

# *Bank Management Application*

Ajitesh Sarma Godavarthi  
(SUID: 574144082)

Tabish Khan  
(SUID: 309670657)

Midhuneshwar Kandasamy Kanivalavan  
(SUID: 351564889)

## Introduction:

The role of bank management is multi-faceted and involves a wide range of responsibilities to ensure that the bank is running efficiently and effectively. Our basic idea behind the Bank Management Help Application is to perform the basic applications of a banking system such as:

1. Creation of account
2. Payment
3. Money Transactions
4. Viewing balance
5. Authorization and etc.



WELLS  
FARGO



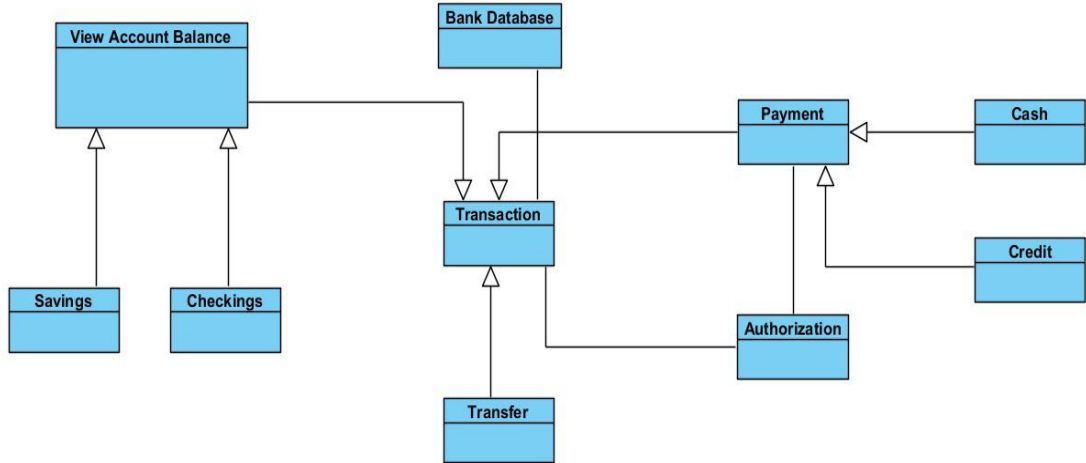
Withdrawing Money



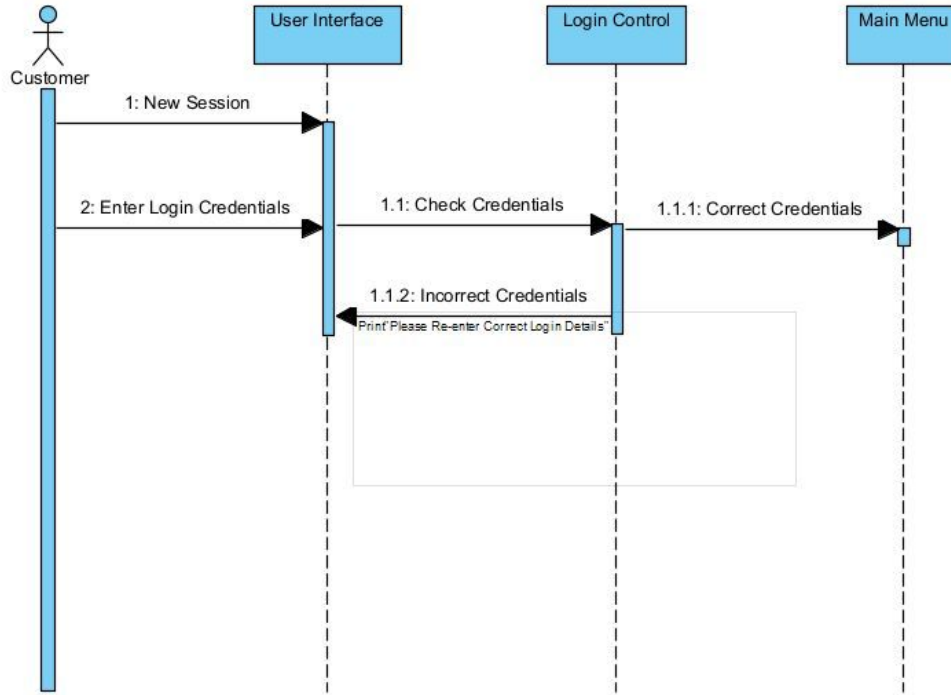
Bank Money Transfers

## Class Diagram:

As you can see from the class diagram below are user interface will consist of function where you can view the account balance depending on the type of account you have either savings or checking accounts. This then takes us to the transaction section from where you can either transfer money from one account to another or get payment in the form of cash or credit but before this is done the system needs to authorize the transaction based on the bank database



