

SOFTWARE DOCUMENTATION

ATM Banking System

**Maryam Fahmi, Samih Wadi, Sidra Musheer,
Jessica Morcos, Tito Osemobor, Nicanor Obasi**

**Toronto Metropolitan University
CPS 406 - Section 6
Introduction to Software Engineering**

CONTENTS

Introduction

Page 2

System Overview

Page 2

Scope

Page 2

Design Considerations

Page 3 - 5

- Assumptions
- Dependencies
- General Constraints
- Goals and Guidelines

System Architecture

Page 5 - 7

Implementation

Page 7 - 25

- Functionalities
- System Classes
- Unit Tests
- Testing

Conclusion

Page 26

Introduction

METCash was designed to simulate an ATM banking system that grants the user several functionalities namely cash withdrawal, balance inquiry, transfer funds, and other banking services.

System Overview

The system provides various functionalities to its users, which are essential for convenient banking services. The system also ensures the security of user transactions by implementing authentication and encryption mechanisms. The system must be available 24/7 to ensure that users can perform transactions at any time, handle multiple transactions simultaneously, and provide fast response times to users.

The requirements to facilitate said processes can be classified into Functional and Non - Functional Requirements, as discussed in detail in the previous phases.

Scope

The scope for an ATM (Automated Teller Machine) banking system is quite vast. ATM banking systems are an integral part of modern banking. Here are some of the key areas where they are utilized:

1. **Cash Withdrawal:** ATM banking systems allow customers to withdraw cash using their debit or credit cards, thus eliminating the need to visit a bank branch or interact with a teller.
2. **Account Balance Inquiry:** Customers can use ATM banking systems to check their account balances, view their transaction history, and print account statements.
3. **Funds Transfer:** ATM banking systems allow customers to transfer funds between their own accounts or to third-party accounts.
4. **Bill Payments:** Many ATM banking systems allow customers to pay bills for various services, such as electricity, water, and telephone, among others.

5. **Card Management:** ATM banking systems enable customers to manage their debit or credit cards, such as changing their PIN or activating or deactivating their cards.
6. **Deposits:** Some ATM banking systems allow customers to deposit cash or checks into their accounts.

Certain features that we wish to include in our system in the future include -

1. Foreign Exchange: ATM banking systems may offer foreign exchange services, allowing customers to withdraw cash in the local currency of the country they are visiting.
2. Biometric authentication technology
3. Real-time transaction monitoring
4. Multiple languages support
5. Cardless transactions
6. Integration with mobile banking apps

Overall, METCash provides a convenient and efficient way for customers to conduct various banking transactions.

Design Considerations

- **Security:** Ensure that only people who have accounts in the bank and who insert their correct personal information can access their accounts to ensure the safety of their finances.
- **User Interface and Experience:** The system should be aesthetically pleasing and provide all users with a seamless and effortless experience while using the system.
- **Functionality:** The system should be able to handle all required actions in an ATM, including withdrawals, deposits, and transfers.
- **Integration:** The ATM system should be connected to the existing database used by the bank, retaining all users and their information.
- modules, components, and their interactions.

Assumptions

- **Security issues:** The system assumes that people do not share their login information with anyone. In reality, it could be stolen, or people could share their information which puts the security of the accounts at risk.

- **Technical skills and resources:** The online system assumes that users have the devices and technical knowledge necessary to navigate the system, while that may not be the case.
- **Accessibility:** The system assumes the user is fluent in written English and can understand and communicate with the ATM system.

Dependencies

- **Banking System:** The ATM depends on the banking system, which handles errors and verification and includes the user database.
- **Digital infrastructure:** The ATM system depends on a stable server and database to guarantee a seamless and safe transaction.

General Constraints

- **Cost:** As a university project, there is no extra cost in supporting the creation and maintenance of this system, meaning it must be cost-effective and efficient with minimal maintenance requirements.
- **Time:** Time is a significant constraint on this project as we only had under 4 months to complete everything along with other responsibilities all team members had, meaning we had to choose a development process that could be done in a shorter amount of time but still produces a user-friendly and effective software.
- **Compatibility:** The software designed runs on Java Swing GUI and, as an older system, may not be compatible with all platforms.
- **Scalability:** Due to lack of cost and time, the system is not made to handle large volumes of transactions simultaneously but to serve the functionalities it's meant to do on a smaller scale.
- **Security:** Another effect of lack of cost and time is the lack of encryption and extensive authentication software. However, the system still successfully provides secure enough customer transactions within the database.

