</b>	<b>4</b>
2.1	CONSIDERED APPROACHES .....	4
2.2	ADOPTED APPROACH.....	4
<b>3</b>	<b>PROJECT PLAN.....</b>	<b>5</b>
3.1	ALLOCATION OF ROLES.....	5
3.2	LIGHT WEIGHT PROJECT PLAN.....	5
<b>4</b>	<b>REQUIREMENTS .....</b>	<b>7</b>
4.1	FUNCTIONAL REQUIREMENTS .....	7
4.2	USE CASE DIAGRAMS.....	8
4.2.1	<i>System Use Case Diagrams .....</i>	<i>8</i>
4.3	KEY USE CASES DESCRIPTIONS.....	12
4.3.1	<i>View Customer Transactions Use Case Description .....</i>	<i>12</i>
4.3.2	<i>Apply for Card Use Case Description.....</i>	<i>13</i>
4.3.3	<i>Transaction Authorization Use Case Description.....</i>	<i>14</i>
4.4	NON-FUNCTIONAL REQUIREMENTS .....	14
4.5	QUALITY SUPPORTING METHODS.....	16
4.6	GUI PROTOTYPES .....	17
<b>5</b>	<b>SYSTEM ARCHITECTURE.....</b>	<b>19</b>
5.1	PACKAGE DIAGRAM.....	19
5.2	TECHNOLOGIES.....	20
5.2.1	<i>Flask.....</i>	<i>20</i>
5.2.2	<i>MySQL .....</i>	<i>20</i>
5.2.3	<i>PyMySQL.....</i>	<i>20</i>
5.3	SYSTEM ARCHITECTURE.....	20
5.4	RUN TIME PROCESSING.....	21
<b>6</b>	<b>SYSTEM ANALYSIS .....</b>	<b>23</b>
6.1	CANDIDATE CLASS IDENTIFICATION .....	23
6.2	ANALYSIS CLASS DIAGRAM .....	23
6.3	COMMUNICATION DIAGRAM FOR KEY USE CASES .....	27
6.3.1	<i>System User Log In .....</i>	<i>27</i>
6.3.2	<i>Approve Card.....</i>	<i>27</i>
6.3.3	<i>Customer Application.....</i>	<i>28</i>
6.3.4	<i>View Statement.....</i>	<i>28</i>
6.4	KEY USE CASE SEQUENCE DIAGRAMS .....	29
6.4.1	<i>Approve Card.....</i>	<i>29</i>
6.4.2	<i>Transaction Authorisation Request.....</i>	<i>29</i>
6.4.3	<i>Get Statement .....</i>	<i>30</i>
6.5	ENTITY RELATIONSHIP MODEL (E-R MODEL) .....	31
<b>7</b>	<b>OBJECT ORIENTED DESIGN (CODE) .....</b>	<b>33</b>
7.1	DESIGN PATTERNS.....	33
7.2	FAÇADE PATTERN.....	33
7.3	BUILDER PATTERN.....	35
7.4	FACTORY PATTERN .....	37
7.5	DECORATOR PATTERN.....	39
7.6	INVERSION OF CONTROL / DEPENDENCY INJECTION .....	42
7.7	MVC.....	43
7.8	CASE TOOLS.....	44
7.8.1	<i>Draw.io.....</i>	<i>44</i>
7.8.2	<i>Version Control.....</i>	<i>44</i>
7.9	TESTING .....	46
7.9.1	<i>Test Cases.....</i>	<i>47</i>
<b>8</b>	<b>CODE ADDED VALUE.....</b>	<b>48</b>

8.1.1	USE OF OBJECT RELATIONAL MAPPING.....	48
<b>9</b>	<b>RECOVERED BLUEPRINTS.....</b>	<b>50</b>
9.1	STATE CHART .....	50
9.2	DESIGN-TIME CLASS DIAGRAM.....	52
<b>10</b>	<b>DEPLOYMENT AND COMPONENT ARCHITECTURE .....</b>	<b>53</b>
<b>11</b>	<b>CRITIQUE .....</b>	<b>55</b>
11.1	STATE CHART .....	55
11.2	DESIGN PATTERNS.....	55
11.3	ARCHITECTURE .....	55
11.4	PYTHON AND DESIGN PATTERNS.....	55
11.5	PROJECT CRITIQUE .....	55
<b>12</b>	<b>BIBLIOGRAPHY .....</b>	<b>56</b>
<b>13</b>	<b>APPENDIX .....</b>	<b>57</b>
13.1	TABLE OF FIGURES.....	57
13.2	TABLE OF TABLES .....	58
13.3	SOURCE CODE INFORMATION.....	58

## 1 Business Scenario

In 2019 the global economy reached a value of \$87.8 trillion (USD), a growth of 2.3% or \$2.3 trillion since 2018. This year however, as a result of the pandemic, the global economy is expected to contract for the first time since WWII. [1]

We, Group 17 Software Technologies, have been approached by a group of investors who have identified a business opportunity through the importance of availability of credit to the individual consumer and small medium enterprises (SMEs) in times of recession.

The benefits of credit through credit cards to local, national and global economies are interconnected and numerous: [2]

- By decreasing the amount of cash in circulation, economic activity is registered and tax revenue is collected. This tax revenue boots the economy providing business opportunity and in turn decreased unemployment.
- The use of a credit card allows an increase of available capital to persons and small business aiding their growth and viability and facilitating trade on a local, national and international level.
- The regulation of credit cards in the EU mandates buyer protection guarantees which provides safety and encourages spending.

The credit card thus should be recognised as an important economic device to keep cashflow running through the economy and as a vital tool in global recovery.

Anticipating a globally synchronous government promotion of consumer spending and a resulting demand for small amounts of credit and credit cards in the coming decade, our investors have identified a gap in the market for a modern out-of-the-box credit card management system solution which can be integrated to work in parallel with existing systems and allows for customer interaction.

The proposed solution will be marketed to European institutions who may not yet offer a consumer credit card <sup>1</sup> or to institutions who have a legacy system which could be replaced.

The contents of this document are intended to capture the design and implementation of the system proposed by our investors.

---

<sup>1</sup> Credit unions, airlines, postal service companies

## 2 Software Lifecycle Model

A high-level goal of this project is to produce quality software according to client specification within a short timeframe. In order to achieve this, careful consideration of software lifecycle model is required. We decided on a model to adopt by assessment under two main categories: suitability to the team and facilitation of delivery of a high-quality product according to client specifications.

### 2.1 Considered Approaches

*Iterative Waterfall Model:* This model follows closely to the 'traditional' waterfall model however the difference is that a feedback loop from each development phase to the previous phase is implemented. While a high level of feedback is desirable to allow the team to deliver a product according to client specifications, it is likely that implementing this model would result in a lot of time being spent in the initial phases of requirements analysis and a slow delivery overall.

*Extreme Programming:* Another model which was considered was extreme programming approach. Characteristics of this approach include incremental planning, continuous testing, small releases and simple design. Incremental planning and simple design are features which suit the specification and delivery goal of this project. As well as that, as our software development team are a team of multi-talented individuals and this approach was favourable to them. However, this approach would not subscribe to the production of much documentation. As a team who are answerable to investors who in turn intend to market this product as an out-of-box solution to other companies, documentation is necessary.

*Rational Unified Process (RUP):* This software lifecycle model involves iterating over four project phases; inception, elaboration, construction and implementation. While the documentation produced by this approach is attractive, the time restraints on project delivery suggest that there would not be time to complete the full four cycles giving adequate attention to each iteration. If this approach was attempted, it is likely that it would either be reduced to a waterfall model or result in a delay.

### 2.2 Adopted Approach

After careful consideration, we decided that a combination of the three above approaches would enable delivery of an on time and quality software product. From RUP, the project phases have been adopted: inception, elaboration, construction and implementation. However, we will consider this as two phases: *inception and elaboration*, and *construction and implementation*. The process will then be as follows: Beginning with the *inception and elaboration* phase, we will complete the requirements and analysis. Then moving into the *construction and implementation* design and implementation is carried out. However during the design and implementation, it will be necessary to return to the previous phase for short amounts of time to adjust some requirements or analysis. This cycling back to previous the previous phase adopted from the iterative waterfall model. Taking example from extreme programming, daily meetings, pair programming, continuous integration and testing will be implemented throughout the *construction and implementation phases*.

### 3 Project Plan

#### 3.1 Allocation of Roles

	<b>Role</b>	<b>Description</b>	<b>Designated Team Member</b>
1	Project Manager	Sets up group meetings, gets agreement on the project plan, and tracks progress.	Yukti
2	Documentation Manager	Responsible for sourcing relevant supporting documentation from each team member and composing it in the report.	Ciara
3	Business Analyst / Requirements Engineer	Responsible for section 6 - Requirements.	Ciara/Riya
4	Architect	Defines system architecture	Yukti
5	Systems Analysts	Creates conceptual class model	Riya
6	Designer	Responsible for recovering design time blueprints from implementation.	Praveen
7	Technical Lead	Leads the implementation effort	Dhavan/Riya
8	Programmers	Each team member to develop at least 1 package in the architecture	ALL
9	Tester	Coding of automated test cases	Dhavan
10	Dev Ops	Must ensure that each team member is competent with development infrastructure, e.g. GitHub, Bamboo, etc.;	Praveen

#### 3.2 Light Weight Project Plan

<b>Week</b>	<b>Section Title</b>	<b>Deliverable</b>
3	Set up GitHub & Project Research	<ul style="list-style-type: none"><li>• Read previous projects</li><li>• Come up with system ideas</li></ul>
4	Requirements:	<ul style="list-style-type: none"><li>• Functional Requirements</li><li>• Non Functional Requirements</li><li>• Use Case Diagram</li><li>• Key Use case descriptions</li></ul>
5	Analysis	<ul style="list-style-type: none"><li>• Analysis Class Diagram</li><li>• Sequence Diagram</li><li>• Communication Diagram</li></ul>
6	High level architecture	<ul style="list-style-type: none"><li>• Discuss potential application framework and technology.</li><li>• High level package diagram</li></ul>

7	Coding Iteration 1: basic infrastructure and 2 key use cases	<ul style="list-style-type: none"> <li>• Set up basic repository file structure in line with package diagram</li> <li>• Log in use case</li> <li>• Customer Application use case</li> <li>• Dashboard factory</li> </ul>
8	Coding Iteration 2: 2 more use cases	<ul style="list-style-type: none"> <li>• Batch Processing</li> <li>• Block card and change pin functionalities</li> </ul>
9	Coding Iteration 3 Another use case and MVC (GUI)	<ul style="list-style-type: none"> <li>• Employee approve card application.</li> <li>• Admin get block request and block card.</li> <li>• Implement database controller</li> </ul>
10	Coding Iteration 4: Another use case and Added Value	<ul style="list-style-type: none"> <li>• Interest addition and customer view statement.</li> <li>• Testing</li> </ul>
11	Over run	
12	Architecture and Design Recovery	<ul style="list-style-type: none"> <li>• Design Time Class Diagram</li> <li>• Recovered Architecture Diagram</li> <li>• Recovered State charts</li> <li>• MVC diagram</li> </ul>

## 4 Requirements

### 4.1 Functional Requirements

The functional requirements of a system are concerned with the intended use of the system. It is for this reason that we discuss functional requirements through the user paradigm. By capturing the goal of the user when interacting with the system, we can ensure that the desired functionality is implemented. Any person or system who interacts with the system can be framed as a system user thus it is assured that all desired system functionality can be documented through this frame of vision.

Users who wish to interact with the system will have a user account. Depending on the user account type, the user will be able to interact differently with the system. The information associated with each user is user id, username, password and account number. Each user has an associated role and permission. The role is captured by a role id and role name, the permission is represented by a permission id number and a name.

#### *Admin:*

- When an admin logs into their account the admin dashboard is shown.
- From the admin dashboard, the admin can view and manage users of the system.
- An admin can block a card by entering the card number.
- The admin can schedule batch processing of account transactions which are received by the system.

#### *Customer (Cardholder) :*

- When a cardholder logs on to the system should view a customer dashboard.
- Cardholder should be able to view a list of transaction statements.
- Customers should be able to request that their card is blocked
- Customers should be able to change their pin via a user interface.
- Customers can choose to renew their card via the system

#### *Employee :*

- When an employee logs into their account they see the employee dashboard.
- The employee can view the list of pending customer applications.
- The employee can check the eligibility of the customer applications and onboard eligible customers.
- The employee can also manually input customer card application forms for the case where physical applications are submitted and need to be recorded in the system.
- The employee can view the transactions related to a certain account.
- The employee can view the debts, interest and the account details associated with an account.

#### *Applicant*

- The applicant is a person who opens the system web page but does not yet have a card or an account. From the log in page the new customer can create an account which will allow them to apply for a card.
- When the customer clicks 'sign up' they are given the option to input their details and apply for an account.



### Machine (Card Network)

- Card terminal machines should be able to send validation requests to the system.
- The card terminal should then receive a request authorised/denied message from the system.

## 4.2 Use Case Diagrams

Upon assessment of the system users, the way they interact with the system and the system response, the following use case diagrams have been devised:

### 4.2.1 System Use Case Diagrams

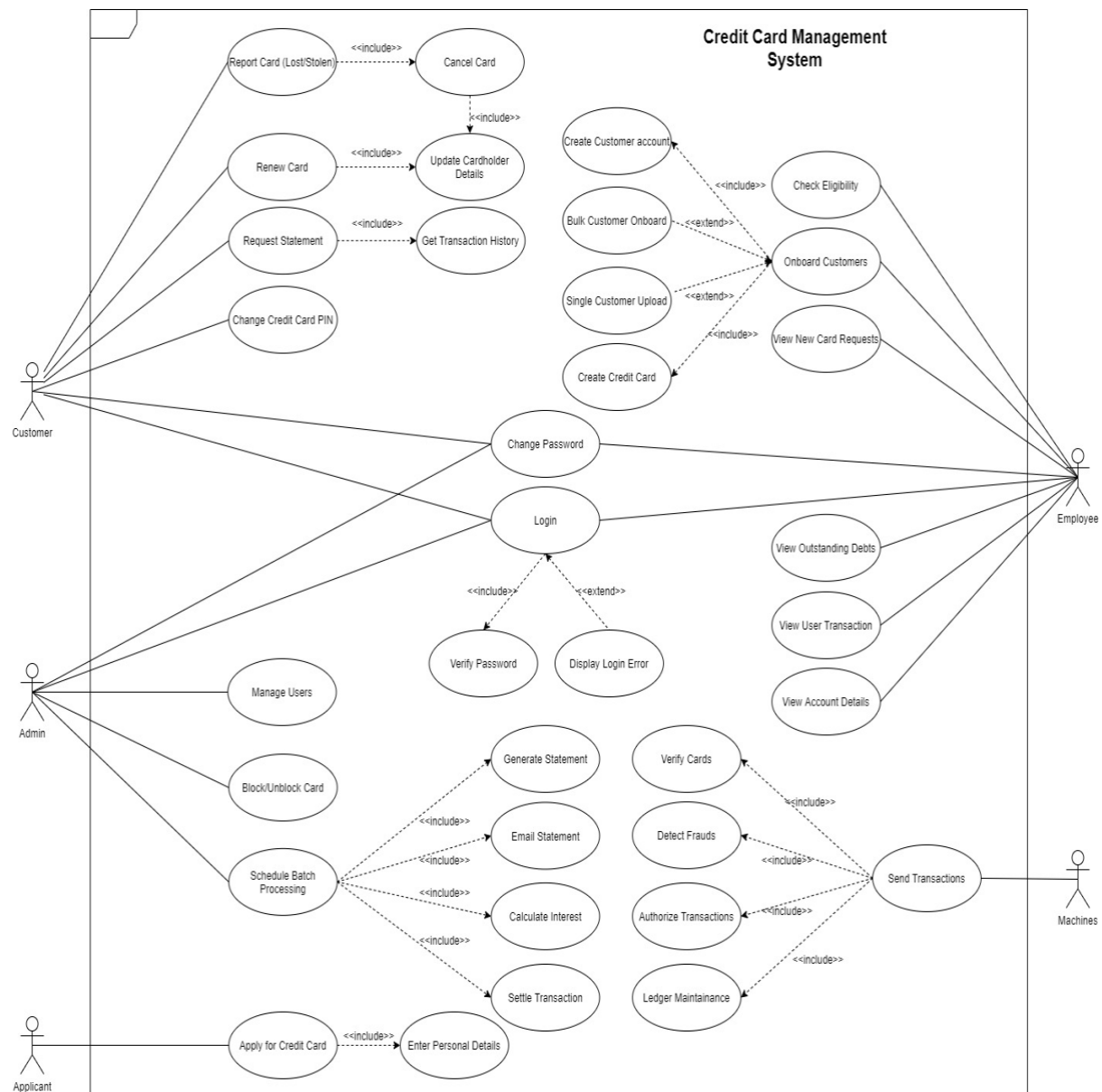


Figure 1 System Use Case Diagram

From the above system use case diagram, we can identify the actors Admin, Employee, Machines as independent actors. An applicant on onboarding changes to a customer in the system. A customer can have services to choose like report card, renew card, request statement, change pin. Functionalities of admin is to manage

users, Block and unblock card and run schedule batch processing. Employee will initiate onboarding the customer, checking eligibility ,viewing the transactions and outstanding debts .

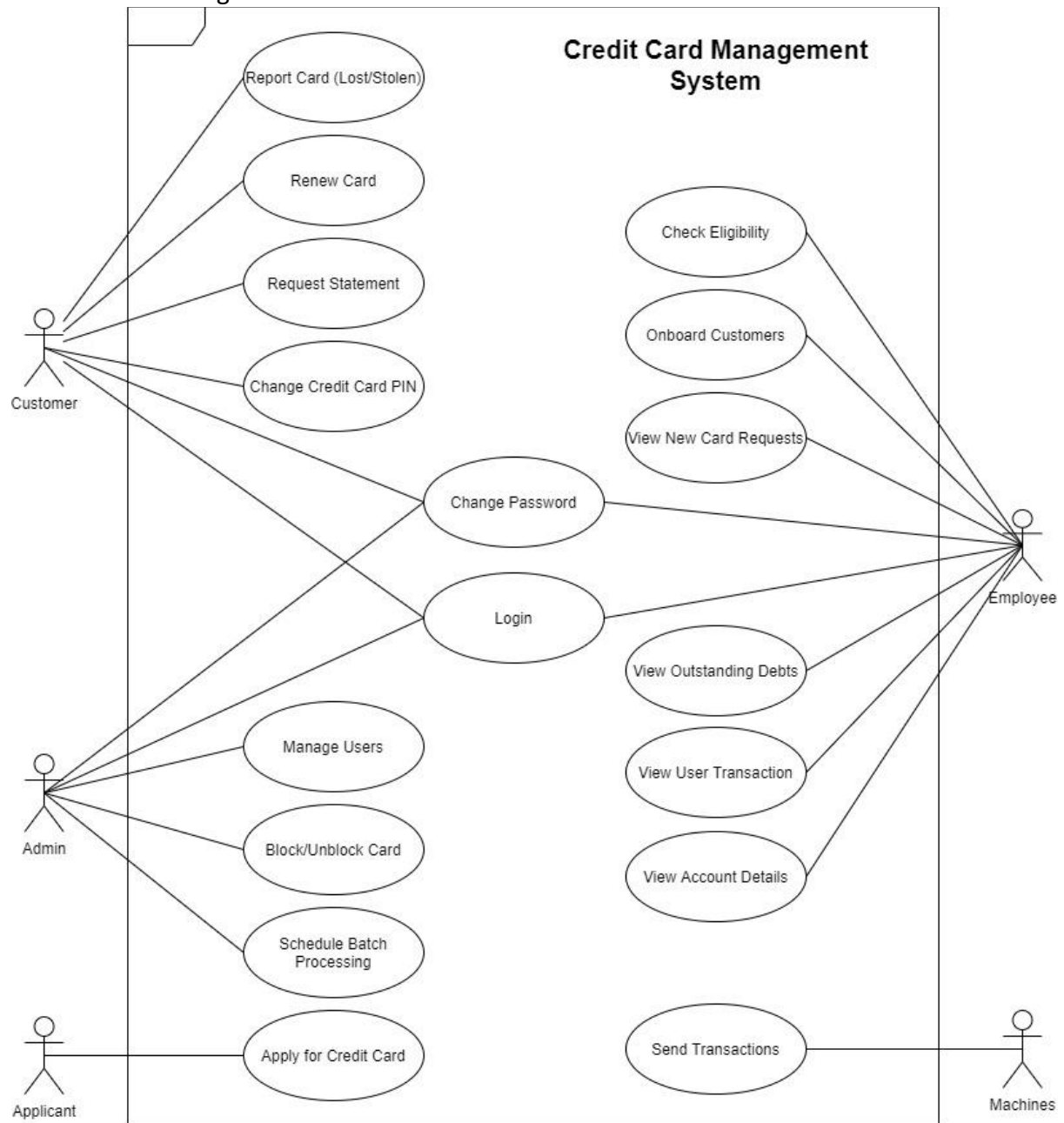


Figure 2 System Use Case Diagram without Extends and Includes

The applicant applies for a credit card and once the applicant is onboarded by the employee, he or she can avail the facilities of the credit card management system. Admin represents the IT support of the credit card system and an employee is an individual hired by the credit card company for managing and customer care service. The admin monitors the batch process which is run at a targeted interval. Customers on using credit card will have incoming transactions for card swipe machines, Automated teller Machines. These transactions are automatically authorized depending on the card details, timestamp, user details. Once it is Authorized it is posted in the database system that tracks the transaction via their transaction id. A customer Requesting for a Credit Card statement can avail the View Statement service which displays each transaction made and the effective money along with

interest to be paid at the end of a month. Customer can request for a block card if the card is stolen, tampered or misplaced. Admin will monitor the list of block cards and initiates the Blocking. The admin, customer and employee have access to the system and their services will be availed based on their role. The term user in general refers to these 3 actors in the card management system. This diagram illustrates the services dependent on the actors.