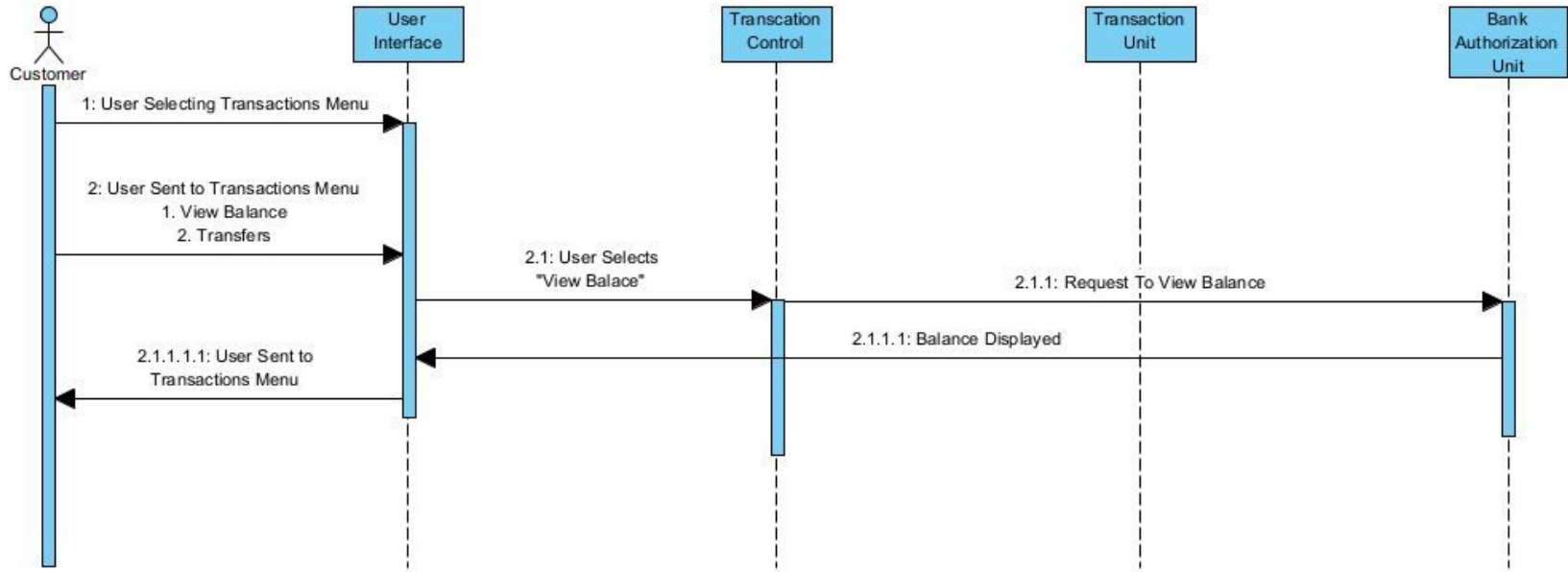
## Sequence Diagrams:



Sequence for Login

This diagram is the Sequence of how Login takes place. The Flow is where a new session is created for the user and the user enter their credentials which is they enter their account number and password. And the server verifies the credentials and the user is directed to the Main Menu if the credentials are correct. If the credentials are not correct it's rerouted to Login page.