Goals and Guidelines

- **User Experience (UX):** The system should always be created with usability in mind, focusing on ease of use and navigation as well as clear feedback and error-handling messages and mechanisms.
- **Security:** The system should have simple verification and pattern-matching mechanisms used on the small database it is tested on.
- **Testing:** The system will be extensively tested to ensure functionality and security and ensure it is bug-free for users.

- **Performance:** The system should not experience any crashing, delay, or unresponsiveness (unless it's an issue with the machine rather than the system itself). It will be made lightweight to be handled by most processors with ease.

Development Methods

This ATM banking system uses 2 central development methods:

1. **Object-Oriented Design (OOD):** This approach allows us to use an object-oriented language (Java) to create a collection of objects that all interact to perform specific tasks and functions.
 2. **Use Case-Driven Design:** This secondary method was used by the team to identify all the required objects that will need to be created based on the requirements of each used case, serving as a guide throughout the design, development, and implementation process.
-

Architectural Strategies

MetCash uses a Client-Server based architecture. It is divided into two key components, the client and the server. The Client handles all UI, including design and user-friendliness. The server handles all back-end requests and functions, as well as data storage and handling. This architecture was chosen because it is the most effective at handling multiple clients and client requests.

System Architecture

The system architecture of the banking ATM software built in NetBean can be described as a client-server model, where the client (ATM machine) interacts with the server (banking system) to perform various banking transactions.

The software system can be divided into the following modules/components:

User Interface: This module is responsible for providing an intuitive and user-friendly interface for the ATM user to interact with. It includes various screens such as a login, withdrawal, deposit, balance inquiry and transaction history screen. However, for this system, the card reader is not present, and the keypad is the device's mouse or touch screen.

Database: This includes storing, creating, and managing valid users within the system. Aspects of this are customer account information and transactions made.

Transaction Processing: This module is responsible for processing various ATM transactions such as withdrawals, deposits, transfers, bill payments and balance inquiries. It interacts with the banking system to validate the user's credentials and account balance and then performs the requested transaction.

Communication: The User class interacts with other modules and components of the system, such as the ATM machine, the bank's central database, and the user interface. These interactions allow users to perform various banking transactions using the ATM machine, such as withdrawing cash, depositing funds, and checking account balances.

Security and Authentication: The system employs security measures to safeguard user data and accounts from unauthorized access, including encryption, authentication, and access control. User objects have built-in security features, such as password hashing and validation, session management, and role-based access control. The system also includes monitoring and logging features to detect and prevent fraudulent activity, comparatively within a small scale.

Interactions: The interaction between these modules can be summarized as follows.

- The user interacts with the ATM machine through the User Interface module.
- The User Interface module sends the user's request to the Transaction Processing module.
- The Transaction Processing module validates the user's credentials and account balance with the banking system through the Communication module.
- The Communication module establishes a secure connection with the banking system and sends the request.
- The banking system processes the request and sends the response back to the ATM machine through the Communication module.
- The Transaction Processing module receives the response and updates the User Interface module accordingly.
- The User Interface module displays the response to the user.

Overall, the system architecture is designed to provide a secure and reliable way for users to perform various banking transactions through an easy-to-use interface.

Detailed System Design

1. Hardware Components

- a. **Device/Display Screen:** The physical machine in which the ATM system is installed and presented to the user.
- b. **Keyboard/Mouse:** Used instead of a keypad in a physical ATM to receive client input.

2. Software Components

- a. **UI:** handles how the user interacts with the system regarding design, aesthetics, and ease of usability
- b. **Authentication software:** handles verification and authentication to ensure security for existing users
- c. **Database:** Stores all information relating to bank account holders.
- d. **Transaction software:** Manages all transactions pertaining to the user and returns information to the database for storage.