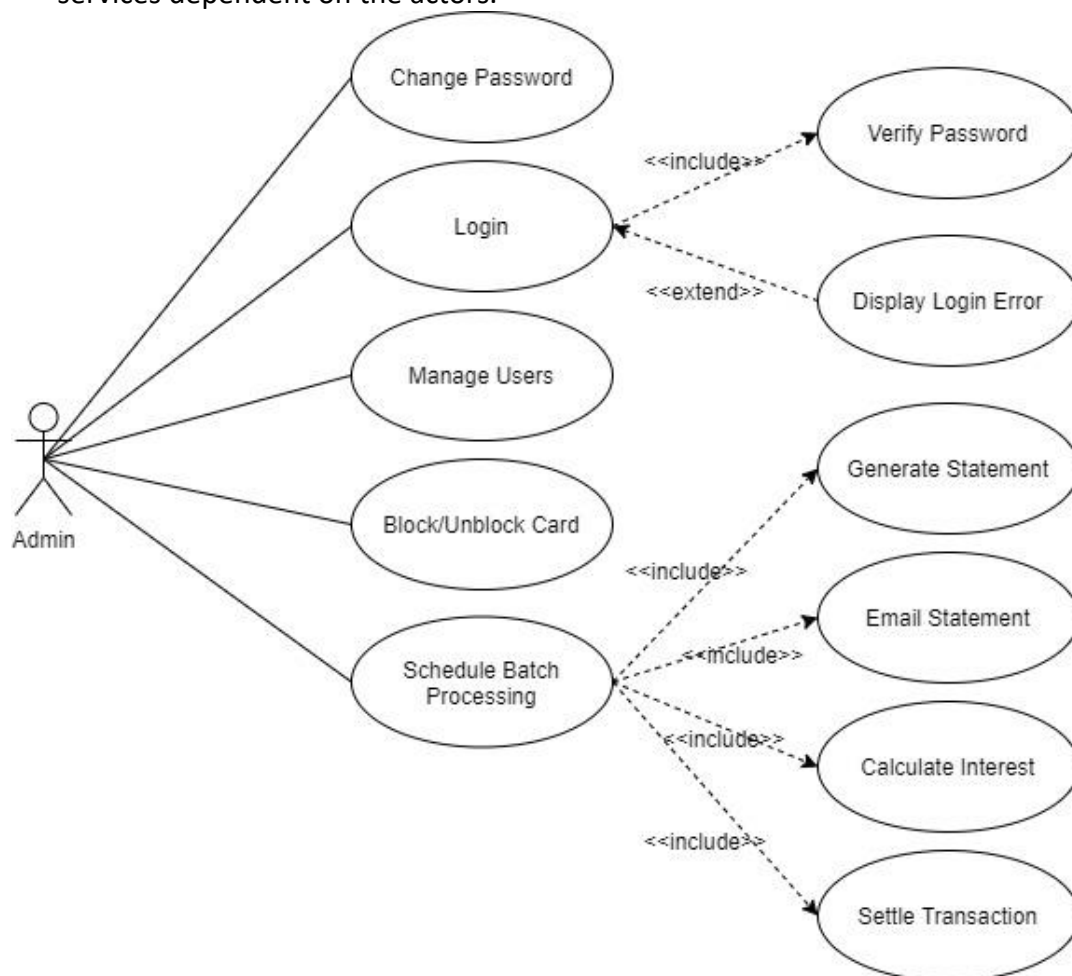


Figure 3 Admin Use Case Diagram

The Admin on invoking the schedule batch processing, the transactions will be bulked up for settlement along with its Interest calculation. Credit card system offers a multitude of cards ranging from basic to different varieties. Each card depending on its grade will have an interest rate incurred at the time of charging. Batch processing shell will auto send Statements to a customer to remind the customer about payment which is to be due and informs him about the transactions completed before batch processing.

The External system sends in transactions, it can be of debit or credit type. checking the authenticity of these transactions, gave rise to fraud detection rules. A basic fraud detection rule would be to check if the transaction card number exists with a valid user. Next would be to check if there are frequent number of transactions occurring at the same time in the system. Such repeated fashion of transaction requests needs to be verified before posting into the database. Authorization is the process of verifying the user and enabling them to transact money from their available balance. Once transactions are posted, the system moves it to a ledger system. The ledger is a bookkeeping system keeps tracks of all incoming transactions be it credit or debit. It also holds the summation details of the transactions. Since millions of transfers occur a day, it is beneficial and safer to keep a ledger to monitor. In case

of data loss ledger could come in handy. Ledger will help in cross verification of transactions while settlement and charging.

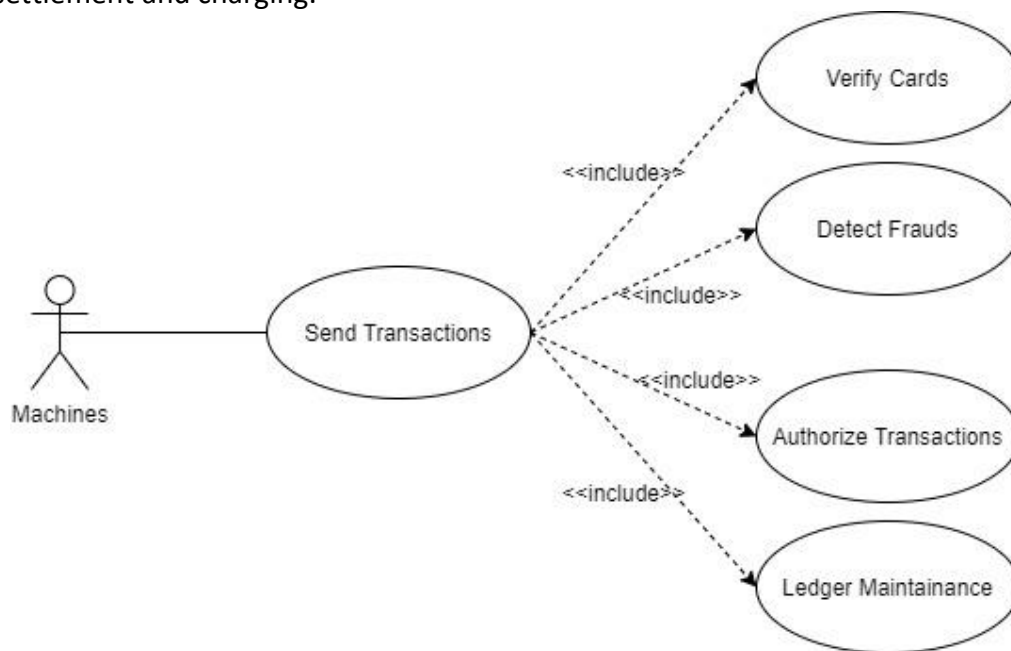


Figure 4 Machine (External System) Use Case Diagram



Figure 5 Employee Use Case Diagram

The employee has a major role in the system. He is responsible for the onboarding the customers. He can onboard customer individually or in bulk. The employee can choose to onboard, based on eligibility (monthly income, card type etc). Employee has the option

to view the outstanding debts and issue new card. In the credit card management system employee has separate login credentials just like admin.

### 4.3 Key Use Cases Descriptions

Some key use cases of the system are the customer “View Statement” use case, the “Report Card Lost/Stolen” use case and the “Apply for Credit Card Use Case”, detailed descriptions of these use cases are provided in these subsections.

#### 4.3.1 View Customer Transactions Use Case Description

Table 1 View Customer Transaction Use Case Description

<b>USE CASE</b>		View Customer Transactions
<b>Goal in Context</b>		To display a particular Customers transactions
<b>Scope &amp; Level</b>		Enterprise, Summary
<b>Preconditions</b>		Admin is logged in successfully, the Customer exists in system
<b>Success End Conditions</b>		Admin has the list of transactions for the Customer
<b>Failed End Condition</b>		The Customer is not found in the bank’s system
<b>Primary, Secondary, Actors</b>		Admin, Bank database
<b>Trigger</b>		Admin sends request to get hold of Customer’s transaction
<b>DESCRIPTION</b>	<b>Step</b>	<b>Action</b>
	1.	Admin requests to get the transaction history of a particular customer
	2.	System queries the database to find the customer and the transaction history
	3.	If customer found, return the transaction history
	4.	If customer not found, flash an error saying user does not exist
<b>EXTENSIONS</b>	<b>Step</b>	<b>Branching Action</b>
	3a	On analysing the transactions if an anomaly is found. Detect fraud possibility. If fraud confirmed, block card
<b>VARIATIONS</b>	<b>Step</b>	
	1	Admin checks the physical user statement for transactions.
<b>RELATED INFORMATION</b>		
<b>Priority</b>		Medium
<b>Performance</b>		5 seconds from the request generation
<b>Frequency</b>		20/day
<b>Channel to actors</b>		Not yet determined
<b>OPEN ISSUES</b>		If an anomaly is found in the user’s transaction. How to determine its a fraud?
<b>Due Date</b>		Release 1.0
<b>Any other management info</b>		N/A

<b>Superordinates</b>	Get User list
<b>Subordinates</b>	Fraud Detection

#### 4.3.2 Apply for Card Use Case Description

Table 2 Apply for Card Use Case Description

<b>USE CASE</b>		Enroll for Card
<b>Goal in Contest</b>		Submit a credit card approval
<b>Scope &amp; Level</b>		New User
<b>Preconditions</b>		User does not have access to the system. Can access the enrolment link.
<b>Success End Conditions</b>		Application is submitted
<b>Failed End Condition</b>		Application is not submitted
<b>Primary, Secondary, Actors</b>		Primary, Customer
<b>Trigger</b>		User wishes to obtain a credit card
<b>DESCRIPTION</b>	<b>Step</b>	<b>Action</b>
	1.	The user accesses the link to enroll for a card.
	2.	The user fills in an interactive form for Customer Application
	3.	The user selects the card type he/she wishes to apply and uploads necessary documents.
	4.	The system verifies that all the fields are filled correctly and that quality of the uploaded documentation is sufficient and in the right format.
	5.	The user clicks submit application button
	6.	The system notifies the user that their application has been received and that they will be contacted shortly notifying them of their application status.
<b>EXTENSIONS</b>	<b>Step</b>	<b>Branching Action</b>
	4a	The user does not fill in a subject field correctly. 4a1. Customer fills in the subject field again.
	4b	The uploaded documentation is not of good enough quality. 4b1. The customer re-uploads alternative supporting documentation.
	6a.	The user does not click submit application. 6a1. After 10 minutes the screen times out and the user is logged out
<b>VARIATIONS</b>	<b>Step</b>	
	3.	The user may choose to submit supporting documentation by post.
<b>Priority</b>		Top
<b>Performance</b>		10 minutes from log-in to submit application.
<b>Frequency</b>		50/day
<b>Channel to actors</b>		Not yet determined.
<b>OPEN ISSUES</b>		What if the user already has a card?
<b>Due Date</b>		Release 1.0
<b>Any other management info</b>		n/a
<b>Superordinates</b>		
<b>Subordinates</b>		Check Eligibility , Approve Card



### 4.3.3 Transaction Authorization Use Case Description

Table 3 Transaction Authorisation Use Case Description

<b>USE CASE</b>	Transaction Authorization	
<b>Goal in Contest</b>	Authorize the incoming transaction requests to the system	
<b>Scope &amp; Level</b>	System Automated	
<b>Preconditions</b>	Transactions should be from registered and valid cards only	
<b>Success End Conditions</b>	Transaction will be authorized	
<b>Failed End Condition</b>	Failed Transaction	
<b>Primary, Secondary, Actors</b>	External system – POS, Credit, Debit, ATM, Online shopping	
<b>Trigger</b>	Customer with a card swipes his card at a store	
<b>DESCRIPTION</b>	<b>Step</b>	
	1	Customer with a card swipes his card at a store
	2	Transaction request is generated
	3	Card management system gets the request, verifies the request and authorizes it
	4	A response is sent to the external system
<b>EXTENSIONS</b>	<b>Step</b>	
	2a	Customer credit card is no longer valid, expiry date incorrect pIN
<b>VARIATIONS</b>	<b>Step</b>	
	1a	A fraud transaction , Card credit deducted but transaction failed.
<b>RELATED INFORMATION</b>		
<b>Priority</b>	2	
<b>Performance</b>	Multiple authorization requests will be sent to the system	
<b>Frequency</b>	100 Requests per hour	
<b>Channel to actors</b>	Not yet Determined	
<b>OPEN ISSUES</b>	Not Identified	
<b>Due Date</b>	Version 1.0	
<b>Any other management info</b>		
<b>Superordinates</b>		
<b>Subordinates</b>		

### 4.4 Non-Functional Requirements

It is the non-functional requirements by which the quality of the produced product is measured by the user. At Group 17 technologies, we strive to always produce high quality software products and this project is no different. However, as this product has been commissioned with the intention of re-sale, extra care must be taken with respect to the non-functional requirements.

### *Usability*

- The system should allow for easy onboarding of new customers, both through online and physical methods.
- The addition and approval of cardholders should be a simple process and require few actions once the employee has logged on.
- Creating a user account should take all take place on one webpage and be a streamlined process.
- All the options that the customer can take should be marked clearly and visible on the customer dashboard.
- For buttons which are actionable but do not redirect to another page, the user should be given some notification that the action that they attempted has been executed/failed.
- All user interfaces should be clear, easy to navigate and responsive.
- Any messages conveyed to the user should be of a non-technical nature.

### *Scalability*

- The system should be scalable to be used with any number of employees and accessed concurrently by multiple users.
- It should be suitable for companies maintaining from a small number to a very large number of cards.

### *Integrability*

- The system should be easy to integrate with existing client software systems.
- The system should be able to interact with payment gateways and card associations to receive transaction authorisation requests.
- The system should also send approvals and display authorisation requests to external systems via existing card networks.

### *Maintainability*

- The structure of the system source code should make changing the code as easy as possible.

### *Security*

- Customer passwords should be stored as encrypted values and should not be accessible to staff.
- The data should be assessed before insertion to the databases to prevent SQL injection attacks.
- A combination of firewalls, anti-spam, load balancing and other techniques should be implemented in order to deter denial of service attacks.
- User data, card data and account data should be held in different databases to help prevent against identification and linkage attacks.

### *Data integrity*

- Databases should be stored on a different server to the system source code to protect data integrity.



### Legal (GDPR)

- Only information related to the processing of customer accounts and card should be requested and stored.
- Terms and conditions of use of the system should be presented to the customer for agreement upon signing up.
- The deletion of cardholder data upon request should be possible.

## 4.5 Quality Supporting Methods

Quality of the software product can be assessed under two main categories: the degree to which the product produced is suitable for its purpose, and the structural quality of the code. In the delivery of this system, the integrity of the source code and functionality of the system go hand in hand – if one of them is substandard than the goal of developing a quality software product fit for purpose has not been realised.

During the inception and elaboration phase and indeed throughout the whole project, a high level of importance is placed on client interaction and communication. A constant feedback loop throughout is effective in keeping the product in line with expectations of the client. As well as this, the client provides market knowledge to the development team. In order for the product be market competitive and fit for purpose, the team must be led by client expertise and expertise. Thus, two-way communication with the client or client representative on a regular basis (via email and video call) is our proposed method to support the delivery of a software system which is suitable for purpose.

Majority of the lifecycle of a software product is spent in maintenance. The structural quality of the code is the key contributor to the ease at which a system can be maintained. As the system will likely be maintained by those who did not develop it,<sup>2</sup> low coupling and high cohesion is highly desirable and important. Use of architecture design and system design patterns should be implemented in order to facilitate this, which in turn facilitates changes being made to the system with minimum risk and effort. The neglect of good architecture and design patterns will undoubtedly result in code with an unnecessarily high level of complexity and low level of reliability.

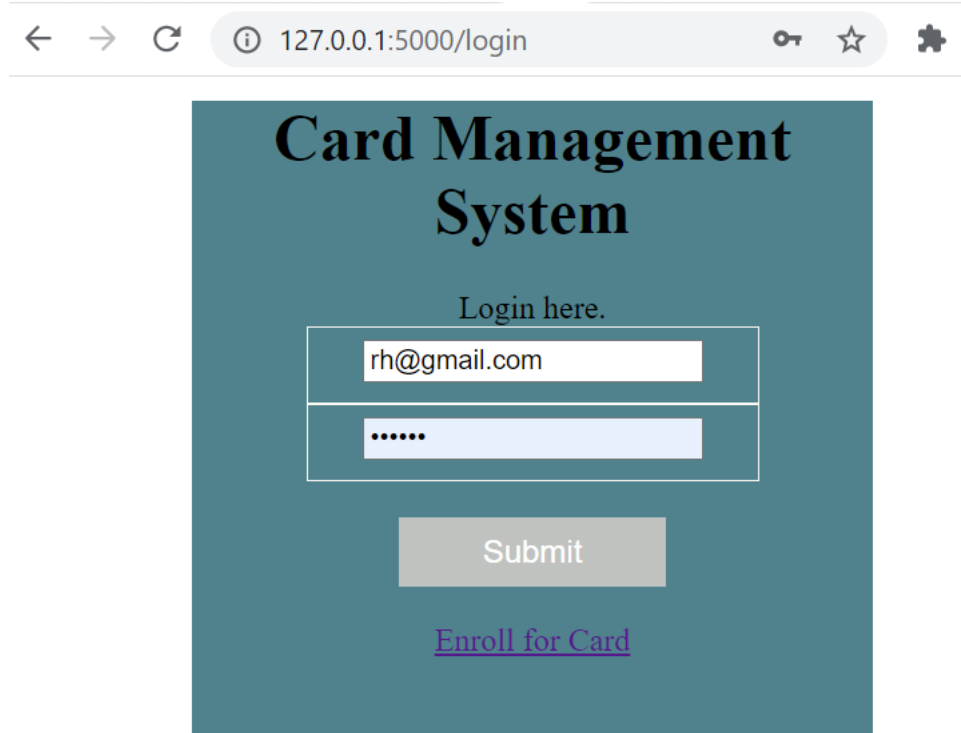
Due to the nature of the interaction of the user of the system, we expect to at least to use the “Model, View, Controller” system architecture pattern. Other design patterns will be considered later in conjunction with a more in-depth class analysis.