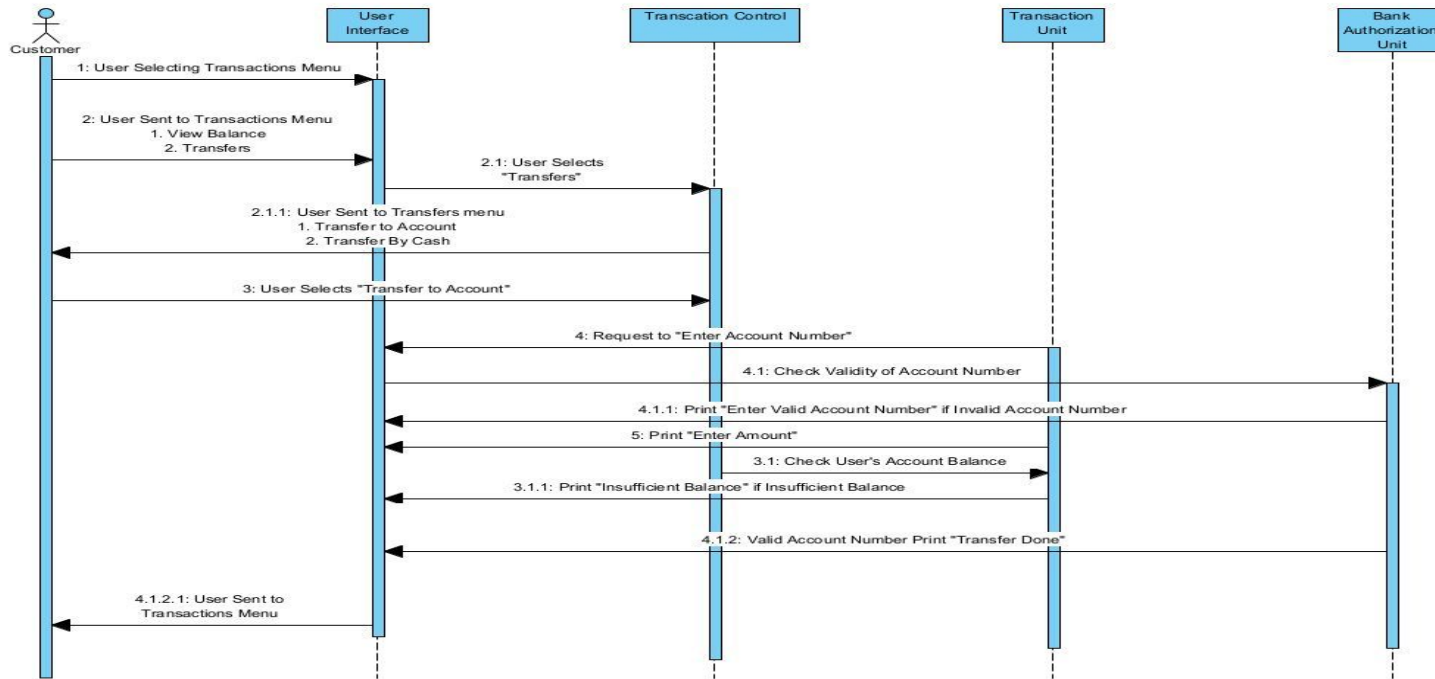




**Sequence for Transactions-View Balance**

This diagram is the sequence of Transaction menu and the sub part of View Balance. User Selects the Transactions menu and user is directed to Transactions menu which has two options 1. View Balance and 2. Transfers. When the user selects “View Balance”. Balance is retrieved from the server and displayed in the user screen.



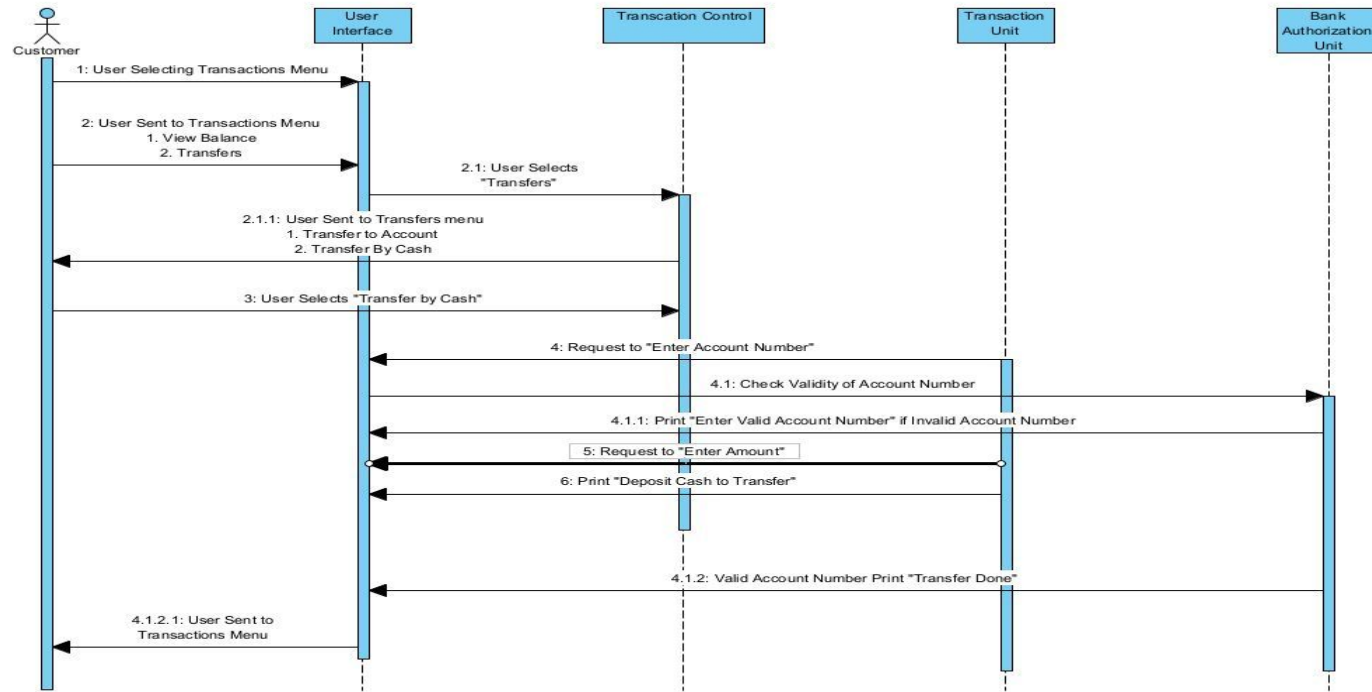


## Sequence for Transaction-Transfer by Account Number



This diagram is the sequence of Transaction menu and the sub part of Transfers. The Transfer option has two options which are 1.Transfer to Account and 2. Transfer by cash. The diagram displays the flow of Transfer to Account. User selects Transfer to Account. Where the user enters transferee's account number and server verifies the account number if valid user enters the amount to be transferred if not user is prompted to enter valid account number and the server checks for the balance and the transfer is done if the balance is sufficient if not user is prompted with "Insufficient Balance".