Implementation

Functionalities

- Login: takes in user input and verifies account validity
- User Menu: Displays all actions Users can perform within the system
- Deposit: Allows users to deposit a set amount of money into their checking or savings accounts.
- Withdraw: Allows users to withdraw a set amount of money into their checking or savings accounts.
- Pay Bills: Allows users to pay a certain amount of money to set Payees through their checking or savings account.
- Transfer Funds: Allows users to transfer money from their checking to their savings and vice versa, or Transfer funds to another account in the system.
- Account Balance: Allows users to check the amount of money in their checking or savings account.
- Change pin: Allows users to change their account pin.

System Classes

These classes contain the system objects and components used to carry out the Functional Requirements.

1. **ATMProject**

2. **Card**

- generateCardNumber():
- getCardNumber():
- getPin():
- setPin():
- getExpirationDate():
- generateExpirationDate():

3. **ChangePin**

- confirmButtonActionPerformed:
- validatePinInput:
- getCard()
- setPin()

4. **AccountInfo**

- optionMenuButtonActionPerformed():

5. **Payee**

- getPayeeCompany():
- getPayeeId():
- getMoneyOwed():
- setMoneyOwed():

6. **InsertCard**

- pinNumberActionPerformed
- insertButtonActionPerformed

Unit Tests

1. ATMProjectTest

Methods

- makeUsers()
- makePayees()

Purpose -

To generate users and payees in database

Expected Results -

Creates a list of users containing their details such as first name, last name, address, card number, PIN

Implementation -

```
public void makeUsers() {
    List<User> users = new ArrayList<>();
    users.add(new User("Sidra", "Musheer", "123 Main St", "1234"));
    users.add(new User("Nicanor", "Obasi", "456 Elm St", "5678"));
    for (User i : users) {
        assertNotNull(i.getFirstName());
        assertNotNull(i.getLastName());
        assertNotNull(i.getAddress());
        assertNotNull(i.getCard().getPin());
    }
}
```

```
public void makePayees() {
    List<Payee> payees = new ArrayList<>();
    payees.add(new Payee("TMU", 1000));
    payees.add(new Payee("NETFLIX", 15));
    payees.add(new Payee("AMAZON PRIME", 5));
    payees.add(new Payee("APPLE MUSIC", 6));
    for (Payee i : payees) {
        assertNotNull(i.getPayeeAccount());
        assertNotNull(i.getPayeeCompany());
    }
}
```

2. InsertCardTest

Methods

- UserReturn()
- CheckIfEmpty()

Purpose -

To ensure that the user's card information is valid and up-to-date before processing a payment or performing any other operation that requires the user's card information.

Expected Results -

is that all test cases pass, indicating that the InsertCard class methods are working as expected and validate that all card information provided by the user is correct.

Implementation -

```
public void testUserReturn(){
    this.users = new ArrayList<>();
    users.add(new User("Sidra", "Musheer", "123 Main St", "1234"));
    users.add(new User("Nicanor", "Obasi", "456 Elm St", "5678"));
    InsertCard insertCard = new InsertCard(users);
    users.get(0).getCard().setCardNumber("9290715");
    users.get(0).getCard().setExpirationDate("5/26");
    users.get(1).getCard().setCardNumber("2345678");
    users.get(1).getCard().setExpirationDate("5/26");
    assertEquals(users.get(0), insertCard.getMatchingUser(users,"9290715","5/26","1234"));
    assertEquals(null, insertCard.getMatchingUser(users,"3423451","8/26","8877"));
}

@Test
public void TestCheckIfEmpty(){
    this.users = new ArrayList<>();
    users.add(new User("Sidra", "Musheer", "123 Main St", "1234"));
    users.add(new User("Nicanor", "Obasi", "456 Elm St", "5678"));
    InsertCard insertCard = new InsertCard(users);
    users.get(0).getCard().setCardNumber("9290715");
    users.get(0).getCard().setExpirationDate("5/26");
    users.get(1).getCard().setCardNumber("2345678");
    users.get(1).getCard().setExpirationDate("5/26");
    assertEquals(false, insertCard.checkIfEmpty("2345678","5/26","5678"));
    assertEquals(true, insertCard.checkIfEmpty("3423451","", "8877"));
}
```

3. ChangePinTest

Methods

- validateInputPin()

Purpose -

To validate whether two input values for a new PIN are the same or not, which is a common requirement for changing a PIN. This method may be called after the user enters their new PIN twice to ensure that they have entered the same value both times. The ChangePin class may also include other methods for changing a user's PIN or for other related functionality.