---

<sup>2</sup> as well as a myriad of other reasons

## 4.6 GUI Prototypes

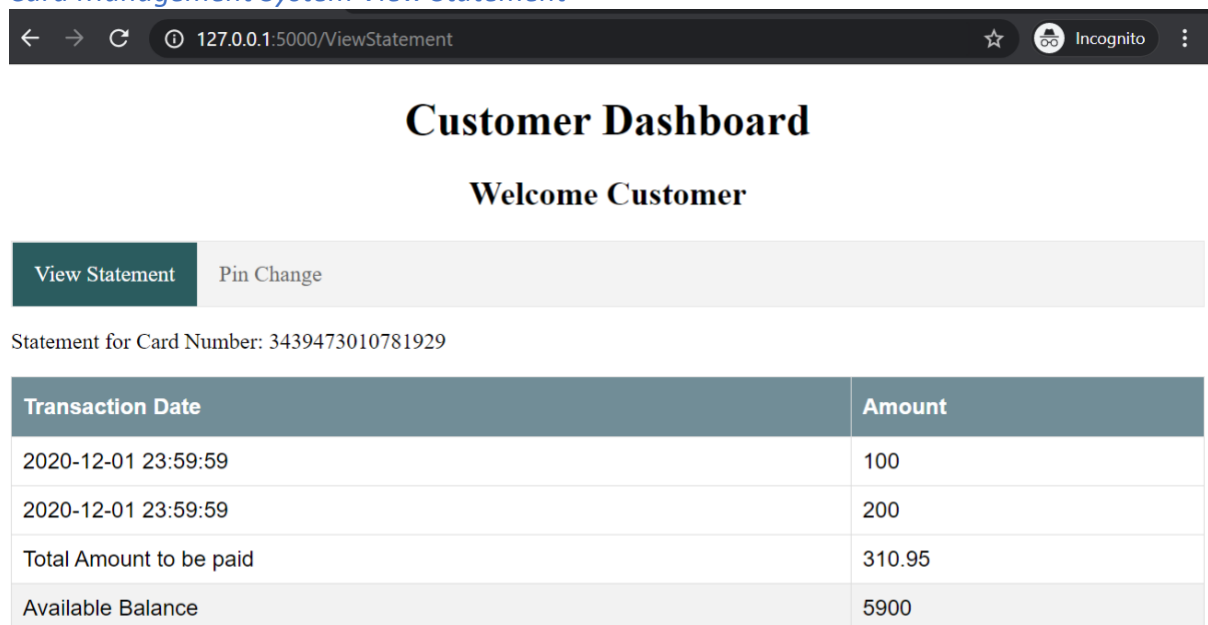
### Card Management Login Screen



A browser window showing the login screen for the Card Management System. The address bar displays '127.0.0.1:5000/login'. The page has a teal background with the title 'Card Management System' in large black font. Below the title, it says 'Login here.' followed by two input fields: one for the email 'rh@gmail.com' and one for a password represented by dots. A 'Submit' button is below the password field. At the bottom, there is a link 'Enroll for Card' in purple.

Figure 6 GUI Prototype Log In

### Card Management System View Statement



A browser window showing the 'View Statement' page for the Card Management System. The address bar displays '127.0.0.1:5000/ViewStatement'. The page has a dark header with the title 'Customer Dashboard' and a subtitle 'Welcome Customer'. Below the header, there are two tabs: 'View Statement' (active) and 'Pin Change'. The main content area shows the statement for Card Number: 3439473010781929. It contains a table with transaction details.

Transaction Date	Amount
2020-12-01 23:59:59	100
2020-12-01 23:59:59	200
Total Amount to be paid	310.95
Available Balance	5900

Figure 7 GUI Prototype: View Statement

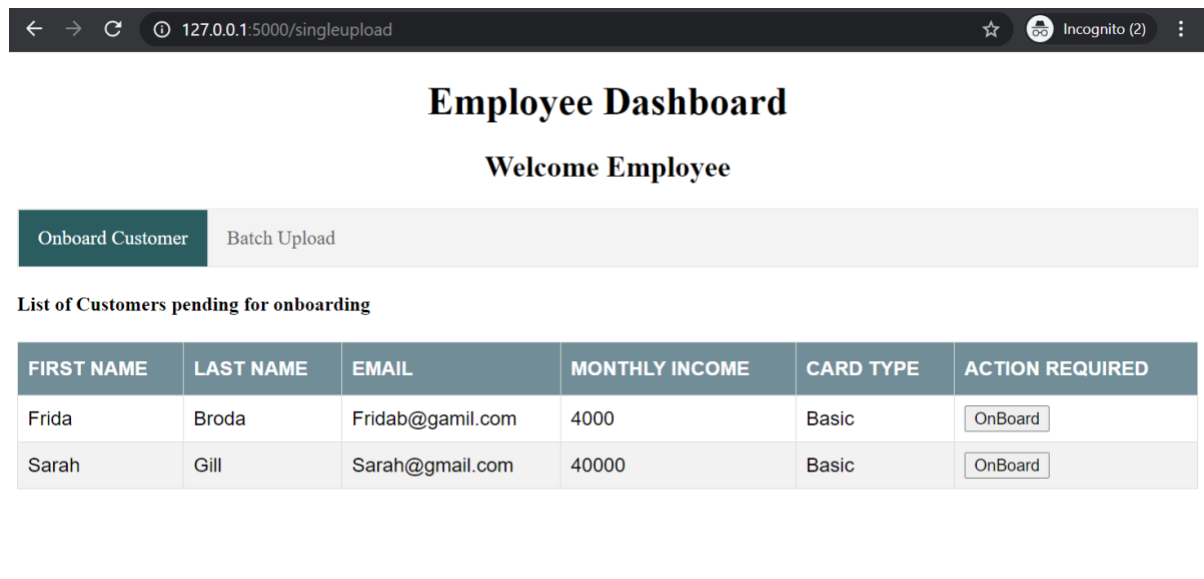


Figure 8 GUI Prototype: Onboard Customer

## 5 System Architecture

### 5.1 Package Diagram

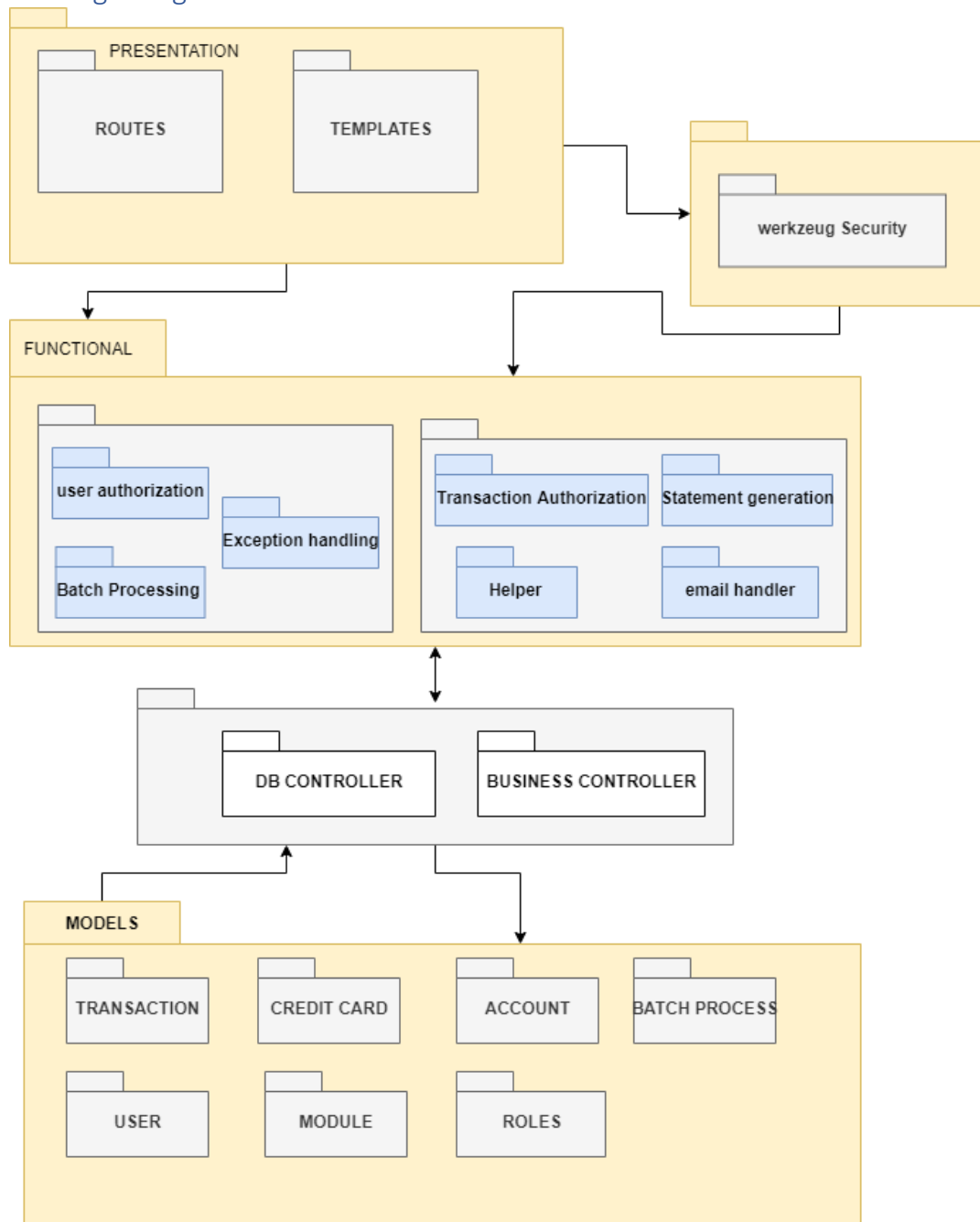


Figure 9 System Package Diagram

A package is a collection of artefacts which are logically grouped together. A quality of good design is the decomposition of system into cohesive packages. The package diagram is a way to represent packages in the context of the whole system. The package diagram is a useful for system maintenance as it is clear and readable representation of the internal structure of the system. The credit card management system groups the packages into functional, security, models, presentation. Presentation package contains the user interface along with routes that help in flow of control. The interface interacts with the user and fetches data, once the user is verified. Login is initiated, other functional services like exception handler, email handler, helper, user - authorization, Batch processing these interact with database and business controller to fetch data from model.

## 5.2 Technologies

The system is implemented using the Python programming language. The software technology used in the implementation of the credit card management system are Flask and MySQL.

### 5.2.1 Flask

Flask is a micro web framework which is written for Python which realises user interfaces using the Jinja2 template engine and uses Werkzeug as the WSGI web application library. While it is a micro web framework, it is highly extensible. Flask does not provide a database abstraction layer or form validation itself, however it allows seamless integration of these extra functionalities through extensions. The way in which flask is used is highly customisable and the flexibility to choose from numerous suitable extensions is an attractive feature of flask. [3]

#### 5.2.1.1 Flask Extensions

Similar in nature to the flask framework, flask extensions provide extra functionalities in a simple and developer friendly manner.

#### *Flask-SQLAlchemy*

The flask-SQLAlchemy extension for flask is an extension which provides support for MySQL in flask applications. It simplifies MySQL function calls from within the source code and provides practical defaults for accessing a MySQL database. [4]

#### *Flask-Login*

Flask-Login provides a simple and customisable way to manage user sessions in a flask applications. This extension stores user details during a session, it does not however, impose constraints on the user storage or authentication mechanisms. [5]

### 5.2.2 MySQL

The database used is MySQL. This database was chosen because it is fast, open source and reliable. MySQL boasts excellent data security and on-demand scalability. As set out in the non-functional requirements, these are desirable features for our system. In addition to this, MySQL is the de-facto industry standard which will aid in the marketing of our software product.

### 5.2.3 PyMySQL

PyMySQL is the interface for connecting Python to the MySQLdb database. Originally, MySQLdb was used, however upon trial PyMySQL was easier to install and behaved better than MySQLdb.

## 5.3 System Architecture

The system architecture is influenced by our use of Flask and MySQL. A flask application is an instance of a flask class and everything inside the application is stored in this class. A flask application has a certain required file structure consisting of an app initialisation file, routes and templates. This existing structure lends itself to the creation of packages. For example, web application route functions should naturally be packaged together as their functionality is the same.

Starting with the user interfaces and working down toward the database in a layered approach, the routes package is interacted with by the system user so appears in the view layer. It is through that routes that users navigate between pages and it is the routes which render templates on webpages.

The business logic of the system is captured in a business controller package. Within this package are classes which manipulate data according to the messages passed by the routes. At this level there is also a package of classes which are database model classes. These database model classes are classes which represent tables in the database.

A level down from this is the database controller package. The database controller package is consisting of classes which can query and update the database. The credit card management system uses python flask. As flask follows MVC pattern. Flask offers object relational mappers, form validations, jinja templates, ORMs, security .

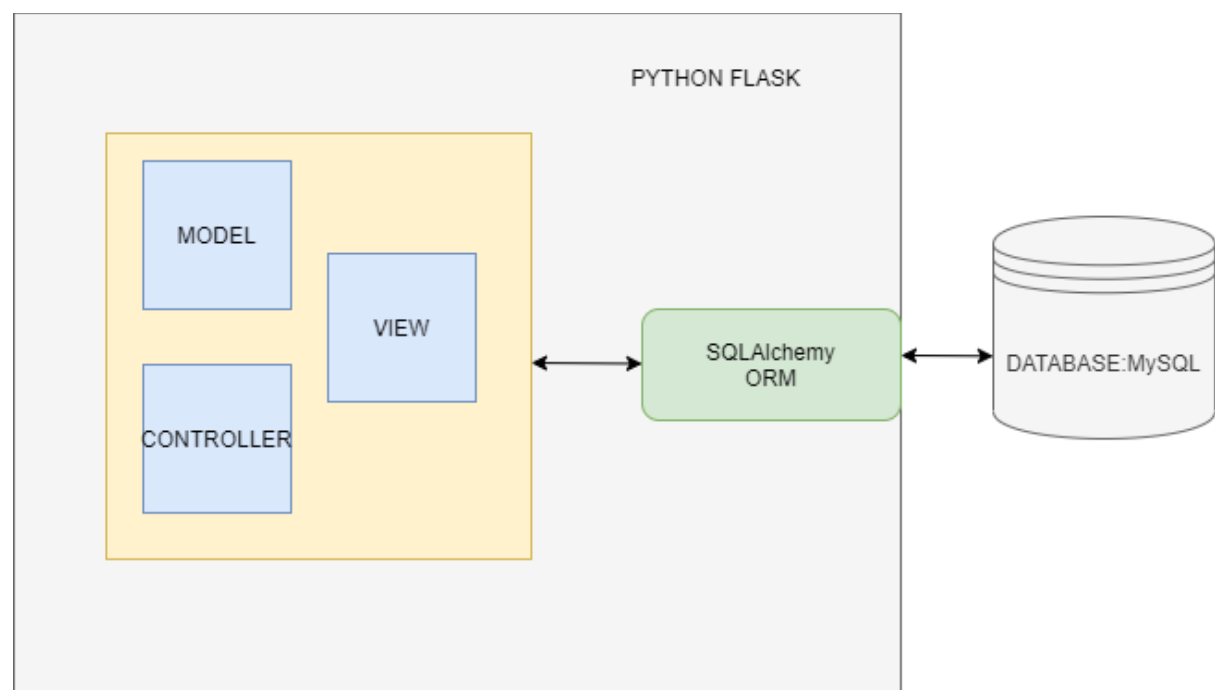


Figure 10 System Architecture diagram

#### 5.4 Run Time Processing

Different objects which are actively executing at the same time must be executed on different logical processors. As objects which are found within the route package will be active at the same time as the business controller objects who will be active at the same time as database controller objects, it is reasonable to assign these three packages to different logical processors. The credit card management system uses python flask which in turn uses the python runtime environment 3. The application deployed on a Google cloud Platform runs

1. On mentioning the entry point.
  2. app.py set as root in the project location
  3. Entry point should be lightweight, each time an instance of application is created
  4. Entry point tunes the performance of the app
  5. we can create number of workers servicing the app for each instance.
- Additional environmental variables could be added in the app.yaml file

In a client server application, it could manage the https, all incoming http requests are validated at the load balancer and later forwarded to the application

File system during runtime involves read only access except from the location /tmp (which is the virtual disk storing data in applications RAM). A metadata instance would be able to query information about the instance of the project.

## 6 System Analysis

### 6.1 Candidate Class Identification

The noun identification technique was used to identify candidate classes. Using the descriptions of how users interact with the system from section 4.1, the use case diagrams and key use case descriptions, a list of all occurring nouns was made. These identified nouns were then listed as potential classes.

Heuristics were then used to discard classes from the list of potentials. With every noun the question, “if this is a class, what are its attributes and methods” was considered. Any nouns which were too vague, described the same thing, could be considered an attribute/operation/association were struck off the list. Through discussion of scenarios in which potential classes were necessary/unnecessary and the consideration of which classes would be required to pass messages to one another, a short-list of potential classes was decided on. This process was iterative, and classes were added and removed from the list several times before a consensus was reached.

While the use of heuristics in this method may seem to lack theoretical grounding and analytic framework, it is important to remember that the optimal conceptual class model of a system is in no way unique. While some might be better than others, different class models can also be of equal quality. The heuristic approach is thus symptomatic of the subjectivity and variety of quality class design.

### 6.2 Analysis Class Diagram

The diagrams below are the analysis class diagrams identifying the main classes in the Card Management system.

The below diagram Figure 11 Analysis Class Diagram 1 showFigure 11 Analysis Class Diagram 1 shows the classes involved in Logging in and Authorizing a User to the Card Management System. Depending on the Role of the User and Modules the user has permission to, a respective Dashboard would be returned. The class diagram shows the dashboard of different types for users and the component classes.



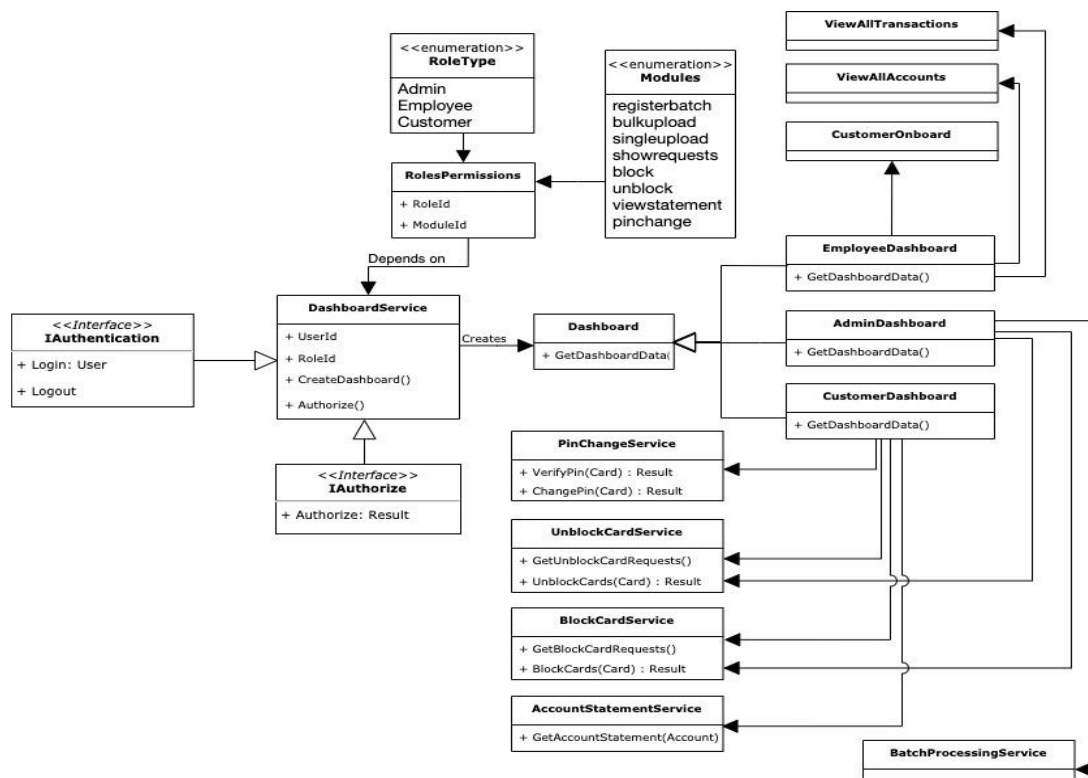


Figure 11 Analysis Class Diagram 1

The below diagram Figure 12 Analysis Class Diagram 2\_ shows the analysis class diagram for a New Customer Onboarding who would enroll for a Credit card. A new User would enroll for a card through EnrollCardService that would be a CustomerApplication. An Admin will onboard that New Customer through the CustomerOnboardService. An Admin can also onboard preapproved users in a batch, this can be done via uploading a file generated by an external system. Through the CustomerApplication, a new User, Account and Card will be generated. The Card would be generated based on user preference and verification at the time of enrolment.