## Sequence for Transactions-Transfer by Cash

This diagram is the sequence of Transaction menu and the sub part of Transfers. The Transfer option has two options which are 1.Transfer to Account and 2. Transfer by cash. The diagram displays the flow of Transfer to Account. User selects Transfer to Account. Where the user enters transferee's account number and server verifies the account number if valid user enter the amount to be transferred and deposit the cash to be transferred if not user is prompted to enter valid account number. And the transfer is done.



## Code Snippets

[illegible]

## Main UI Interface

```

void enqueue()
{
    withdrawDepositNode* temp = new withdrawDepositNode();

    int opt = 0;
    string flag = "";

    cout << "\t\t\t\t\t Kindly choose from the following two options: 1) Withdraw Cash (cheque) or 2) Deposit Cash -> ";
    cin >> opt;

    if (opt == 1)
    {
        temp->withdrawCash = true;
        temp->depositCash = false;
        flag = "Withdraw Cash";
    }
    else if (opt == 2)
    {
        temp->depositCash = true;
        temp->withdrawCash = false;
        flag = "Deposit Cash";
    }
    else
    {
        cout << "\t\t\t\t\t Wrong option entered, press any key to return to menu." << endl;
        _getch();
        return;
    }

    char confirmation;

    cout << "\t\t\t\t\t Kindly write down the 10 digit Account Number -> ";
    cin >> temp->accountNumber;
    cout << "\t\t\t\t\t Kindly write down the cash amount -> ";
    cin >> temp->cashAmount;
    cout << "\t\t\t\t\t-----" << endl;
    cout << "\t\t\t\t\tAccount#: " << temp->accountNumber << endl;
    cout << "\t\t\t\t\tCash: " << temp->cashAmount << endl;
    cout << "\t\t\t\t\tTransaction Type: " << flag << endl;
    cout << "\t\t\t\t\t-----" << endl;
    cout << "\t\t\t\t\t Press 'Y' to confirm, or 'N' to exit" << endl;
    cout << "\t\t\t\t\t ", cin >> confirmation;
    temp->next = NULL;

    if (confirmation == 'y' || confirmation == 'Y')
    {
        if (isEmpty())
        {
            temp->prev = NULL;
            front = rear = temp;
            cout << "\t\t\t\t\t Transaction Completed." << endl;
        }
        else
        {
            temp->prev = rear;
            rear->next = temp;
            rear = temp;
            cout << "\t\t\t\t\t Transaction Completed." << endl;
        }
    }
    else if (confirmation == 'n' || confirmation == 'N')
    {
        temp = NULL;
        cout << "\t\t\t\t\t Transaction not Completed." << endl;
    }
    else
    {
        cout << "\t\t\t\t\t ERROR ERROR ERROR!!! Wrong input entered, transaction failed!" << endl;
        temp = NULL;
    }

    cout << "\t\t\t\t\t Press any key to return to Bank Menu" << endl;
    _getch();
}

```

## Enqueue

This code defines the function enqueue(), which adds a new node to the end of a linked list. The linked list represents a bank queue for customers who want to deposit or withdraw cash.

The function starts by creating a new withdrawDepositNode object, which contains information about the transaction, such as the account number, cash amount, and transaction type.

```

void dequeue()
{
    withdrawDepositNode* temp = front;

    if (isEmpty())
    {
        cout << "\t\t\t\t Withdraw/Deposit Queue is empty, nothing left to process" << endl;
        cout << "\t\t\t\t Press any key to return to Bank Menu" << endl;
        _getch();
        return;
    }
    else if (front == rear)
    {
        cout << "\t\t\t\t-----" << endl;
        cout << "\t\t\t\t " << temp->accountNumber << endl;
        cout << "\t\t\t\t " << temp->cashAmount << endl;
        cout << "\t\t\t\t-----" << endl;
        cout << "\t\t\t\t Transaction has been done." << endl;
        front = rear = NULL;
    }
    else
    {
        front = front->next;
        cout << "\t\t\t\t-----" << endl;
        cout << "\t\t\t\t " << temp->accountNumber << endl;
        cout << "\t\t\t\t " << temp->cashAmount << endl;
        cout << "\t\t\t\t-----" << endl;
        cout << "\t\t\t\t Transaction has been done." << endl;
    }
    delete temp;
    cout << "\t\t\t\t Press any key to return to Bank Menu" << endl;
    _getch();
}

```

## Dequeue

This code implements a queue data structure for a banking system that can handle withdrawal and deposit transactions. The enqueue() function is used to add a transaction to the back of the queue and the dequeue() function is used to remove the transaction from the front of the queue. The displayWithdrawDepositQueue() function is used to display all transactions currently in the queue.