Expected Results -

To provide a secure and reliable way for users to change or validate their PINs, which is an important aspect of many applications that require user authentication and authorization.

Implementation -

```
@Test
public void validateInputPin() {
    users.add(new User("Sidra", "Musheer", "123 Main St", "1234"));
    users.add(new User("Nicanor", "Obasi", "456 Elm St", "5678"));
    ChangePin changepin = new ChangePin(users);
    String first = "5884";
    String second = "5884";
    String third = "898";
    String fourth = "898";
    assertEquals("String fourth = \"898\"", first, second);
    assertEquals("String fourth = \"898\"", first, third);
    assertEquals(false, changepin.validatePinInput(third, fourth));
}
```

4. TransferToCheckSavTest:

Methods

- checkingsToSavings()
- savingsToCheckings()

Purpose -

To test the functionality of the TransferToCheckSav class's methods, checkingToSavings() and savingsToCheckings(), using JUnit test cases. These methods allow for the transfer of funds between a user's checking and savings accounts. By testing these methods using different input values, the test cases ensure that the methods are functioning correctly and returning expected results.

Expected Results -

To show that the expected new balances for the user's checking account, savings account, and an error message if the transfer is not successful due to insufficient funds.

Implementation -

```
public void testCheckingsToSavings() {
    users.add(new User("Sidra", "Musheer", "123 Main St", "1234"));
    users.add(new User("Nicanor", "Obasi", "456 Elm St", "5678"));
    TransferToCheckSav transfer = new TransferToCheckSav(users);
    users.get(0).setCheckingsBalance(50);
    users.get(0).setSavingsBalance(100);
    users.get(1).setCheckingsBalance(300);
    users.get(1).setSavingsBalance(100);
    assertEquals(new String[] {"0", "150", ""}, transfer.checkingToSavings(users.get(0), 50));
    assertEquals(new String[] {"", "", "Insufficient Funds"}, transfer.checkingToSavings(users.get(0), 70));
    assertEquals(new String[] {"230", "170", ""}, transfer.checkingToSavings(users.get(1), 70));
    assertEquals(new String[] {"", "", "Insufficient Funds"}, transfer.checkingToSavings(users.get(1), 500));
}

@Test
public void testSavingsToCheckings() {
    users.add(new User("Sidra", "Musheer", "123 Main St", "1234"));
    users.add(new User("Nicanor", "Obasi", "456 Elm St", "5678"));
    TransferToCheckSav transfer = new TransferToCheckSav(users);
    users.get(0).setCheckingsBalance(200);
    users.get(0).setSavingsBalance(30);
    users.get(1).setCheckingsBalance(380);
    users.get(1).setSavingsBalance(140);
    assertEquals(new String[] {"215", "15", ""}, transfer.savingsToCheckings(users.get(0), 15));
    assertEquals(new String[] {"", "", "Insufficient Funds"}, transfer.savingsToCheckings(users.get(0), 50));
    assertEquals(new String[] {"450", "70", ""}, transfer.savingsToCheckings(users.get(1), 70));
    assertEquals(new String[] {"", "", "Insufficient Funds"}, transfer.savingsToCheckings(users.get(1), 200));
}
```

5. DepositTest

Methods -

AddToCheckings()
AddToSavings()

Purpose -

The class handles cash deposits into a user's checking or savings account through methods like "addToCheckings()" and "addToSavings()" that update the account balance accordingly.

Expected Results -

The account balance will increase by the amount of the deposit.