## Customer Onboard

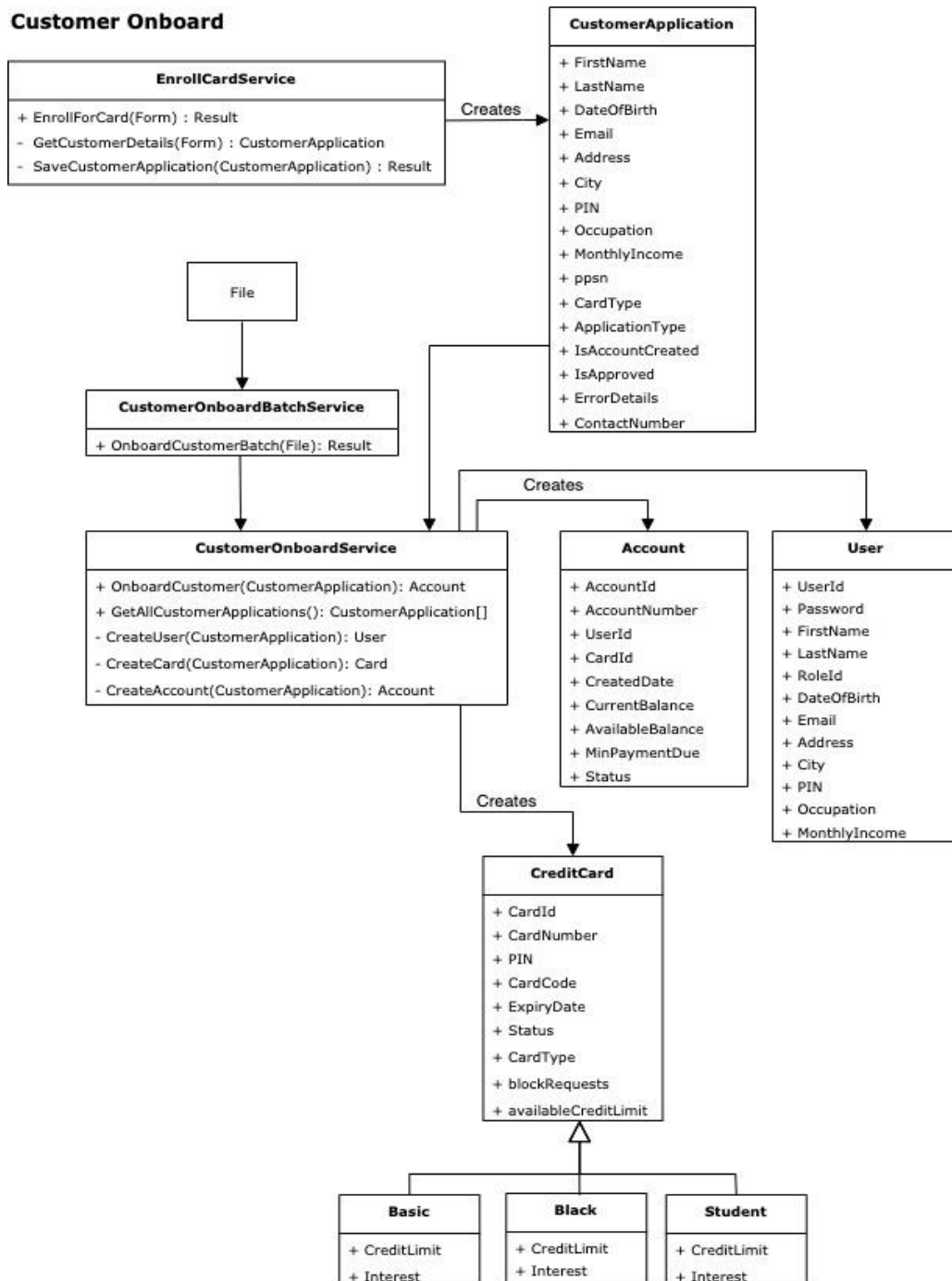


Figure 12 Analysis Class Diagram 2

The below diagram Figure 13 Analysis Class Diagram 3\_ shows the classes and interface involved in Creating Batch Services. Several scheduled processes were identified for a Card Management System that involved settlement of transactions of a credit card at the end of the day, generating account statements, etc. A copy of transactions would be scheduled to be created for the purpose of reporting which is represented by UpdateReportingDB. These batches would run at the scheduled times.

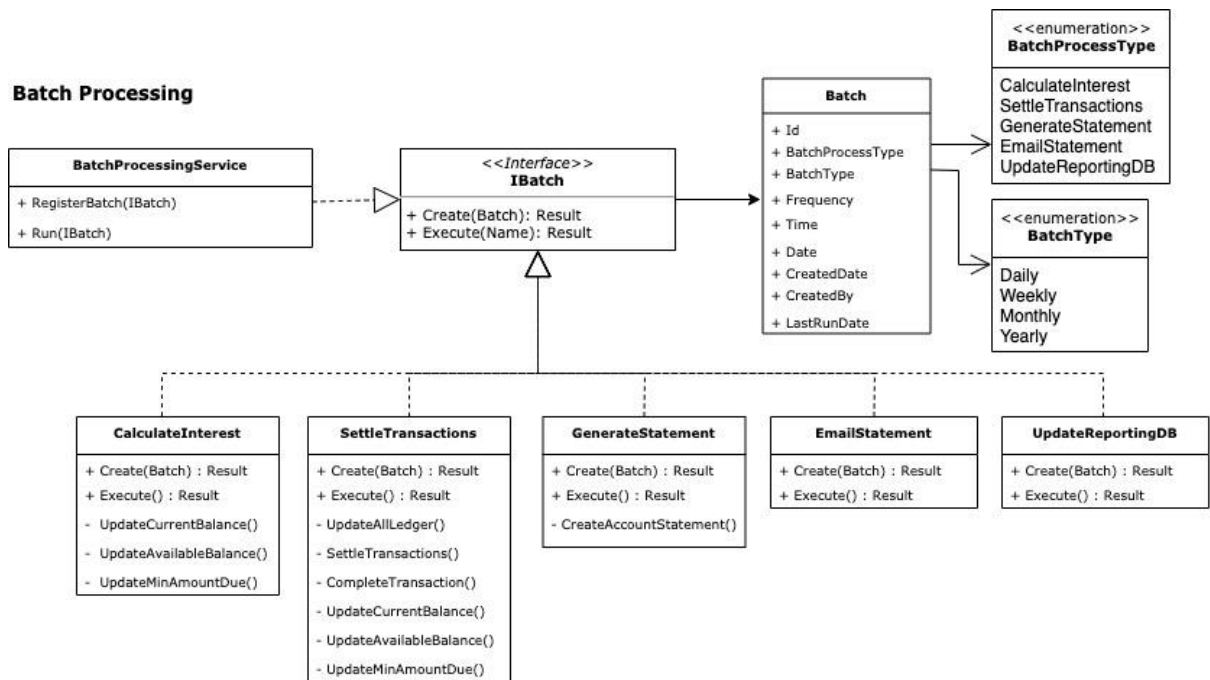


Figure 13 Analysis Class Diagram 3

The following diagram Figure 14 Analysis Class Diagram 4 is the class diagram to show a transaction request coming into the system from an external system. These transactions would be authorized by the card management system where the account and the card will be verified. An appropriate response would be sent to the external system.

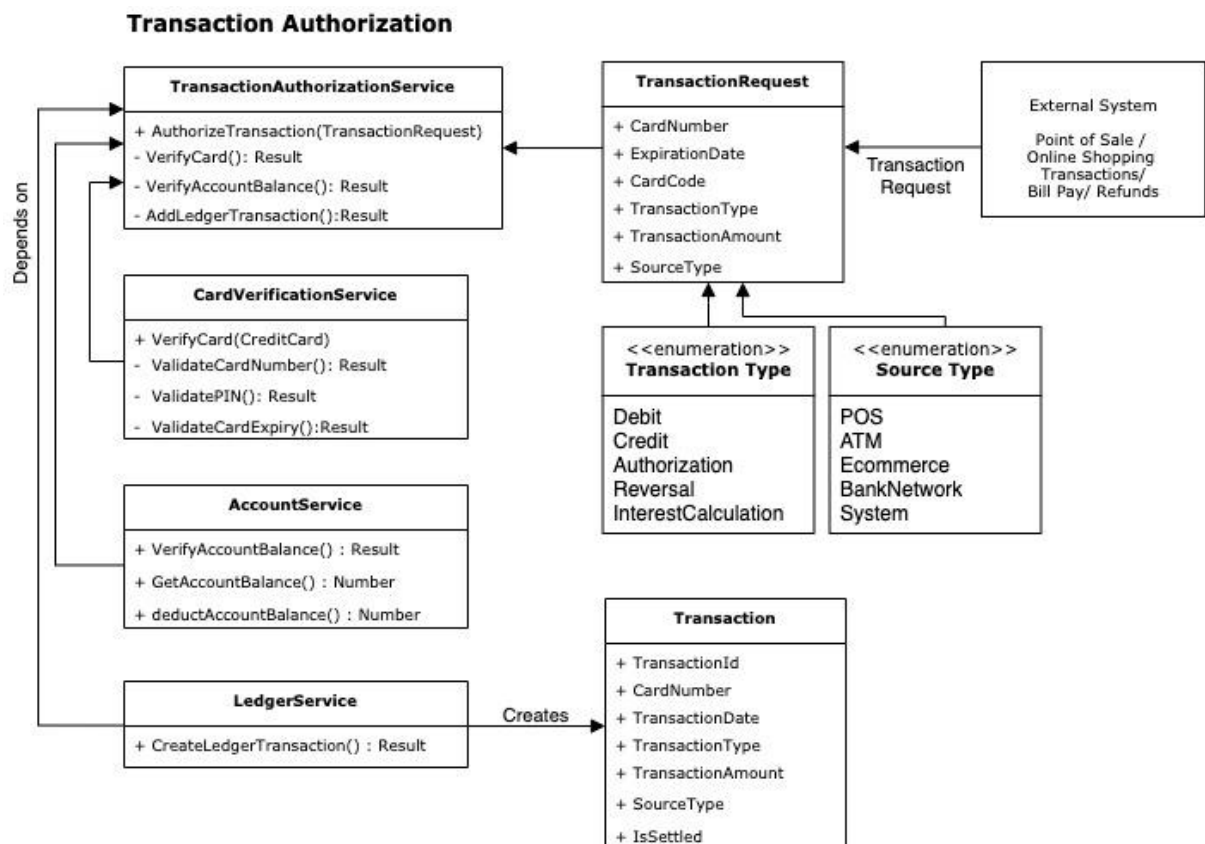


Figure 14 Analysis Class Diagram 4

### 6.3 Communication Diagram for Key Use Cases

Communication diagrams- formerly known as collaboration diagrams enhance the interaction between the components of the system. These diagrams focus more on the links between the parts of the software system. To be precise, communication diagrams display the messages that are passed between the components of a system. They are a good way to model the message flow in the system. The diagrams in the following sub-sections model the key use cases of our system.

#### 6.3.1 System User Log In

The below diagram shows how the different components interact when the user logs in to the system.

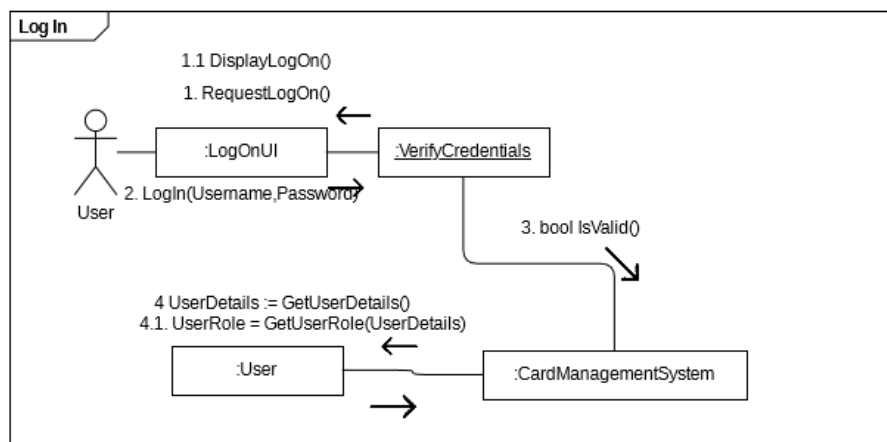


Figure 15 Log In Communication Diagram

#### 6.3.2 Approve Card

Communication diagram here shows how an employee approves card by verifying the user details entered by the user.

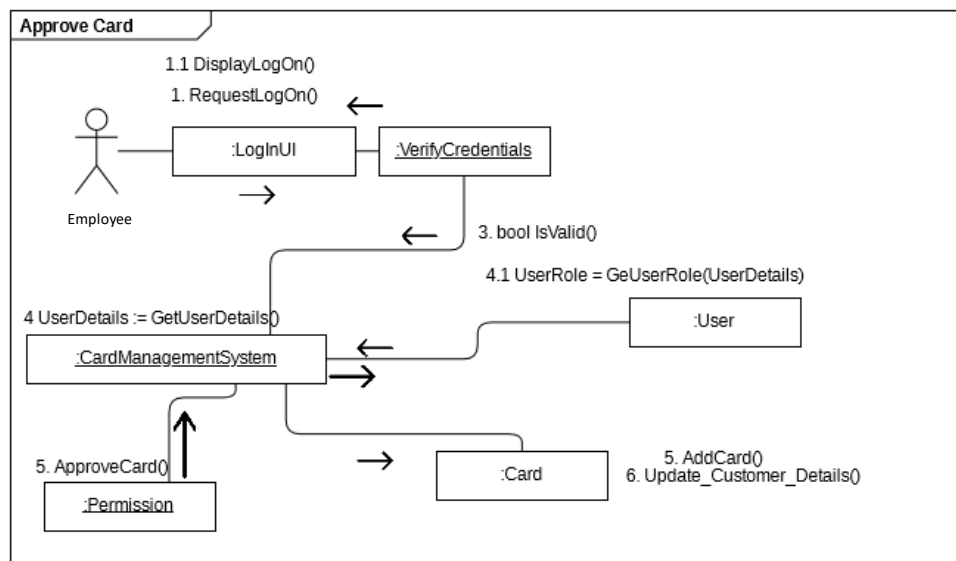


Figure 16 Employee Approve Card Communication Diagram

### 6.3.3 Customer Application

Customer applying for the card is an important workflow and the diagram below shows how components in the pass on messages to carry out the task of creating the customer account and assign them with the requested card.

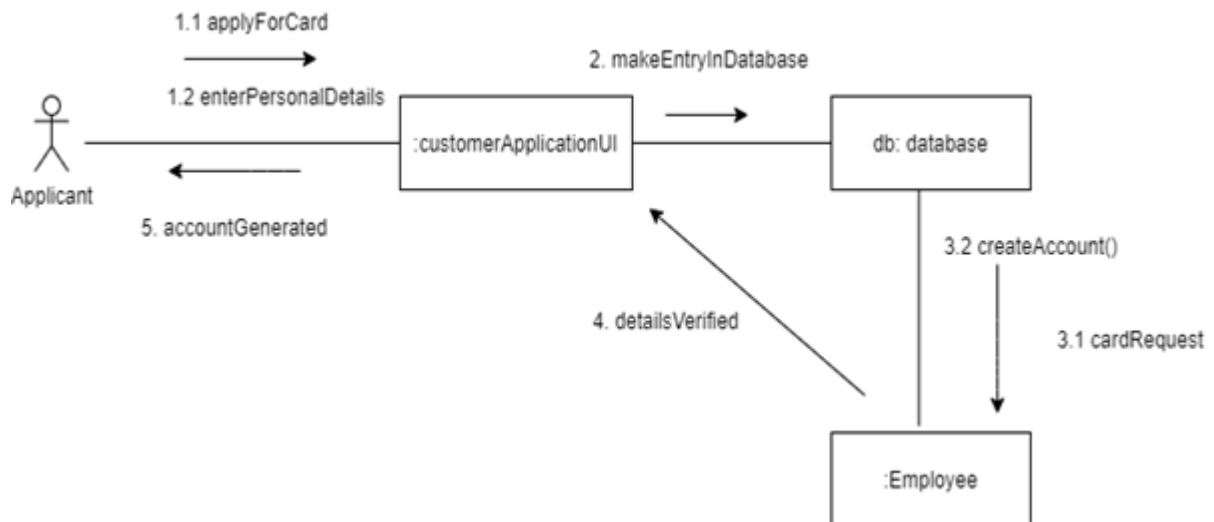


Figure 17 Customer Application Communication Diagram

### 6.3.4 View Statement

Once the customer has been onboarded and starts using the Card, they will require a record of those transactions. View statement functionality does the same for the user and the diagram below shows a comprehensive set of interactions between different parts of the system to generate the statement.

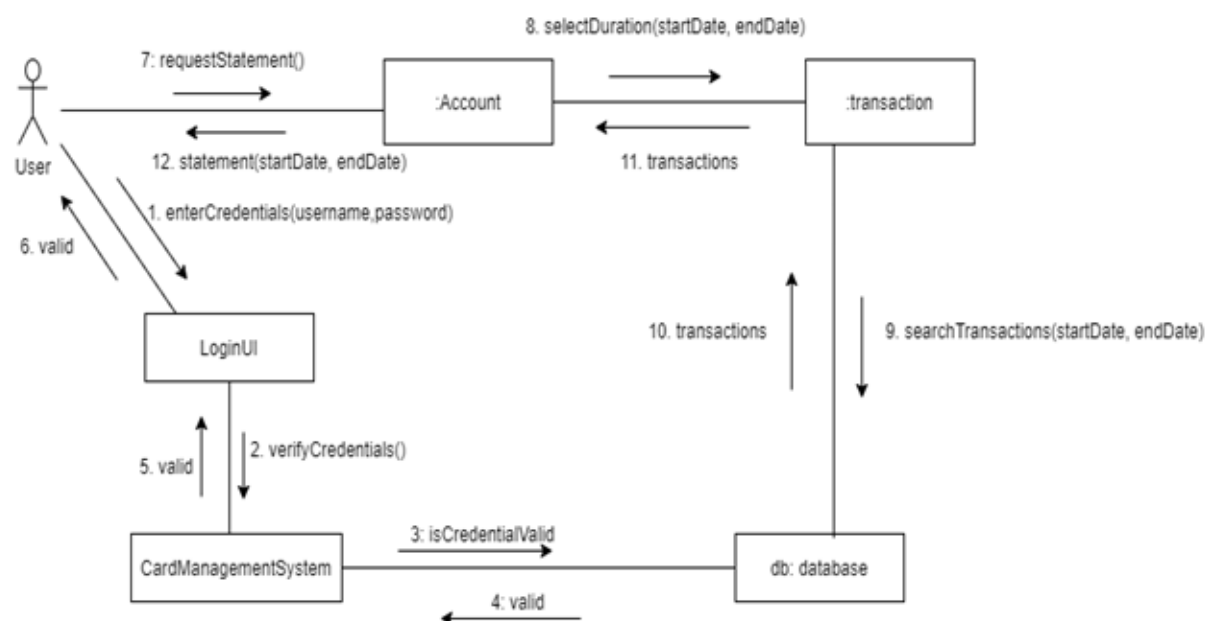


Figure 18 View Statement Communication Diagram

## 6.4 Key Use Case Sequence Diagrams

Sequence diagrams are UML diagrams which depict how objects interact over a time frame. The purpose of sequence diagrams is to capture the order of message passing between objects when a certain scenario is unfolding within the system. As sequence diagrams show the ordered message passing of objects when a certain functionality of the system is invoked, it is logical to represent the key use cases of the system as sequence diagrams.

### 6.4.1 Approve Card

The sequence diagram below shows the interaction of objects as an employee approves a card application. When the customer is approved the customerOnboardService object creates a user, card and account object associated with the user.