## Account Processing

```
}
void processAccounts()
{
    newAccountNode* temp = head;

    if (head == NULL)
    {
        cout << "\t\t\t\t\t Account Database is empty" << endl;
        cout << "\t\t\t\t\t Press any key to return to Bank Menu" << endl;
        _getch();
        return;
    }

    int count = 1;
    int opt;

    while (temp != NULL)
    {
        if (temp->accountStatus == "In Process")
        {
            cout << "\t\t\t\t\t----- # << count++ << " -----" << endl;
            cout << "\t\t\t\t\t-----" << endl;
            cout << "\t\t\t\t\tFirst Name: " << temp->firstName << endl;
            cout << "\t\t\t\t\tLast Name: " << temp->lastName << endl;
            cout << "\t\t\t\t\tCNIC: " << temp->CNIC << endl;
            cout << "\t\t\t\t\t---Address---" << endl;
            temp->homeAddress.displayAddress();
            cout << "\t\t\t\t\t-----" << endl;
            cout << "\t\t\t\t\tUsername: " << temp->username << endl;
            cout << "\t\t\t\t\tPassword: " << temp->password << endl;
            cout << "\t\t\t\t\t----- # << count << " -----" << endl << endl;
            cout << "\t\t\t\t\t Do you want to process this account or not? Yes(1) No(0)" << endl;
            cin >> opt;

            if (opt == 1)
            {
                temp->accountStatus = "Completed";
                temp->accountNumber = generateAccountNumber();
            }
            else if (opt == 0)
            {
                temp->accountStatus = "Declined";
            }
            else
            {
                cout << "\t\t\t\t\t Wrong option entered, account not completed nor declined." << endl;
            }
        }
        temp = temp->next;
    }
    cout << "\t\t\t\t\t Press any key to return to Bank Menu" << endl;
    _getch();
}
```

This code implements a queue data structure for a banking system that can handle withdrawal and deposit transactions. The enqueue() function is used to add a transaction to the back of the queue and the dequeue() function is used to remove the transaction from the front of the queue. The displayWithdrawDepositQueue() function is used to display all transactions currently in the queue.

## Database

This code seems to be a part of a larger program for managing bank accounts. The class `bankAccountsDatabase` represents a database of bank accounts and has a `newAccountNode` as a nested class that represents a node of the linked list that will hold the bank accounts.

After initializing the node, the function prints out the details of the new account and prompts the user to confirm the details. If the user confirms, the function adds the new node to the linked list. If the linked list is empty, the new node becomes the head of the list.

```
#include <iostream>
using namespace std;

class bankAccountsDatabase
{
public:
    newAccountNode* head, * tail;

public:
    bankAccountsDatabase()
    {
        head = tail = NULL;
    }
    void insertNewAccount()
    {
        newAccountNode* temp = new newAccountNode();

        char confirmation;
        int debitCardOpt = 0;
        int chequeBookOpt = 0;
        time_t t = time(NULL);
        tm* tPtr = localtime(&t);

        cout << "\n\t\t\t\t\t Kindly write down your First Name -> ";
        cin >> temp->firstName;
        cout << "\n\t\t\t\t\t Kindly write down your Last Name -> ";
        cin >> temp->lastName;
        cout << "\n\t\t\t\t\t Kindly write down your Email Address -> ";
        cin >> temp->emailAddress;
        cout << "\n\t\t\t\t\t Kindly write down your Home Address -> " << endl;
        temp->homeAddress.setAddress();
        cout << "\n\t\t\t\t\t Kindly write down your Phone Number -> ";
        cin >> temp->phoneNumber;
        cout << "\n\t\t\t\t\t Kindly write down your CNIC -> ";
        cin >> temp->CNIC;
        cout << "\n\t\t\t\t\t Do you want to get a Debit Card: write Yes(1) or No(0)-> ";
        cin >> debitCardOpt;

        if (debitCardOpt == 1)
        {
            temp->debitCard = true;
            cout << "\n\t\t\t\t\t Kindly write down your Debit Card PIN (4 Digits) -> ";
            cin >> temp->debitCardPin;
        }
        else if (debitCardOpt == 0)
        {
            temp->debitCard = false;
            temp->debitCardPin = 0;
            cout << "\n\t\t\t\t\t Debit card is set as No(0)" << endl;
        }
        else
        {
            temp->debitCard = false;
            temp->debitCardPin = 0;
            cout << "\n\t\t\t\t\t Invalid Option entered, debit card is set as No(0)" << endl;
        }

        cout << "\n\t\t\t\t\t Do you want to get a ChequeBook: write Yes(1) or No(0)-> ";
        cin >> chequeBookOpt;
        if (chequeBookOpt == 1)
        {
            temp->chequeBook = true;
        }
        else if (chequeBookOpt == 0)
        {
            temp->chequeBook = false;
        }
        else
        {
            cout << "\n\t\t\t\t\t Invalid Option entered, cheque book is set as No(0)" << endl;
        }
    }
};
```