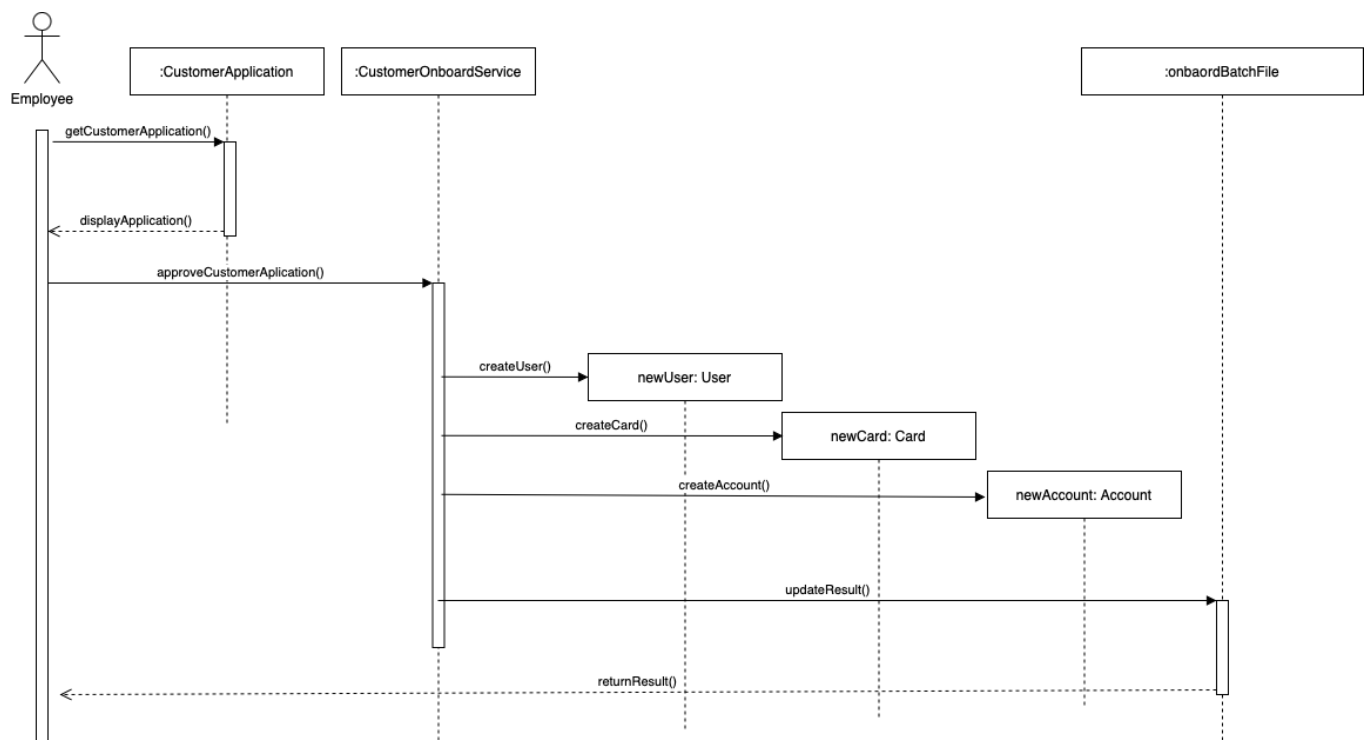


Figure 19 Employee Approve Card Request

### 6.4.2 Transaction Authorisation Request

The transaction authorisation request is an interesting use case to examine because the authorisation request is a message which comes from outside the system. The black dot on the far left of the diagram with the arrow leading to TransactionAuthorisationService represents a message being passed from an unknown source.

The frame labelled opt shows a conditional creation of a transaction and its addition to the ledger. This is guarded by the condition in square brackets; the authorisation of the transaction. The message passing within the square brackets occurs only if the transaction is authorised.

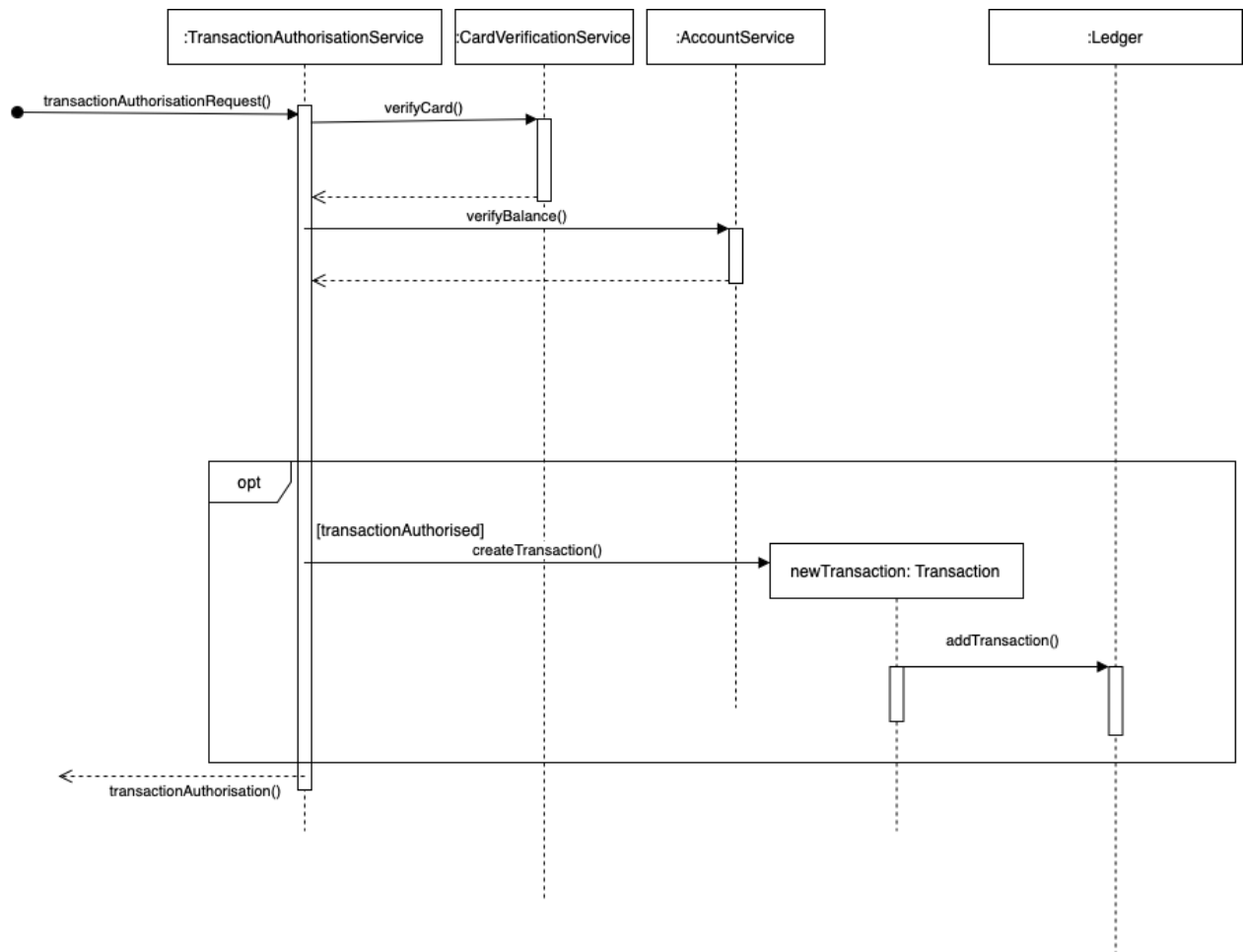


Figure 20 Transaction Authorisation Request Sequence

### 6.4.3 Get Statement

Get statement is another use case where the interaction of several objects occurs. In this instance we examine the message passing beginning at the customer log in. The log in will be the first step of interaction that the customer has with the system and so it is logical to begin the sequence diagram at that point.

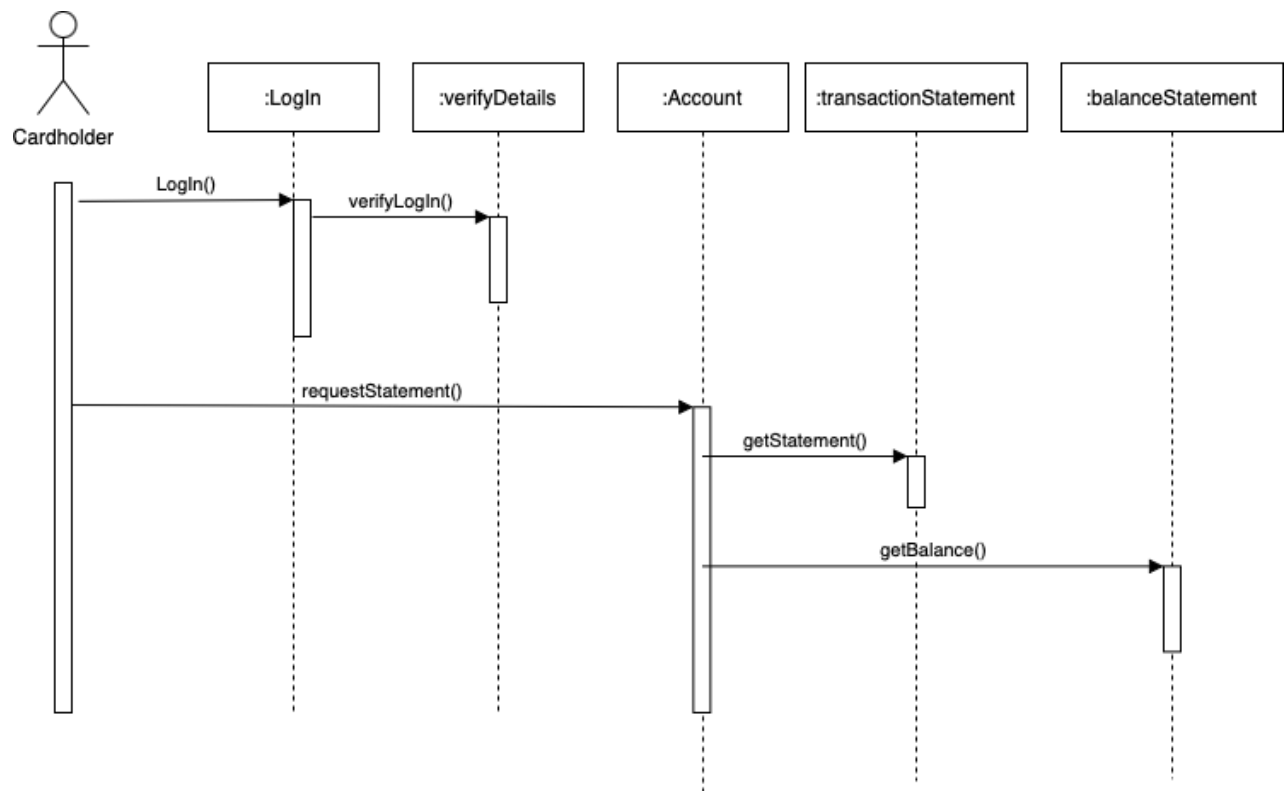


Figure 21 View Statement Sequence Diagram

## 6.5 Entity Relationship Model (E-R model)

Entity relationship model describes the important aspects of a business workflow. It is a result of a systematic analysis. In other words, it represents the data model or schema which are important for business processes. As the name suggests, the models have entities (represented by rectangular boxes) and the relationships (represented by connectors/lines) between those entities.

ER model is used to implement a database where entities (rectangle boxes) represent tables in the database and each entity have attributes. A connection between two tables show a relation between them by means of storing Primary Keys (of one table) as a Foreign Key in another table.

E-R model, as shown below, was used to implement the database schema for the system. Visualising the model assisted a great deal in the Database implementation. Once all the attributes are listed for each entity it can be converted to tables.

In the below model, we have three master tables namely, 1) role 2) module and 3) roles\_permission. These decide the roles of the users logging into the system and what permissions will be allotted to them. Based on, which user logs in the permissions will be read for them from the module table via the roles\_permission table.



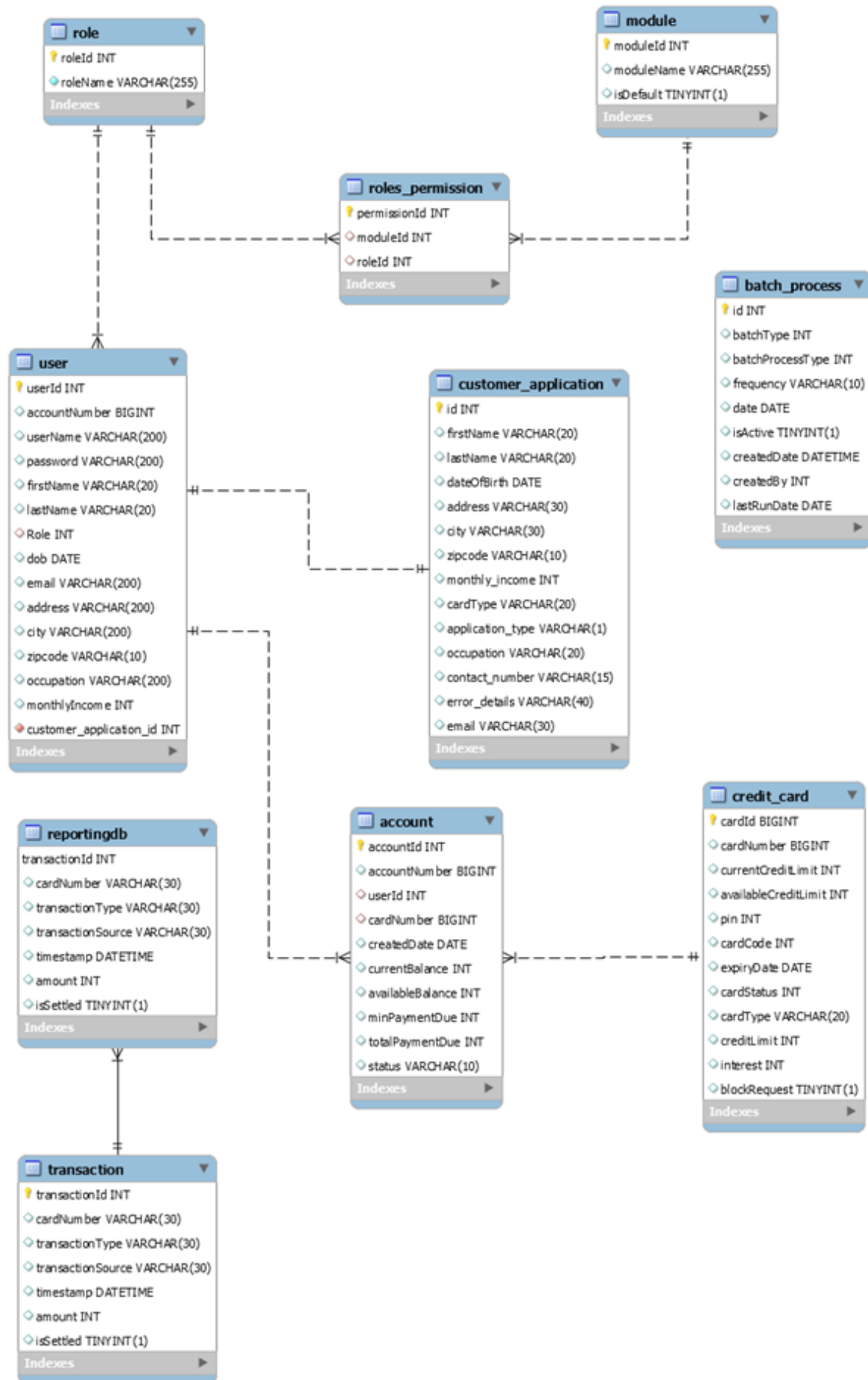


Figure 22 E-R Diagram

## 7 Object Oriented Design (Code)

### 7.1 Design Patterns

We have identified the below design patterns that could be implemented in our proposed system design.

### 7.2 Façade Pattern

According to the Gang of Four, the intent of the Facade pattern is to provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. [6]

#### Implementation:

The TransactionAuthorizationService accepts a Transaction Request and it is followed by a number of validation checks and data is updated and a response is returned to the requesting system. The TransactionAuthorizationService acts as an interface here for the subsystems AccountService, Ledger Service, CardVerificationService. It encapsulates the original actions that are performed under the hood.

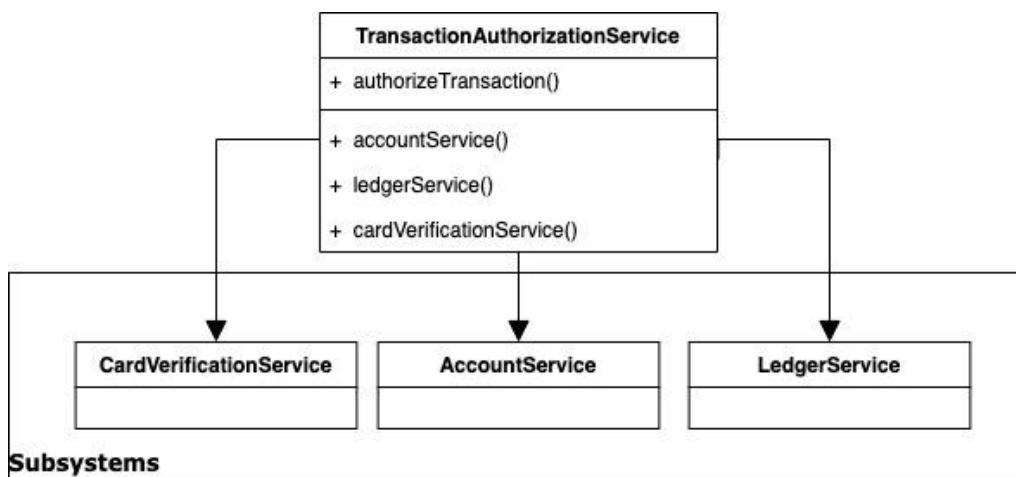


Figure 23 Façade Pattern

Code:

```
class TransactionAuthorizationService():
    def __init__(self):
        self.cardVerificationService = CardVerificationService()
        self.accountService = AccountService()
        self.ledgerService = LedgerService()

    def authorizeTransaction(self, transaction):
        resultCard = self.cardVerificationService.verifyCard(transaction.cardNumber)
        resultAccount = self.accountService.verifyAccountBalance(transaction.cardNumber, transaction.amount)
        if resultCard.isSuccess and resultAccount.isSuccess:
            resultLedger = self.ledgerService.createLedgerTransaction(transaction)
            return resultLedger
        else:
            if not resultCard.isSuccess:
                return resultCard
            elif not resultAccount.isSuccess:
                return resultAccount
```

Figure 24 Transaction Authorisation Service Class

```
class LedgerService():

    def __init__(self):
        self = self

    def createLedgerTransaction(self, transEntry):
        # Use Try and Exception
        transDB=TransactionDBController()
        transaction = Transaction()
        transaction.cardNumber=transEntry.cardNumber
        transaction.transactionType=transEntry.transactionType
        transaction.transactionSource=transEntry.transactionSource
        transaction.timestamp=transEntry.transactionDate
        transaction.amount=transEntry.transactionAmount
        transDB.addTransaction(transaction)
        return Result(True, "Transaction Added")
```

Figure 25 Ledger Service Class

```
class AccountService():

    def __init__(self):
        self = self

    def verifyAccountBalance(self, cardNum, amount):
        accountDB = AccountDBController()
        account = accountDB.getAccountByCardNumber(cardNum)

        hasBalance = account.availableBalance > amount
        if hasBalance:
            return Result(hasBalance, "Account Balance Verified")
        else:
            return Result(hasBalance, "Not enough Account Balance")

    def deductAccountAvailableBalance(self, amount, cardNum, transType):
        accountDB = AccountDBController()
        #transType - Depending on transaction type the amount would be added or deducted
        account = accountDB.getAccountByCardNumber(cardNum)
        accBalance = account.availableBalance - amount
        accountDB.setAccountBalance(accBalance)
```

Figure 26 Account Service Class

```

class CardVerificationService():

    def __init__(self):
        self = self

    def verifyCard(self, cardNum):
        valid = self.validateCardNumber(cardNum)
        if(valid):
            return Result(valid,"Card Verified")
        else:
            return Result(valid,"Card Invalid")

    def validateCardNumber(self, cardNum):
        cardDB = CardDBController()
        card = cardDB.getCardByNumber(cardNum)
        if card:
            return True
        else:
            return False

```

Figure 27 Card Verification Service Class

### 7.3 Builder Pattern

A builder pattern is used because it encapsulated the way a complex object is created. It allows objects to be constructed in a multistep and varying process. It hides the internal representation of the product from the client. Product implementations can be swapped in and out because the client only sees an abstract interface. [7]

#### Implementation:

When users are onboarded to the card management system, they are customer applications. Every Customer will then have an Account created i.e. a Credit Card account. The Credit Card account will have a User and a Card associated with it. We have implemented the Builder pattern to build an Account Object.

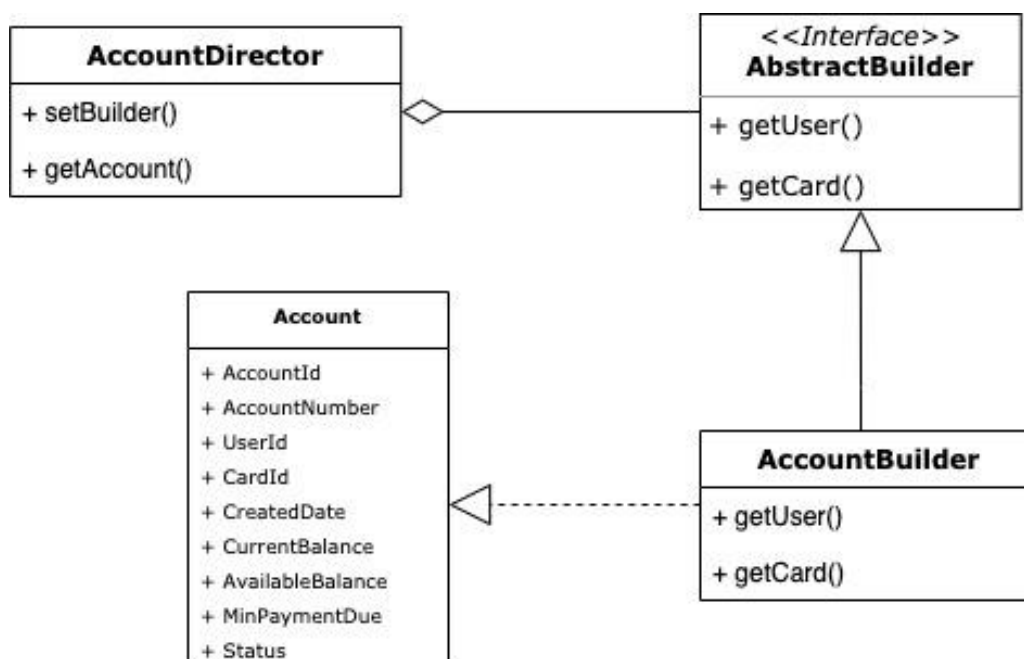


Figure 28 Builder Pattern

Code:

```
class CustomerOnBoardService():

    def __init__(self):
        self = self

    def onBoardCustomer(self,application):
        #Create Account using Builder Pattern
        accountBuilder = ab.AccountBuilder()
        accountDirector = ad.AccountDirector()
        accountDirector.setBuilder(accountBuilder)
        account = accountDirector.getAccount(application)

        cardDB = CardDBController()
        cardDB.addCard(account.card)

        userDB = UserDBController()
        userDB.addUser(account.user)
```

*Figure 29 Customer Onboard Service Class*

```
class AbstractBuilder(abc.ABC):

    @abc.abstractmethod
    def getUser(self,customerApplication):
        pass

    @abc.abstractmethod
    def getCard(self,customerApplication):
        pass
```

*Figure 30 Abstract Builder Class*

```

class AccountBuilder(abs.AbstractBuilder):

    def getUser(self, customerApplication):
        randomVar=Randpass()
        user = User()
        user.userId = randomVar.userGen()
        user.accountNumber = randomVar.accGen()
        user.userName = customerApplication.lastName + ',' + customerApplication.fi
        user.firstName = customerApplication.firstName
        user.lastName = customerApplication.lastName
        user.password = generate_password_hash(randomVar.passGen(), method='sha256
        user.Role = Role(Role.Customer).value
        user.dob = customerApplication.dateOfBirth
        user.email = customerApplication.email
        user.address = customerApplication.address
        user.city = customerApplication.city
        user.zipcode = customerApplication.zipcode
        user.occupation = customerApplication.occupation
        user.monthly_income = customerApplication.monthly_income
        return user

    def getCard(self, customerApplication):
        randomVar=Randpass()

        card = CreditCard()
        card.cardNumber = randomVar.cardGen()

```

Figure 31 Account Builder Class

```

class AccountDirector:

    def setBuilder(self, builder,):
        self.builder = builder

    def getAccount(self, customerApplication):
        randomVar=Randpass()
        account = Account()
        #User
        user = self.builder.getUser(customerApplication)
        #Card
        card = self.builder.getCard(customerApplication)

        #Account
        account.user = user
        account.userId = user.userId
        account.setCard(card)
        #account.card = card
        account.cardNumber = card.cardNumber
        account.currentBalance=card.creditLimit
        account.availableBalance=card.creditLimit
        account.createdDate=datetime.datetime.now()
        account.status=Status(Status.Active).value
        account.accountNumber=user.accountNumber

        return account

```

Figure 32 Account Director Class

## 7.4 Factory Pattern

The Factory Method is a pattern intended to help assign responsibility for creation. According to the Gang of Four, the intent of the Factory Method is to define an interface

for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. (Shalloway, A. (2004). *Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition*. Addison-Wesley Professional)

### Implementation:

We have implemented the factory pattern to create Dashboard for our Users of the Card Management System. Depending on the user type logging in and the requested URL, the Dashboard Factory would create the respective Dashboard.

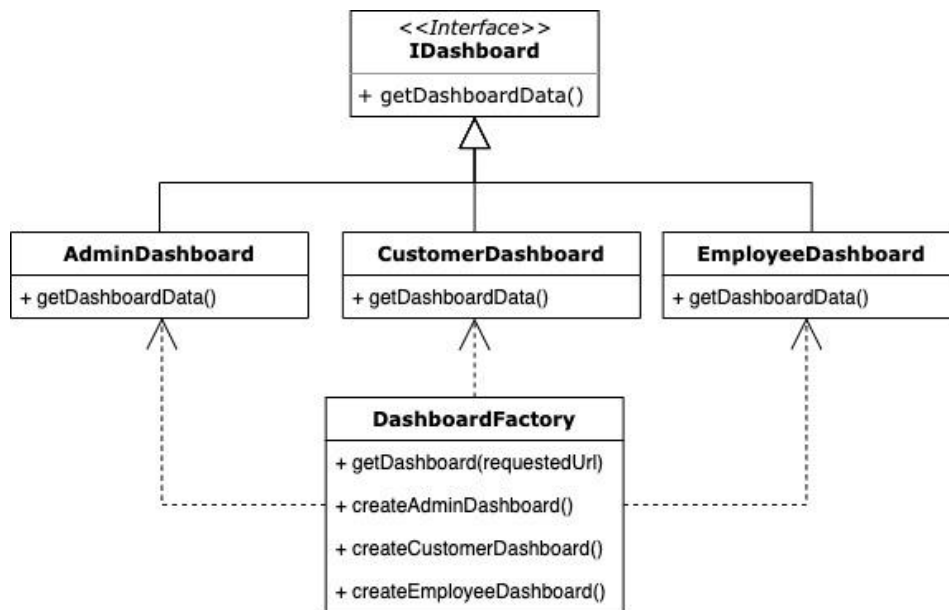


Figure 33 Factory Patter Diagram

### Code:

```

class IDashboard(abc.ABC):
    @abc.abstractmethod

    def getDashboardData(self, requestedPage):
        pass
  
```

Figure 34 Dashboard Interface

```

class EmployeeDashboard(IDashboard):

    def __init__(self, template=None):
        self.template = template

    def getDashboardData(self, requestedPage):
        if requestedPage == 'singleupload':
            app_type = '0'
  
```

Figure 35 Employee Dashboard Class

```

class CustomerDashboard(IDashboard):

    def __init__(self, template=None):
        self.template = template

    def getDashboardData(self, requestedPage):
        self.url = '/customerdashboard'
        self.dashboardName = "Customer Home"
        self.template = "CustomerHome.html"
        return self

```

Figure 36 Customer Dashboard Class

```

class DashboardFactory():

    def __init__(self):
        self = self

    def getDashboard(self, url):

        request = url.split('/')
        if request[0] == Role(Role.Admin).name:
            return AdminDashboard()
        elif request[0] == Role(Role.Employee).name:
            return EmployeeDashboard()
        elif request[0] == Role(Role.Customer).name:
            return CustomerDashboard()

```

Figure 37 Dashboard Factory Class

## 7.5 Decorator Pattern

According to Gang of Four, the Decorator pattern's intent is to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality (Shalloway, A. (2004). *Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition*. Addison-Wesley Professional)

### Implementation:

We have implemented the decorator pattern to decorate Credit Cards for our system. Depending on the type of Credit Card, the Credit Limit and Interest would differ. These are attached to the credit card object dynamically.



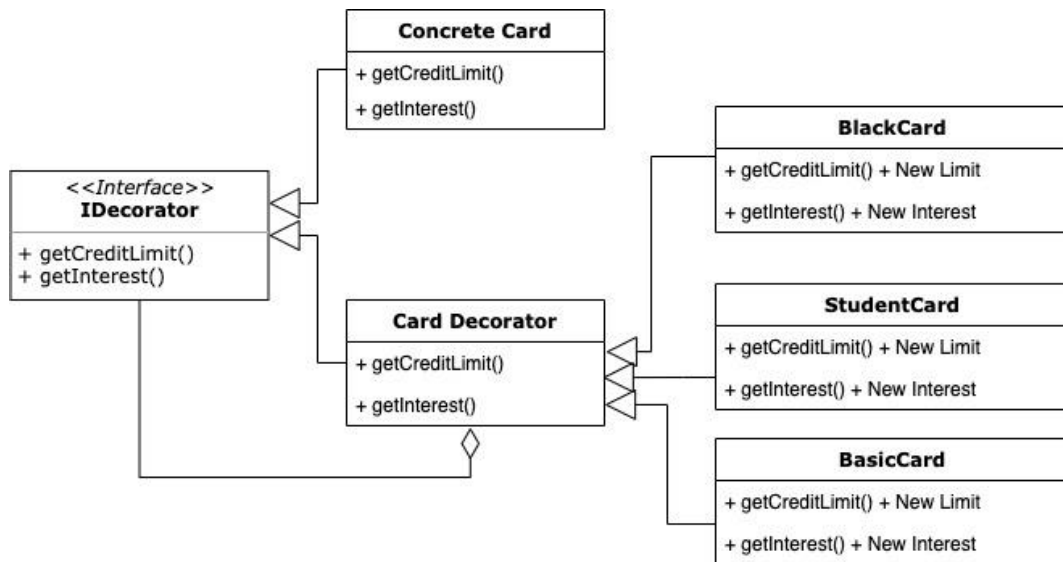


Figure 38 Decorator Pattern Diagram

Code:

```

class CardDecorator(IDecorator):

    def __init__(self, decoratedCard):
        self.decoratedCard = decoratedCard

    def getCreditLimit(self):
        return self.decoratedCard.getCreditLimit()

    def getInterest(self):
        return self.decoratedCard.getInterest()
  
```

Figure 39 Card Decorator

```

class ConcreteCard(IDecorator):

    def getCreditLimit(self):
        return 1000

    def getInterest(self):
        return 1
  
```

Figure 40 Card Class

```

class IDecorator(abc.ABC):
    @abc.abstractmethod

    def getCreditLimit(self):
        pass

    def getInterest(self):
        pass

```

Figure 41 Card Decorator Interface

```

class StudentCard(CardDecorator):

    def __init__(self,decoratedCard):
        CardDecorator.__init__(self,decoratedCard)

    def getCreditLimit(self):
        return self.decoratedCard.getCreditLimit() + 4000

    def getInterest(self):
        return self.decoratedCard.getInterest() - 0.2

```

Figure 42 Student Card Class

```

class BasicCard(CardDecorator):

    def __init__(self,decoratedCard):
        CardDecorator.__init__(self,decoratedCard)

    def getCreditLimit(self):
        return self.decoratedCard.getCreditLimit() + 7000

    def getInterest(self):
        return self.decoratedCard.getInterest() + 0.2

```

Figure 43 Basic Card Class

```

class BlackCard(CardDecorator):

    def __init__(self,decoratedCard):
        CardDecorator.__init__(self,decoratedCard)

    def getCreditLimit(self):
        return self.decoratedCard.getCreditLimit() + 9000

    def getInterest(self):
        return self.decoratedCard.getInterest() + 0.8

```

Figure 44 Black Card Class

## 7.6 Inversion of Control / Dependency Injection

Dependency Injection is a set of software design principles and patterns that enables you to develop loosely coupled code. [8]

Inversion of control or IoC is a Design Pattern that depends on Dependency Injection as the instantiation of the object is the responsibility of the architecture and not the developer.

Batch Processing Service was one of the areas we saw fit to implement this pattern. Figure 13 Analysis Class Diagram 3 shows this.

```
import abc
from models.Result import Result

class IBatch(abc.ABC):

    @abc.abstractmethod
    def Create(self,form) -> Result:
        pass

    @abc.abstractmethod
    def Execute(self,name)-> Result:
        pass
```

Figure 45 Batch Interface

```
from Interfaces.IBatch import IBatch

class BatchProcessingService():

    def __init__(self,iBatch):
        self.iBatch = iBatch

    def Register(self,form):
        return self.iBatch.Create(form)

    def Run(self):
        self.iBatch.Run()
```

Figure 46 Batch Processing Service Class

```
from app import db

class CalculateInterest(IBatch):

    def __init__(self):
        self=self

    def Create(self,form):
        bhelper = bh.batchHelper
        batch = bhelper.createBatchObject(request.form)

    def Execute(self):
        return Result(True,"Batch Run Successfully")
```

Figure 47 Calculate Interest Class

```

@app.route("/registerbatch",methods=['POST','GET'])
def registerBatch():

    bController = bc.BatchController()
    dash = DashboardController()
    data = dash.createDashboard('registerbatch')

    if request.method == 'POST':
        bController.registerBatch(request.form)
        batchlist = bController.getAllbatches()
        return (render_template(data.template,BatchTypes=data.batchTypes, BatchProcessTypes=d

```

Figure 48 Register Batch Class

## 7.7 MVC

We have used the MVC framework for our system, an architectural pattern that divides our application logically into three components, Model, View, and Controller. Each component is built to handle specific development facets of an application separating the business logic from the presentation layer. We have business controllers that control the interaction between Views and Models where Views handle the data shown to the users and Models responsible for maintaining the data.

An application developed using the MVC model can be completed way faster than the application developed using other development patterns supporting rapid and parallel development. It also allows us to create multiple views for a model while any modification done to any component does not affect the entire model making it an efficient architecture.

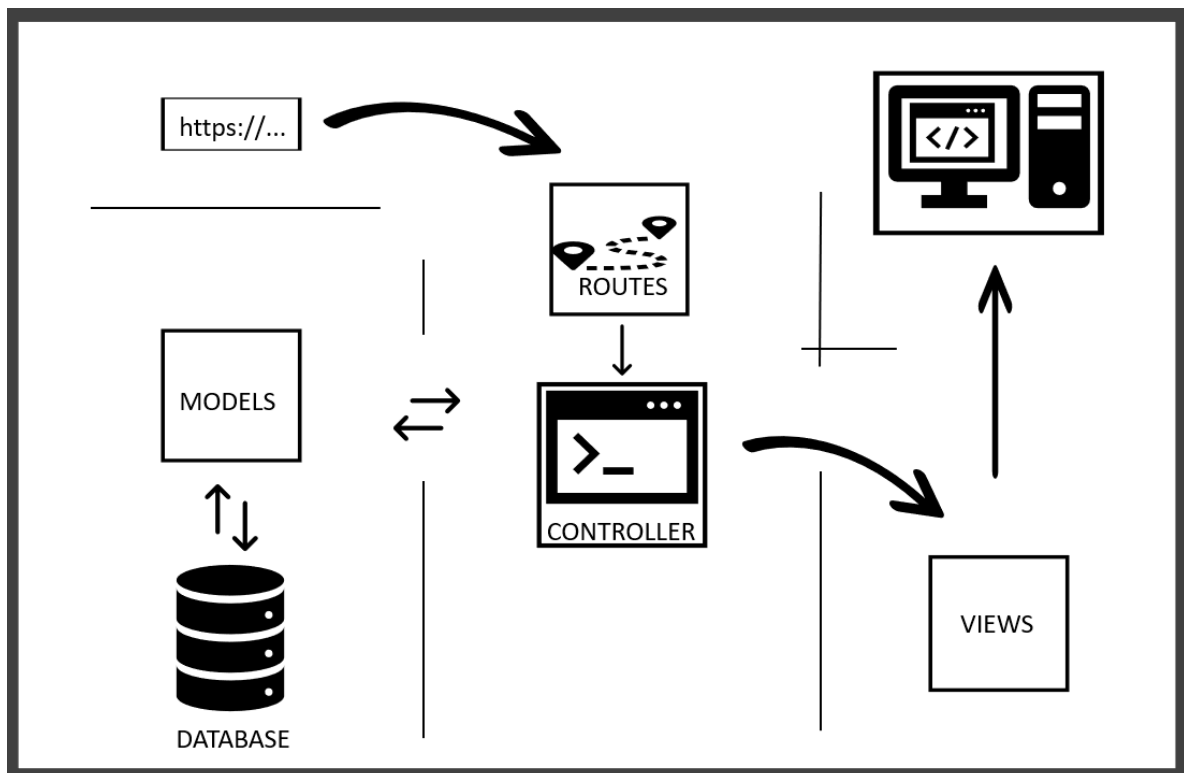


Figure 49 Implemented MVC Diagram

We are using Python Flask for development. We saw python flask a very good fit for using the MVC Pattern. With Python Flask, every method you write is mapped to a route. In the code below, the method registerBatch is mapped to the route “/registerbatch”. Therefore, the

method here acts as a controller. The controller would then interact with the underlying classes to get data, that is our models. The controller uses `render_template` which is a html file to render a page. This is the view. This is how we implemented MVC Pattern using Python Flask.

## 7.8 Case Tools

A number of software development tools were used the course of the project.

### 7.8.1 Draw.io

Draw.io [9] is an online diagram tool which supports UML and other types of software diagrams. It was used to create the final diagrams for the project. In many instances, the several drafts of the diagrams would be drawn on paper before being transcribed online. Draw.io allowed multiple people to edit the same diagram by downloading and uploading an xml file, this was important as the team collaborated virtually.

### 7.8.2 Version Control

This project was hosted on GitHub [10]. All team members were added to the CS5721\_Software\_Design repository. As we implemented the project using Python Flask the framework and organisation of files was important. After the initial upload of a skeleton framework each team member cloned the repository and worked locally. The majority of the coding took place over a two-week period, during this time the team met daily to discuss what they were working on and what was still to be done.

At the beginning of the project branching was implemented but we found it to be unnecessary. Changes to files were small, incremental and frequent. As well as this, each team member was working on different section of the project. Individuals who were working on the same sections made use of a pair programming tactic. We decided that as long as we refrained from pushing broken code (meaning that a level of local testing should be carried out), the risks of merging straight to main from our local repositories were few and small.

The main branch recorded over 88 commits, across these commits, we ran into very few merge conflicts. Had this project been of larger scale with less communication between team members, it would have been necessary to make better use of branching.