## OOPS Concepts Used in the Code:

### Encapsulation -

billPaymentQueue, payBillNode, withdrawDepositQueue, newAccountNode, BankQueue

The code uses classes to encapsulate related functions and data together, such as the BankQueue class and the Address class. The code defines two classes, billPaymentQueue and withdrawDepositQueue, which encapsulate the properties and behavior of a queue for bill payments and withdraw/deposit transactions, respectively.

```
12 struct payBillNode
13 {
14     string billType;
15     long billInvoiceNumber;
16     long double billAmount;
17
18
19
20     payBillNode* next;
21     payBillNode* prev;
22 };
23
```

```
24 class billPaymentQueue
25 {
26 private:
27     payBillNode* front;
28     payBillNode* rear;
29 public:
30     billPaymentQueue()
31     {
32         front = rear = NULL;
33     }
34     bool isEmpty()
35     {
36         if (rear == NULL && front == NULL)
37         {
38             return true;
39         }
40         else
41         {
42             return false;
43         }
44     }
45 }
```

```
180 class withdrawDepositQueue
181 {
182 private:
183     withdrawDepositNode* front;
184     withdrawDepositNode* rear;
185 public:
186     withdrawDepositQueue()
187     {
188         front = rear = NULL;
189     }
190     bool isEmpty()
191     {
192         if (rear == NULL && front == NULL)
193         {
194             return true;
195         }
196         else
197         {
198             return false;
199         }
200     }
201 }
```

# OOPS Concepts Used in the Code

**Inheritance** : billPaymentQueue and bankAccountsDatabase are in a parent and child relationship, where billPaymentQueue inherits functions and data from bankAccountsDatabase. Its a single inheritance. In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.

```
class billPaymentQueue : public bankAccountsDatabase
{
private:
    payBillNode* front;
    payBillNode* rear;
public:
    billPaymentQueue()
    {
        front = rear = NULL;
    }
    bool isEmpty()
    {
        if (rear == NULL && front == NULL)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

## OOPS Concepts Used in the Code

**Polymorphism** - isEmpty()(uses function overloading)

The code does not explicitly use polymorphism, but it does use function overloading, which is a form of polymorphism. For example, the isEmpty() function is overloaded in both the billPaymentQueue and withdrawDepositQueue classes to provide different behavior based on the queue type.

```
34 bool isEmpty()  
35 {  
36     if (rear == NULL && front == NULL)  
37     {  
38         return true;  
39     }  
40     else  
41     {  
42         return false;  
43     }  
44 }
```

```
if (isEmpty())  
{  
    cout << "\t\t\t\t\t Bill Payment Queue is empty, nothing left to process" << endl;  
    cout << "\t\t\t\t\t Press any key to return to Bank Menu" << endl;  
    _getch();  
    return;  
}
```

# OOPS Concepts Used in the Code

**Abstraction** - billPaymentQueue, withdrawDepositQueue, newAccountNode, BankMenu, enqueue(), dequeue()

The code creates instances of the billPaymentQueue and withdrawDepositQueue classes using the new keyword to allocate memory dynamically. The instances are then manipulated using member functions, such as enqueue() and dequeue(), to perform operations on the queues.

```
24 class billPaymentQueue
25 {
26 private:
27     payBillNode* front;
28     payBillNode* rear;
29 public:
30     billPaymentQueue()
31     {
32         front = rear = NULL;
33     }
34     bool isEmpty()
35     {
36         if (rear == NULL && front == NULL)
37         {
38             return true;
39         }
40         else
41         {
42             return false;
43         }
44     }
45 }
```

```
180 class withdrawDepositQueue
181 {
182 private:
183     withdrawDepositNode* front;
184     withdrawDepositNode* rear;
185 public:
186     withdrawDepositQueue()
187     {
188         front = rear = NULL;
189     }
190     bool isEmpty()
191     {
192         if (rear == NULL && front == NULL)
193         {
194             return true;
195         }
196         else
197         {
198             return false;
199         }
200     }
201 }
```