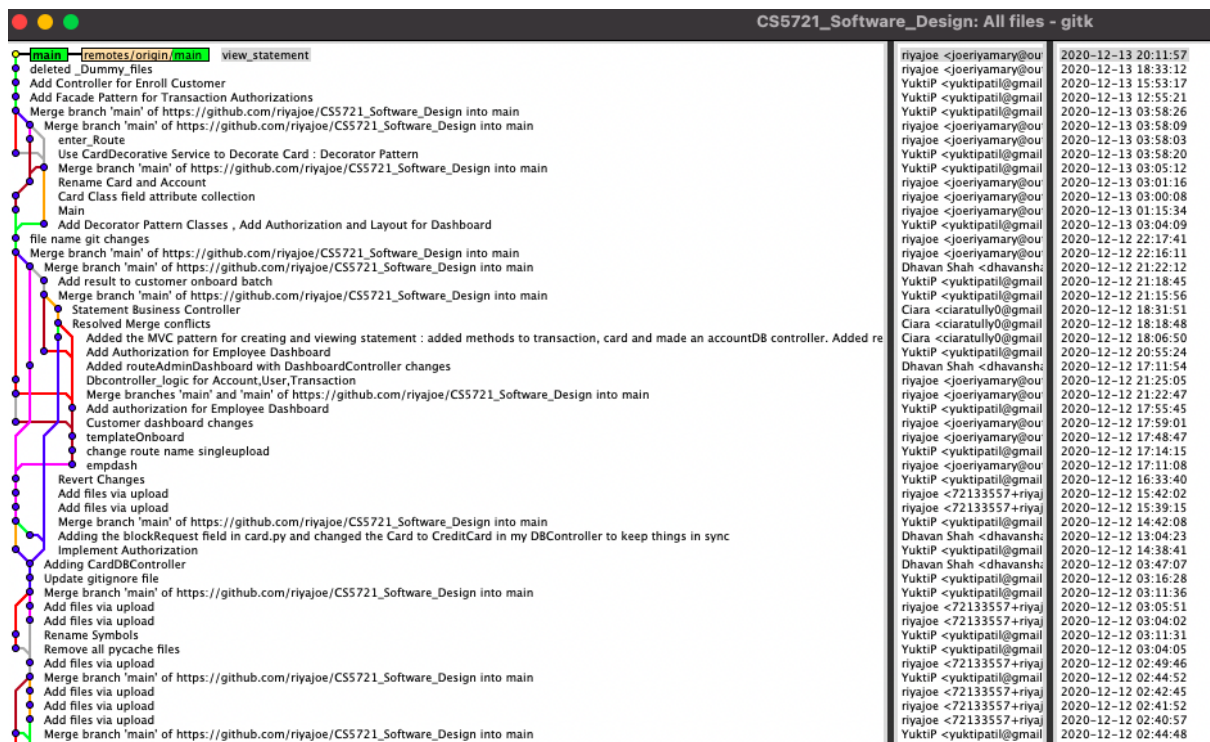


Figure 50 Branching and Merging in GitHub shown using Gitk

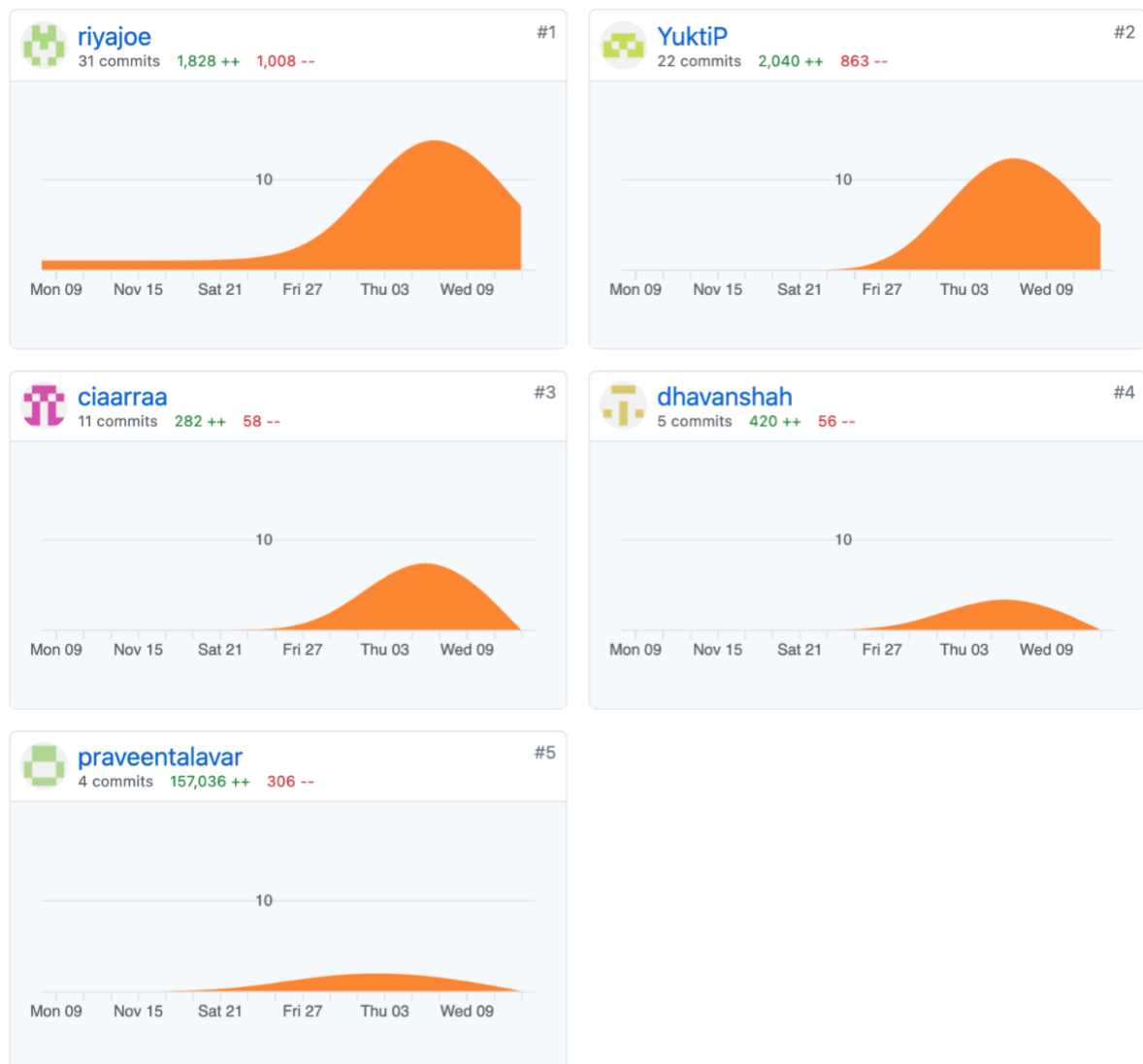


Figure 51 Range of Commits During Implementation

## 7.9 Testing

Test Driven Development approach was used. It refers to an approach where three major tasks are blended closely: development, test and refactor. In this methodology we write tests that tests the smallest part of our functionality, incrementally refactoring the code and testing after each small change. For example, if tests were passing till the last build and started failing now there might be something wrong in the code deployed with the latest build, this precisely takes care of the bugs. Such small tests are called as Unit tests. Such process goes on till the code is released to production significantly reducing, but not all, the bugs.

We took the approach of Continuous Integration (CI) in this system. Here, we integrated code into the shared repository many times a day allowing us to align the code base. CI along with TDD assists in integration of the entire code base helping in reducing the software breakdowns in the later stage when there will be more lines of code.

Unit tests were written to test some important workflows of the system. We used the inbuilt unittest standard library to write the tests for 'login','blockCard' and 'unlockCard' routes. Below are the code snippets for the unit tests.

- 1) Tests the login by logging with the admin credentials and validating the response for a page 'showrequests' in admin dashboard.

```
def setUp(self):
    self.app = app
    app.config['TESTING'] = True
    app.config['WTF_CSRF_ENABLED'] = False
    self.client = self.app.test_client()
    self.ctx = self.app.test_request_context()
    self.ctx.push()

Run Test | Debug Test
def test0_login_user(self):
    print("*****Running test case 1 to check whether admin user is logged in*****")
    response = self.client.get('/login', content_type = 'html/text')
    self.assertEqual(response.status_code,200,'Unable to load login page.' +str(response.status_code)+' was returned')
    with self.client:
        response_post_login = self.client.post('/login', data = dict(email="test@test.com",password="password",submit="submit"),follow_redirects=True)
        self.assertEqual(response_post_login.status_code,200,'Unable to load login page.' +str(response_post_login.status_code)+' was returned')

Run Test | Debug Test
def test1_admin_route(self):
    self.test0_login_user()
    print("*****Running test case 1 to check whether admin route is running correctly*****")
    response = self.client.get('/showrequests', content_type = 'html/text')
    self.assertEqual(response.status_code,200,'Response code is not 200. Instead '+str(response.status_code)+' was returned')

Run Test | Debug Test
def test2_admin_page_loads(self):
    self.test0_login_user()
    print("*****Running test case 2 to check whether admin page is loading correctly*****")
    response = self.client.get('/showrequests', content_type = 'html/text')
    self.assertTrue(b'Welcome Admin' in response.data)
```

Figure 52. Test Code 1

- 2) Tests the login by logging with the admin credentials and validating the block card route by checking whether it is blocking the card.



```

Run Test | Debug Test
def test1_login_user(self):
    print("*****Checking login before the start of testcases*****")
    response = self.client.get('/login', content_type = 'html/text')
    self.assertEqual(response.status_code,200,'Unable to load login page.' +str(response.status_code)+' was returned')
    with self.client:
        response_post_login = self.client.post('/login', data = dict(email="test@test.com",password="password",submit="submit"),follow_redirects=True)
        self.assertEqual(response_post_login.status_code,200,'Unable to load login page.' +str(response_post_login.status_code)+' was returned')

Run Test | Debug Test
def test2_block_route(self):
    self.test1_login_user()
    print("*****Running test case 1 to check whether block route is running correctly*****")
    response = self.client.get('/block', content_type = 'html/text')
    self.assertEqual(response.status_code,200,'Response code is not 200. Instead '+str(response.status_code)+' was returned')
    self.assertTrue(b'Requests to Block Card' in response.data, 'Unable to see the block data')

Run Test | Debug Test
def test3_block_card(self):
    self.test1_login_user()
    print("*****Running test case 3 to check whether block card makes the card inactive*****")
    data_button_1 = {'delete':1234}
    data_button_2 = {'delete':12345}
    self.client.post('/block', data = data_button_1)
    response = self.client.get('/block', follow_redirects = True)
    self.assertTrue(b'The card: 1234 has been blocked successfully' in response.data)
    self.client.post('/block', data = data_button_2)
    response_1 = self.client.get('/block')
    self.assertTrue(b'The Card: 12345 has been blocked successfully' in response_1.data)
    self.assertTrue(b'No card block requests' in response_1.data)

if __name__ == "__main__":
    unittest.main()

```

Figure 53 Test Code 2

- 3) Tests the login by logging with the admin credentials and validating the unblock card route by checking whether it is unblocking the card.

```

Run Test | Debug Test
def test0_login_user(self):
    print("*****Checking login before the start of testcases*****")
    response = self.client.get('/login', content_type = 'html/text')
    self.assertEqual(response.status_code,200,'Unable to load login page.' +str(response.status_code)+' was returned')
    with self.client:
        response_post_login = self.client.post('/login', data = dict(email="test@test.com",password="password",submit="submit"),follow_redirects=True)
        self.assertEqual(response_post_login.status_code,200,'Unable to load login page.' +str(response_post_login.status_code)+' was returned')

Run Test | Debug Test
def test1_unblock_route(self):
    self.test0_login_user()
    print("*****Running test case 2 to check whether unblock route is running correctly*****")
    response = self.client.get('/unblock', content_type = 'html/text')
    self.assertEqual(response.status_code,200,'Response code is not 200. Instead '+str(response.status_code)+' was returned')
    self.assertTrue(b'Requests to Unblock Card' in response.data, 'Unable to see the unblock card data')

Run Test | Debug Test
def test2_unblock_card(self):
    self.test0_login_user()
    print("*****Running test case 3 to check whether unblock card makes the card active again*****")
    data_button_1 = {'add':1234}
    data_button_2 = {'add':12345}
    self.client.post('/unblock', data = data_button_1)
    response = self.client.get('/unblock', follow_redirects=True)
    self.assertTrue(b'The Card: 1234 has been unblocked successfully' in response.data)
    self.client.post('/unblock', data = data_button_2)
    response_1 = self.client.get('/unblock', follow_redirects=True)
    self.assertTrue(b'The Card: 12345 has been unblocked successfully' in response_1.data)
    self.assertTrue(b'No cards to unblock' in response_1.data)

if __name__ == "__main__":
    unittest.main()

```

Figure 54 Test Code 3

## 7.9.1 Test Cases

### Admin User

Table 4 Admin Test Case

Sr. No	Test Cases	Result
1)	Verify user can login using the credentials	Pass
2)	Verify admin's dashboard is loaded successfully	Pass
3)	Verify the admin can navigate successfully to block card page	Pass
4)	Verify the admin can block the cards of users seen on the block card list	Pass
5)	Verify the admin can navigate to unblock card page	Pass



6)	Verify the admin can unblock cards of the users as seen on the unblock card list	Pass
----	--	------

## Customer User

Table 5 Customer Test Case

Sr. No	Test Cases	Result
1)	Verify user can login using the credentials	Pass
2)	Verify customer's dashboard is loaded successfully	Pass
3)	Verify the customer can navigate successfully to 'View Statement' page	Pass
4)	Verify the customer can view the statement successfully	Pass
5)	Verify the customer can navigate to Pin change page	Pass
6)	Verify the customer can change the Pin successfully	Pass

## Employee User

Table 6 Employee Test Case

Sr. No	Test Cases	Result
1)	Verify user can login using the credentials	Pass
2)	Verify employee's dashboard is loaded successfully	Pass
3)	Verify the employee can navigate successfully to 'Onboard Customers' page	Pass
4)	Verify the employee can Onboard customers successfully	Pass

## 8 Code Added Value

### 8.1.1 Use of Object Relational Mapping

#### Use of Object Relation Mapper: SQLAlchemy

Consider the situation where we need to write complex queries in any programming language. Database queries are concrete and powerful. Not necessarily everyone will be well versed in formulating queries. On using an object-relational mapper we can communicate with a database using the object-oriented techniques and programming language of our choice. It smoothes database switching of an application in the future if required. An Object Relational Mapper will automatically translate the data stored in the database to objects that can be accessed in applications.

The Credit Card Management System was built on the web application framework FLASK. Flask offers Flask-SQLAlchemy extension. SQLAlchemy is written in python. It was used throughout the project for inserting, querying, and updating into the relational tables via table objects, since classes can be mapped into tables it provides a clean way of developing the object model and database schema. The Database chosen for the system was MYSQL. MYSQL is an open-source relational database. It offers creation, modification, data extraction, and controlled access to the tables. All necessary dependent modules for the project must be imported.

Below are the steps involved in creating and manipulating SQLAlchemy objects along with the code snippets.

1. Firstly import flask\_sqlalchemy from SQLAlchemy.
2. Declared and Initialised MYSQL Database Host details for connection.
3. We explicitly mention configuration for Database in app.config['SQLALCHEMY\_DATABASE\_URI']
4. Database URI is appended with MYSQL\_HOST details.
5. Created an instance of SQLAlchemy(db) which will be used for manipulation

<Below is a screenshot from app.py in the project file >

```
from flask_sqlalchemy import SQLAlchemy
#getting values from yaml file
MYSQL_HOST = dbase['mysql_host']
MYSQL_USER = dbase['mysql_user']
MYSQL_PASSWORD = dbase['mysql_password']
MYSQL_DB = dbase['mysql_db']

#Using SQL Alchemy [ORM]
app.config['SQLALCHEMY_DATABASE_URI'] = 'mysql+pymysql://' + MYSQL_USER + ':' + MYSQL_PASSWORD + '@' + MYSQL_HOST + '/' +
+MYSQL_DB
db=SQLAlchemy(app)
```

6. After connection is established , Model classes are created . Each model is a table representation with the number of columns . Table name Columns and keys could be specified inside the Model classes .Model classes extends from baseclass db.Model ,This class is a declarative base which is used to declare models in flask\_sqlalchemy.
7. Table name , Column attributes along with constraints and their data types could be set inside the model ,

<given is a snippet of model creation and related data manipulations in project>

```
from app import db
class Transaction(db.Model):
    transactionId=db.Column(db.Integer,primary_key=True)
    cardNumber=db.Column(db.String(30))
    transactionType=db.Column(db.String(30))
    transactionSource=db.Column(db.String(30))
    timestamp=db.Column(db.DateTime)
    amount=db.Column(db.Integer)
    isSettled=db.Column(db.Boolean)
```

Figure 55 Transaction Class

This is similar to Create SQL Query

CREATE TABLE Transaction (transactionId Int, cardNumber int,transactionType varchar(30),transactionSource varchar(30),timestamp Date,amount int , isSettled Boolean);The Model class attributes needs to be initialized by class constructor init() method

8. Once we initialise the constructor , we can define class methods in it .As there is a separation between DB layer and business logic . we have defined data setter, data getter, fetch, update method inside DBcontroller and used their instances in Businesscontroller .

The TransactionDBController contains add method that sets value for the Transaction table attributes and adds into Database using db.session.add(Transaction) once data is added commit the insert change via db.session.commit().similarly there is another method that returns the a transaction object .

<given is a snippet from project on invocation of data getter and setter methods >

```
def __init__(self,cardNumber=None,transactionType=None,transactionSource=None,
timestamp=None,amount=None,isSettled=None):
    self.cardNumber = cardNumber
    self.transactionType = transactionType
    self.transactionSource=transactionSource
    self.timestamp=timestamp
    self.amount=amount
    self.isSettled=isSettled
```

Figure 56 Transaction Class Initialisation

From the above snippet querying of class objects are used via SQLAlchemy object db. Interaction with database is done using session object. A session had methods like add, commit, add all, rollback, flush etc. most selection statements generated are formulated by query object. Query Object has a query method which takes the model class as a parameter. Where clause is depicted by filter or filter by options. This above result set will contain all elements from the table because of all() .In case , if we wish to query first element from the table we use .first() .Filtering of tables, adding data value to instances were with the help of SQLAlchemy.