```
386 struct newAccountNode
387 {
388     string firstName;
389     string lastName;
390     string accountStatus;
391     long long phoneNumber;
392     long long CNIC;
393     long double cash;
394     string emailAddress;
395     address homeAddress;
396     bool debitCard;
397     bool chequeBook;
398     int debitCardPin;
399     string username;
400     string password;
401     long accountNumber;
402     newAccountNode* next;
403     newAccountNode* prev;
404 };
405
406 class bankAccountsDatabase
407 {
408 public:
409     newAccountNode* head, * tail;
410
411 public:
412     bankAccountsDatabase()
413     {
414         head = tail = NULL;
415     }
416     void insertNewAccount()
417     {
418         newAccountNode* temp = new newAccountNode();
419     }
420 }
```

# JIRA Snippets

The screenshot displays the JIRA interface for the 'Bank Management Application' project. The left sidebar shows navigation options: PLANNING (Roadmap, Backlog, Board) and DEVELOPMENT (Code, Project pages, Add shortcut, Project settings). The main area is titled 'Backlog' and shows three sprints:

- BMA Sprint 1**: Add dates (1 issue). Description: Develop the login and registration pages for users to access the system. Create a dashboard for users to view their account information, including b...
  - BMA-18 Sprint 1: This is the first step of the project
- BMA Sprint 2**: Add dates (1 issue). Description: Create a system for users to create and manage their bank accounts, including savings, checking, and credit accounts. Implement a feature for users ...
  - BMA-20 Sprint 2: Account Management
- BMA Sprint 3**: Add dates (1 issue). Description: Develop a transaction processing system for deposit and withdrawal transactions. Implement a feature for users to transfer funds to other bank acco...
  - BMA-24 Sprint 3: Transaction Processing

On the right, the 'SRS Documentation Edits' task detail is shown. It includes a 'Description' section with the text: 'Edit the SRS document to fit the flow of the example SRS documents for submission'. The 'Details' section shows the assignee as 'ajitesh godavarthi' and labels as 'None'. A comment box is also visible with the text 'Add a comment...'. A 'Pro tip' at the bottom suggests pressing 'M' to comment.

Sprint 1: First step of project

Sprint 2: Account Management

Sprint 3: Transaction Processing

Jira Software Your work ▾ Projects ▾ Filters ▾ Dashboards ▾ Teams ▾ Apps ▾ Create

Search

Bank Management Ap...  
Software project

PLANNING

Roadmap

Backlog

Board

DEVELOPMENT

Code

Project pages

Add shortcut

Project settings

You're in a team-managed project  
Learn more

Projects / Bank Management Application

## Backlog

AG + Epic ▾

▼ BMA Sprint 4 Add dates (1 issue) 0 0 0 Start sprint ...

Develop a reporting system for generating customer reports, such as transaction history and account balances. Add a feature for users to view spen...

BMA-27 Sprint 4: Reporting and Analytics TO DO ▾

+ Create issue

▼ BMA Sprint 5 Add dates (1 issue) 0 0 0 Start sprint ...

Test the entire system to ensure that it meets the user requirements. Perform integration testing with external systems, such as payment gateways a...

BMA-30 Sprint 5: Integration and Testing TO DO ▾

+ Create issue

▼ Backlog (5 issues) 0 0 0 Create sprint

BMA-1 As owners of this project, we want to develop a banking application that h... TO DO ▾

Insights

Add epic / BMA-11

## SRS Documentation Edits

Done ▾ Actions ▾

Description

Edit the SRS document to fit the flow of the example SRS documents for submission

Details

Assignee AG ajitesh godavarthi

Labels None

AG Add a comment...

Pro tip: press M to comment

Sprint 4: Reporting and Analytics

Sprint 5: Integration and Testing

Jira Software

Your work

Projects

Filters

Dashboards

Teams

Apps

Create

Q Search

?

AG

Bank Management Ap...

Software project

PLANNING

Roadmap

Backlog

Board

DEVELOPMENT

Code

Project pages

Add shortcut

Project settings

You're in a team-managed project

Learn more

Projects / Bank Management Application

Backlog

AG

Epic

test the entire system to ensure that it meets the user requirements. Perform integration testing with external systems, such as payment gateways a...

BMA-30 Sprint 5: Integration and Testing

TO DO

+ Create issue

Backlog (5 issues)

000Create sprint

BMA-1 As owners of this project, we want to develop a banking application that he...

TO DO

BMA-2 Project Proposal

DONE

BMA-6 Gas Pump Exercise

DONE

BMA-10 SRS Documentation

IN PROGRESS

BMA-14 Sprints Discussions

IN PROGRESS

+ Create issue

Add epic

BMA-11

1

Insights

SRS Documentation Edits

Done

Actions

Description

Edit the SRS document to fit the flow of the example SRS documents for submission

Details

Assignee

AGajitesh godavarthi

Labels

None

AG

Add a comment...

Pro tip: press **h** to comment

Other Stories

Jira Software Your work ▾ Projects ▾ **Filters** ▾ Dashboards ▾ Teams ▾ Apps ▾ Create

Q Search 🔔 ? ⚙️ AG

**Filters**

Search issues

OTHER

My open issues

Reported by me

All issues

Open issues

Done issues

Viewed recently

Created recently

Resolved recently