## 9 Recovered Blueprints

### 9.1 State Chart

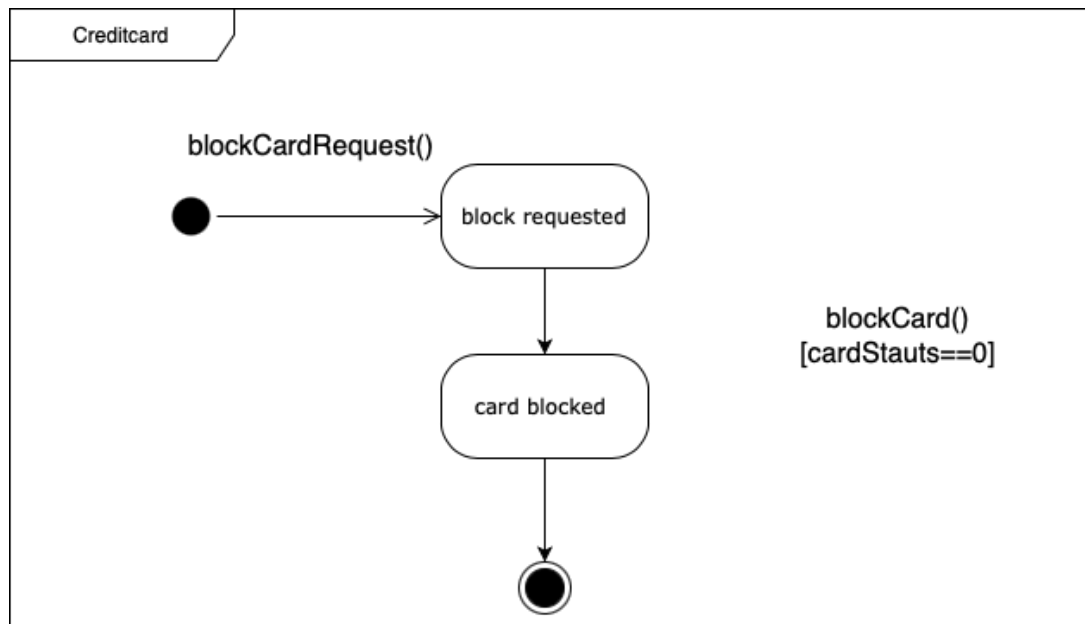


Figure 57. Credit Card state diagram

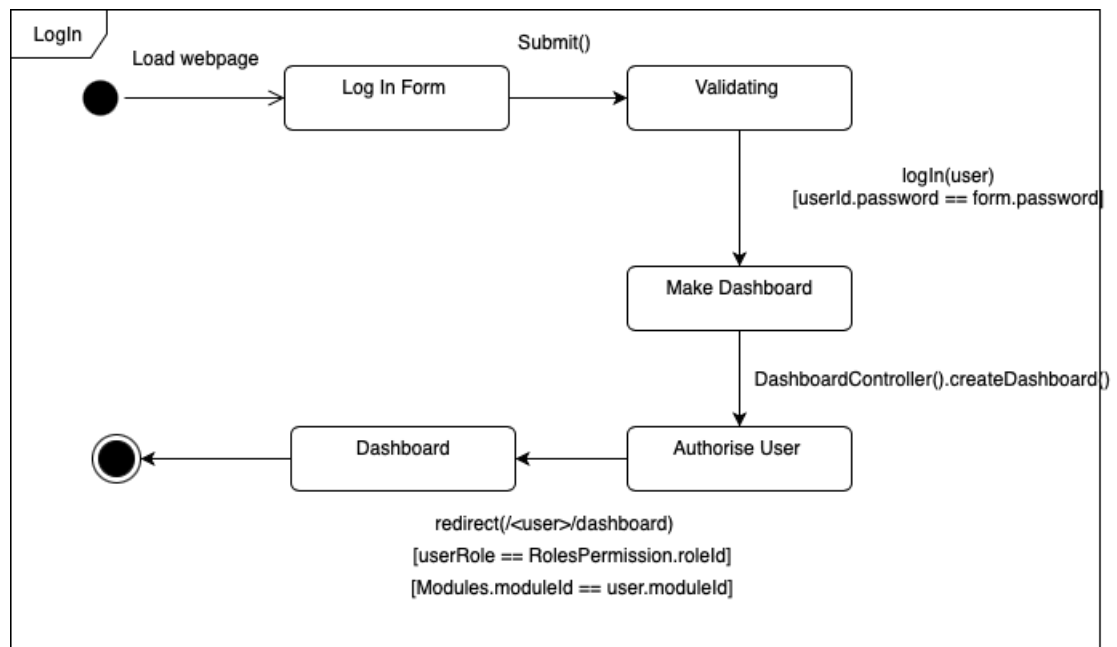


Figure 58. Log In State Chart Diagram

## 9.2 Design-Time Class Diagram

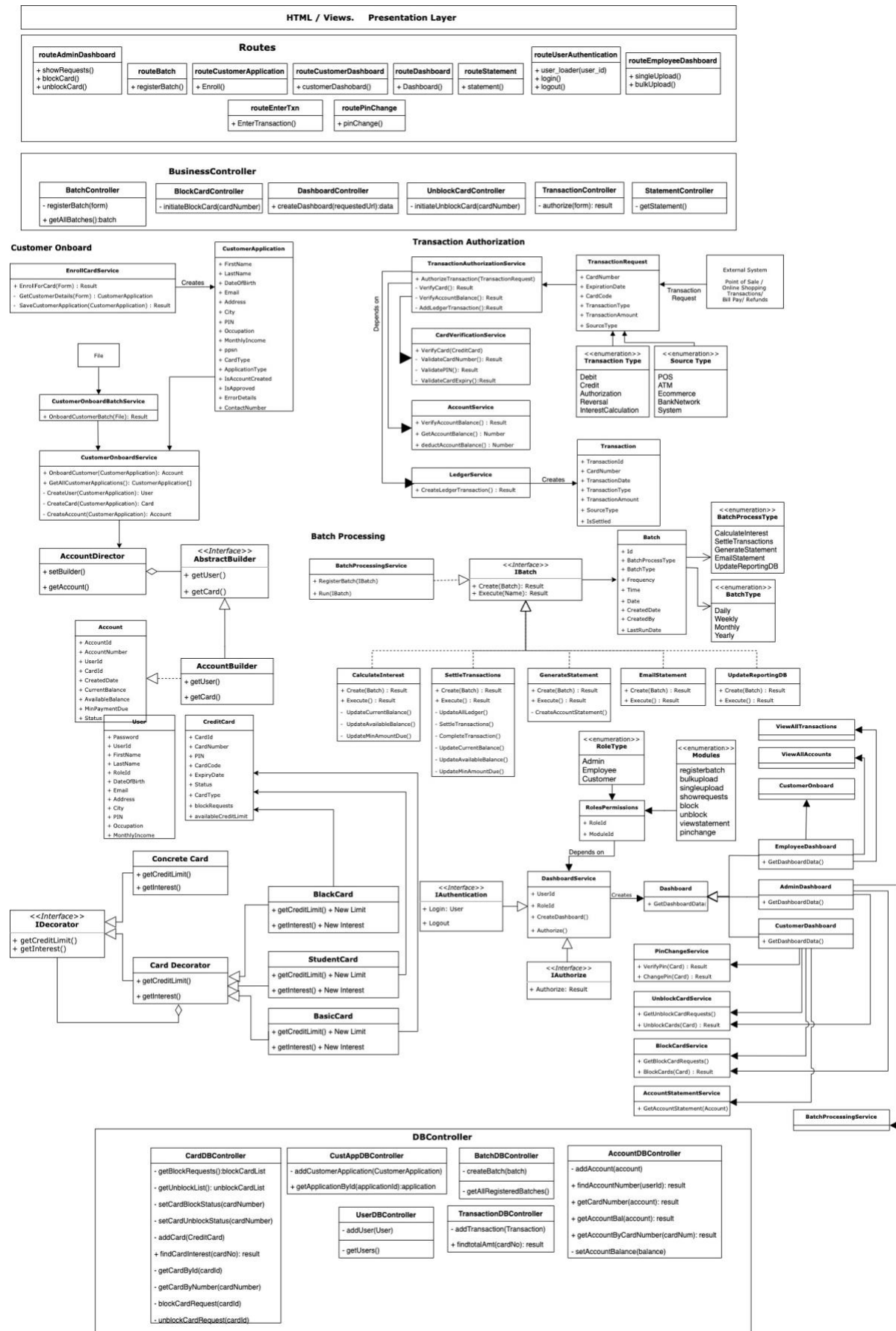


Figure 59 Design Time Class Diagram

## 10 Deployment and Component Architecture

The below UML component diagram represents our system where day to day transactions that happen (Card swipes in machines) is processed through our Business Controller. Business Controller is heart of our system which is the control part of MVC structure, left side of the diagram showing the Web UI which represents views, and the DB Controller handles the models. Database is accessed through the ORM. Authentication and Authorization happens every time a user of the system logs in and every time a transaction request is received.

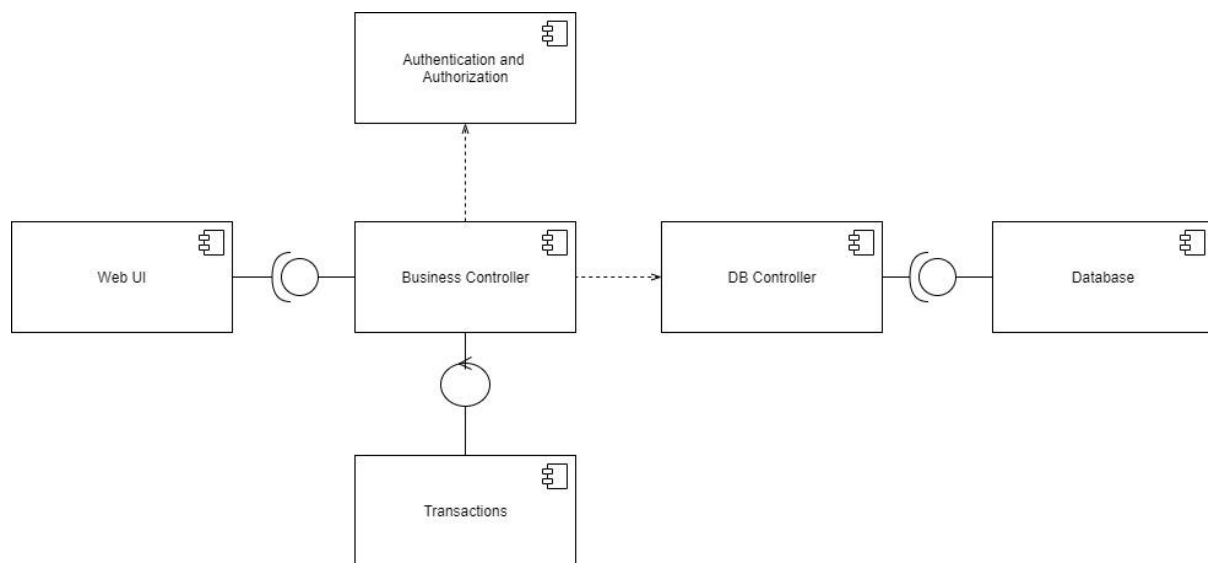


Figure 60 Component Diagram

### Deployment

Google Cloud is just one of many providers for Python flask applications. The below diagram shows various components that would be deployed and also components of Google Cloud that would play an important role in smooth running of the application.

Components : AppEngine – Python Flask, Static Content – Html files, TransactionDB – MySQL, BatchApp – Batch Processing, Cloud deployment manager- continuous code integration, Load balancing – handling multiple requests

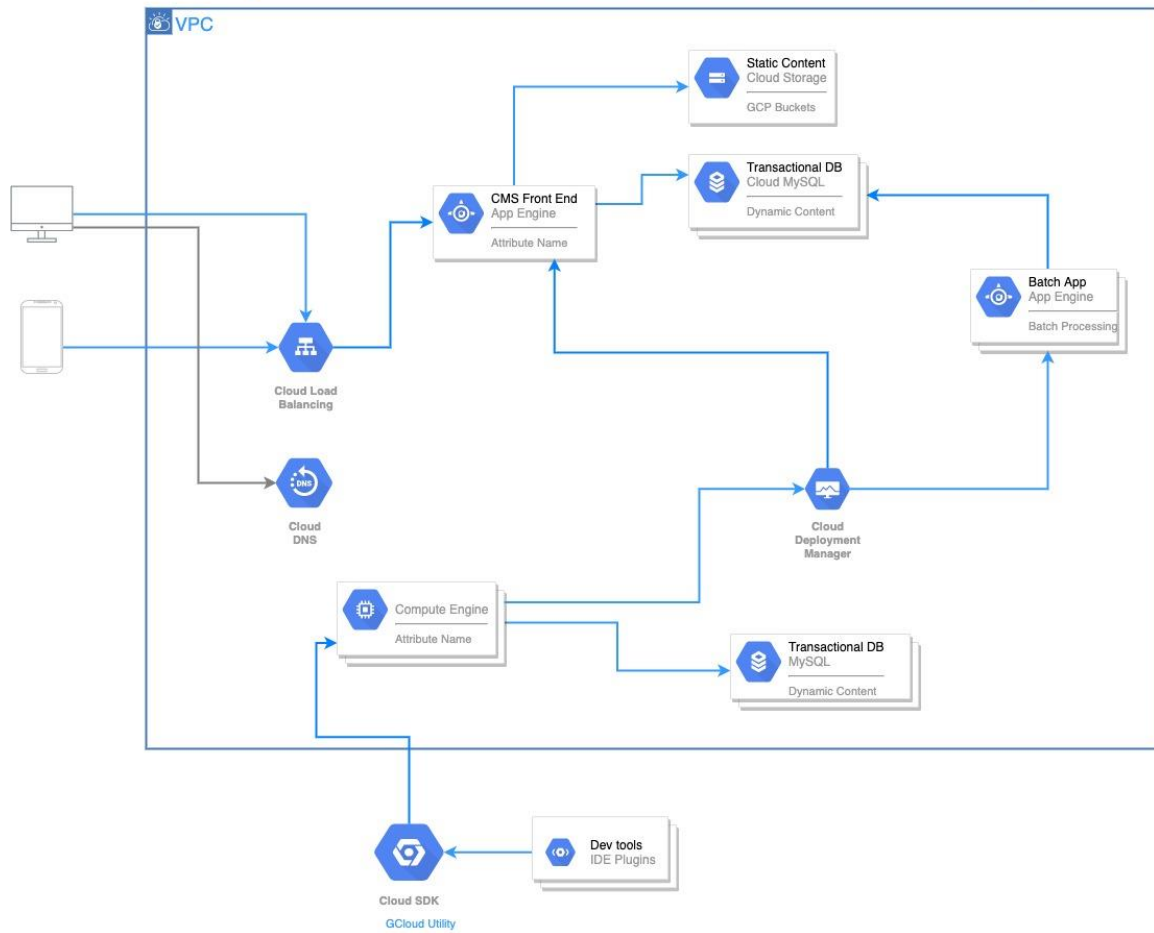


Figure 61 Deployment Diagram

## 11 Critique

### 11.1 State Chart

The recovered state chart diagrams although somewhat similar to the initial communication diagrams are not the same. In the original communication diagrams it is clear that not enough thought was not given to the method by which the correct dashboard for a user would be rendered.

### 11.2 Design Patterns

Decorator pattern is used for different types of cards. As the system would be further developed, the type of cards in the system could increase and also the varying attributes. This would cause a lot of files to change in order add the decorating attributes. This goes against the object oriented open closed principle of class should be open for extension and closed for modification. Instead of Decorator pattern for Card, it could be a Card Factory, where different types of cards could be generated. Using a factory pattern, would change just one file i.e. the factory class and new classes could be added for new card types.

### 11.3 Architecture

Although we have used MVC architecture pattern here. The application could be converted into a REST API architecture Considering the application would communicate with external systems. With Python flask, the transformation is easier as the methods with routes would still act as controllers for the RESTAPI. One of them is the transaction authorization requests.

### 11.4 Python and Design Patterns

Implementation of design patterns with Python was a challenge considering, python does not support or rather does not require Interfaces. Python uses the concept of abstract classes. Sometimes the implementation does not mimic the exact design of a design pattern. However for learning could still be implemented using python.

### 11.5 Project Critique

Many challenges were encountered during the project execution. After doing the use case and the key use case scenarios the development had to be started. This also involved a lot of learning as coding experience and background was uniform in the team. Python was a new language for everyone in the team. The analysis class diagram took a lot of iterations to be evolved. As the development had already started. The analysis class diagram was a detailed one with actual design time classes that were implemented. However, the diagrams played an essential part that gave an overview of the entire system. Every team member contributed fairly throughout the project and brought in their expertise.



## 12 Bibliography

- [1] I. Ghosh, "The \$88 Trillion World Economy in One Chart," September 2020. [Online]. Available: <https://www.visualcapitalist.com/the-88-trillion-world-economy-in-one-chart/>. [Accessed December 2020].
- [2] "Contributions of Card Payment Systems to Economy," BKM, [Online]. Available: <https://bkm.com.tr/en/useful-information/card-awareness/benefits-of-card-payment-systems-to-consumers/>. [Accessed December 2020].
- [3] "Flask," [Online]. Available: <https://flask.palletsprojects.com/en/1.1.x/>. [Accessed December 2020].
- [4] "Flask-SQLAlchemy," [Online]. Available: <https://flask-sqlalchemy.palletsprojects.com/en/2.x/#>. [Accessed December 2020].
- [5] N/A, "Flask-Login," [Online]. Available: <https://flask-login.readthedocs.io/en/latest/>. [Accessed December 2020].
- [6] A. Shalloway, Design Patterns Explained: A New Perspective on Object-Oriented Design, Second Edition, Addison-Wesley Professional, 2004.
- [7] E. Freeman, Head First Design Patterns, 2nd Edition, O'Reilly Meida, Inc, 2020.
- [8] M. Seemann, Dependency Injection Principles, Practices and Patterns, Manning Publications, 2019.
- [9] "Draw.io," [Online]. Available: <https://app.diagrams.net>.
- [10] G. 17, "CS5721\_Software\_Design," [Online]. Available: [https://github.com/riyajoe/CS5721\\_Software\\_Design](https://github.com/riyajoe/CS5721_Software_Design).
- [11] "Python Runtime 3," [Online]. Available: <https://cloud.google.com/appengine/docs/standard/python3/runtime> . [Accessed December 2020].

## 13 Appendix

### 13.1 Table of Figures

Figure 1 System Use Case Diagram.....	8
Figure 2 System Use Case Diagram without Extends and Includes .....	9
Figure 3 Admin Use Case Diagram.....	10
Figure 4 Machine (External System) Use Case Diagram .....	11
Figure 5 Employee Use Case Diagram .....	11
Figure 6 GUI Prototype Log In.....	17
Figure 7 GUI Prototype: View Statement .....	17
Figure 8 GUI Prototype: Onboard Customer .....	18
Figure 9 System Package Diagram .....	19
Figure 10 System Architecture diagram .....	21
Figure 11 Analysis Class Diagram 1 .....	24
Figure 12 Analysis Class Diagram 2 .....	25
Figure 13 Analysis Class Diagram 3 .....	26
Figure 14 Analysis Class Diagram 4.....	26
Figure 15 Log In Communication Diagram.....	27
Figure 16 Employee Approve Card Communication Diagram .....	27
Figure 17 Customer Application Communication Diagram .....	28
Figure 18 View Statement Communication Diagram .....	28
Figure 19 Employee Approve Card Request .....	29
Figure 20 Transaction Authorisation Request Sequence.....	30
Figure 21 View Statement Sequence Diagram .....	31
Figure 22 E-R Diagram.....	32
Figure 23 Façade Pattern .....	33
Figure 24 Transaction Authorisation Service Class .....	34
Figure 25 Ledger Service Class .....	34
Figure 26 Account Service Class.....	34
Figure 27 Card Verification Service Class.....	35
Figure 28 Builder Pattern.....	35
Figure 29 Customer Onboard Service Class .....	36
Figure 30 Abstract Builder Class .....	36
Figure 31 Account Builder Class.....	37
Figure 32 Account Director Class .....	37
Figure 33 Factory Patter Diagram .....	38
Figure 34 Dashboard Interface .....	38
Figure 35 Employee Dashboard Class .....	38
Figure 36 Customer Dashboard Class .....	39
Figure 37 Dashboard Factory Class.....	39
Figure 38 Decorator Pattern Diagram.....	40
Figure 39 Card Decorator.....	40
Figure 40 Card Class .....	40
Figure 41 Card Decorator Interface .....	41
Figure 42 Student Card Class .....	41
Figure 43 Basic Card Class.....	41
Figure 44 Black Card Class.....	41
Figure 45 Batch Interface.....	42
Figure 46 Batch Processing Service Class.....	42
Figure 47 Calculate Interest Class .....	42
Figure 48 Register Batch Class .....	43

Figure 49 Implemented MVC Diagram .....	43
Figure 50 Branching and Merging in GitHub shown using Gitk .....	45
Figure 51 Range of Commits During Implementation .....	45
Figure 52. Test Code 1.....	46
Figure 53 Test Code 2.....	47
Figure 54 Test Code 3.....	47
Figure 55 Transaction Class.....	49
Figure 56 Transaction Class Initialisation.....	50
Figure 57. Credit Card state diagram .....	50
Figure 58. Log In State Chart Diagram .....	51
Figure 59 Design Time Class Diagram .....	52
Figure 60 Component Diagram .....	53
Figure 61 Deployment Diagram .....	54

### 13.2 Table of Tables

Table 1 View Customer Transaction Use Case Description .....	12
Table 2 Apply for Card Use Case Description .....	13
Table 3 Transaction Authorisation Use Case Description.....	14
Table 4 Admin Test Case .....	47
Table 5 Customer Test Case.....	48
Table 6 Employee Test Case.....	48

### 13.3 Source Code Information

#### Models

UserAuthorization.py	Yukti
Transactions.py	Riya
UnblockCard.py	Dhavan
UserAuthentication.py	Praveen
AccountService.py	Yukti
CardVerificationService.py	Yukti
LedgerService.py	Yukti
TransactionAuthorizationService.py	Yukti
TransactionRequest.py	Yukti
Statement.py	Ciara, Riya
CustomerOnboardBatchService.py	Yukti, Riya
CustomerOnboardService.py	Yukti, Riya
PinChange.py	Praveen
Result.py	Yukti
Role.py	Yukti
RolesPermission.py	Yukti
Modules.py	Yukti
BasicCard.py	Yukti
BlackCard.py	Yukti
CardDecorator.py	Yukti

CardDecoratorService.py	Yukti
ConcreteCard.py	Yukti
IDecorator.py	Yukti
StudentCard.py	Yukti
EmployeeDashboard.py	Yukti, Riya
DashboardFactory.py	Yukti
BatchCreateHelper.py	Yukti
BatchProcess.py	Yukti
BatchProcessingService.py	Yukti
CalculateInterest.py	Yukti
EmailStatement.py	Yukti
GenerateStatement.py	Yukti
SettleTransactions.py	Yukti
UpdateReportingDB.py	Yukti
BlockCard.py	Dhavan
Card.py	Dhavan, Riya
CustomerApplication.py	Praveen, Riya
CustomerDashboard.py	Riya, Ciara
AdminDashboard.py	Dhavan
AbstractBuilder.py	Yukti
AccountBuilder.py	Yukti, Riya
AccountDirector.py	Yukti, Riya
Account.py	Yukti, Riya

#### Interfaces

IActionOnCard.py	Dhavan
IAuthorization.py	Yukti
IBatch.py	Yukti
IDashboard.py	Yukti

#### Helper

randGenerator.py	Riya
Enums.py	Yukti

#### DefineRoutes

routeUserAuthentication.py	Praveen
routeAdminDashboard.py	Dhavan, Riya
routeBatch.py	Yukti
routeCustomerApplication.py	Praveen, Riya
routeCustomerDashboard.py	Ciara, Riya
routeDashboard.py	Yukti, Dhavan, Riya, Ciara, Praveen
routeEmployeeDashboard.py	Riya, Yukti

routeEnterTxn.py	Riya, Yukti
routePinChange.py	Praveen
routeStatement.py	Ciara, Riya
routeUser.py	Praveen

AccountDBController.py	Riya, Ciara
BatchDBController.py	Yukti
CardDBController.py	Riya, Ciara, Dhavan
CustAppDBController.py	Yukti
TransactionDBController.py	Riya
UserDBController.py	Riya

BatchController.py	Yukti
BlockCardController.py	Yukti, Dhavan
DashboardController.py	Yukti
EnrollController.py	Praveen, Yukti
Setup.py	Riya
StatementController.py	Ciara, Riya
TransactionController.py	Riya, Yukti
UnblockCardController.py	Dhavan

#### Tests.py

Test_AdminDashboard.py	Dhavan
Test_BlockCardPage.py	Dhavan
Test_UnblockCardPage.py	Dhavan

#### Main Files

app.py	Dhavan,Praveen,Riya,Yukti,Ciara
forms.py	Ciara, Dhavan,Praveen,Riya,Yukti