Updated recently

View all filters

**Search** Save as

Bank Managem... ▾ Type: All ▾ Status: All ▾ Assignee: All ▾ + More

Contains text Search Switch to JQL

Created Date: Within the last 4... ▾

☒ BMA-11  
SRS Documentation Edits

☒ BMA-10  
SRS Documentation

☒ BMA-9  
Gas Pump Exercise Test Cases

☒ BMA-8  
Gas Pump Exercise Code

☒ BMA-7  
Gas Pump Exercise

☒ BMA-6  
Gas Pump Exercise

☒ BMA-5  
Project Format

☒ BMA-2  
Project Proposal

☒ BMA-1  
As owners of this project, we want to develop a banking application tha...

Projects / Bank Management Appl... /  
Add epic / BMA-1

**As owners of this project, we want to develop a banking application that helps end users with essential banking transactions**

📎 👤 🔗 ⋮

**Description**  
Project Proposal

SRS

Diagrams

27 of 27 ^ ▾

🔒 👁️ 1 📄 🔗 ⋮

To Do ▾

⚡ Actions ▾

**Details** ^

Assignee  
👤 Unassigned  
[Assign to me](#)

Labels  
None

Sprint  
None

Stories and Tasks in order



Jira Software Your work ▾ Projects ▾ Filters ▾ Dashboards ▾ Teams ▾ Apps ▾ Create

Q Search 🔔 ? ⚙️ AG

Filters

Search issues

OTHER

My open issues

Reported by me

All issues

Open issues

Done issues

Viewed recently

Created recently

Resolved recently

Updated recently

View all filters

Search Save as

Bank Managem... ▾ Type: All ▾ Status: All ▾ Assignee: All ▾ + More Contains text Search Switch to JQL

Created Date: Within the last 4... ⌵

✓ BMA-23  
Task 3: Implement a feature for users to transfer funds between their ac...

✓ BMA-22  
Task 3: Implement a feature for users to transfer funds between their ac...

✓ BMA-21  
Task 2: Create a system for users to create and manage their bank acco...

■ BMA-20  
Sprint 2: Account Management

✓ BMA-19  
Task 1: Develop the login and registration pages for users to access the ...

■ BMA-18  
Sprint 1: This is the first step of the project

■ BMA-14  
Sprints Discussions

✓ BMA-13  
Use Cases

✓ BMA-12  
SRS Documentation System Block Diagram

Projects / Bank Management Appl... /  
Add epic / ✓ BMA-12

### SRS Documentation System Block Diagram

📎 🗺️ 🔗 ⋮

Description  
Make the System Block Diagram

Activity  
Show: Comments ▾ Newest first ↕

AG Add a comment...

Pro tip: press **M** to comment

🔒 👁️ 1 📄 🔗 ⋮

Done ▾ ✓ Done

⚡ Actions ▾

Details ▴

Assignee  
TK Tabish Khan  
Assign to me

Labels  
None

Sprint  
None

Story point estimate  
None

Sprint 1: Task 1: Develop the login and registration pages for users to access the system. Create a dashboard for users to view their account information

Sprint 2: Task 2: Create a system for users to create and manage their bank accounts, including savings, checking, and credit accounts.

Sprint 2: Task 3: Implement a feature for users to transfer funds between their accounts. Add the ability for users to view and download their account statements.



Jira Software Your work ▾ Projects ▾ **Filters** ▾ Dashboards ▾ Teams ▾ Apps ▾ Create

Q Search 🔔 ? ⚙️ AG

**Filters**

Search issues

OTHER

My open issues

Reported by me

All issues

Open issues

Done issues

Viewed recently

Created recently

Resolved recently

Updated recently

View all filters

**Search** Save as

Bank Managem... ▾ Type: All ▾ Status: All ▾ Assignee: All ▾ + More

Contains text Search Switch to JQL

Created Date: Within the last 4... ⌵

☒ BMA-32  
Task 9: Create documentation for the system and provide user training.

☒ BMA-31  
Task 8: Test the entire system to ensure that it meets the user require...

☒ BMA-30  
Sprint 5: Integration and Testing

☒ BMA-29  
Task 7: Add a feature for users to view spending patterns and trends. Im...

☒ BMA-28  
Task 6: Develop a reporting system for generating customer reports, su...

☒ BMA-27  
Sprint 4: Reporting and Analytics

☒ BMA-26  
Task 5: Create a feature for users to set up automatic bill payments.

☒ BMA-25  
Task 4: Develop a transaction processing system for deposit and withdr...

☒ BMA-24  
Sprint 3: Transaction Processing

Projects / Bank Management Appl... /  
Add epic / ☒ BMA-32

**Task 9: Create documentation for the system and provide user training.**

🔗 🏗️ 🔗 ...

**Description**  
Add a description...

**Activity**  
Show: Comments ▾ Newest first ⌵

AG Add a comment...

Pro tip: press M to comment

**Details** ^

Assignee  
AG ajitesh godavarthi

Labels  
None

Sprint  
BMA Sprint 5

Story point estimate  
None

Sprint 3: Task 4: Develop a transaction processing system for deposit and withdrawal transactions. Implement a feature for users to transfer funds to other bank accounts.

Sprint 3: Task 5: Create a feature for users to set up automatic bill payments.

Sprint 4: Task 7: Add a feature for users to view spending patterns and trends. Implement a system for detecting fraud and suspicious activity.

Sprint 4: Task 6: Develop a reporting system for generating customer reports, such as transaction history and account balances.

Sprint 5: Task 8: Test the entire system to ensure that it meets the user requirements. Perform integration testing with external systems, such as payment gateways and credit bureaus.

Sprint 5: Task 9: Create documentation for the system and provide user training.



Any Questions?