Implementation -

```
public void testAddToCheckings() {
    users.add(new User("Sidra", "Musheer", "123 Main St", "1234"));
    users.add(new User("Nicanor", "Obasi", "456 Elm St", "5678"));
    Deposit deposit = new Deposit(users);
    assertEquals(40, Deposit.addToCheckings(users.get(0), "40"));
    assertEquals(80, Deposit.addToCheckings(users.get(0), "80"));
    assertEquals(100, Deposit.addToCheckings(users.get(1), "100"));
}

@Test
public void testAddToSavings() {
    users.add(new User("Sidra", "Musheer", "123 Main St", "1234"));
    users.add(new User("Nicanor", "Obasi", "456 Elm St", "5678"));
    Deposit deposit = new Deposit(users);
    assertEquals(40, Deposit.addToSavings(users.get(0), "40"));
    assertEquals(80, Deposit.addToSavings(users.get(0), "80"));
    assertEquals(100, Deposit.addToSavings(users.get(1), "100"));
}
```

6. WithdrawTest

Methods

- setCheckingsBalance():
- withdrawCheckings():
- withdrawSavings():

Purpose - To verify that the Withdraw class can properly handle withdrawals from both checking and savings accounts for a list of users.

Expected Results -

Account balance decreases by the withdrawal amount. In `testWithdrawCheckings()`, withdrawals of 40 and 80 decrease balances by 40 and 80 respectively and return remaining balances of 10 and 20. Withdrawal of 100 from the first user's checking account returns "Insufficient Funds". In `testWithdrawSavings()`, withdrawal of 100 from the second user's savings account decreases balance by 100 and returns a balance of 0. Withdrawals of 100 and 80 from the first user's savings account return "Insufficient Funds".

Implementation -

```
public void testWithdrawCheckings() {
    users.add(new User("Sidra", "Musheer", "123 Main St", "1234"));
    users.add(new User("Nicanor", "Obasi", "456 Elm St", "5678"));
    Withdraw withdraw = new Withdraw(users);
    users.get(0).setCheckingsBalance(50);
    users.get(1).setCheckingsBalance(100);
    assertEquals("10", withdraw.withdrawCheckings(users.get(0), "40"));
    assertEquals("20", withdraw.withdrawCheckings(users.get(1), "80"));
    assertEquals("Insufficient Funds", withdraw.withdrawCheckings(users.get(0), "100"));
}

@Test
public void testWithdrawSavings() {
    users.add(new User("Sidra", "Musheer", "123 Main St", "1234"));
    users.add(new User("Nicanor", "Obasi", "456 Elm St", "5678"));

    Withdraw withdraw = new Withdraw(users);
    users.get(0).setSavingsBalance(50);
    users.get(1).setSavingsBalance(100);
    assertEquals("Insufficient Funds", withdraw.withdrawSavings(users.get(0), "100"));
    assertEquals("Insufficient Funds", withdraw.withdrawSavings(users.get(0), "80"));
    assertEquals("0", withdraw.withdrawSavings(users.get(1), "100"));
}
```

7. PayBillTest

Methods

- `CheckIfInsufficient()`
- `PayByCheckings()`
- `PayBySavings()`

Purpose -

To manage payments made by users to payees.

Expected Results -

Checks if a user has enough funds, pays bills using the user's accounts, and returns payment status. The `PayBillTest` class tests the code by comparing expected results to actual results.

Implementation -

```
public void testCheckIfInsufficient(){...}
@Test
public void testPayByCheckings(){
    users.add(new User("Sidra", "Musheer", "123 Main St", "1234"));
    users.add(new User("Nicanor", "Obasi", "456 Elm St", "5678"));
    payees.add(new Payee("TMU", 1000));
    payees.add(new Payee("NETFLIX", 15));
    payees.add(new Payee("AMAZON PRIME", 5));
    payees.add(new Payee("APPLE MUSIC", 6));
    PayBill pay = new PayBill(users, payees);
    users.get(0).setCheckingsBalance(1000);
    users.get(1).setCheckingsBalance(400);

    assertEquals(new String[] { "0", "0", "0", "" }, pay.payByCheckings(users.get(0), "payee1", 1000));
    assertEquals(new String[] { "994", "0", "3", "" }, pay.payByCheckings(users.get(0), "payee4", 6));
    assertEquals(new String[] { "", "", "", "Wrong Amount" }, pay.payByCheckings(users.get(1), "payee3", 500));
    assertEquals(new String[] { "", "", "", "Wrong Amount" }, pay.payByCheckings(users.get(1), "payee2", 20));
}
@Test
public void testPayBySavings(){
    users.add(new User("Sidra", "Musheer", "123 Main St", "1234"));
    users.add(new User("Nicanor", "Obasi", "456 Elm St", "5678"));
    payees.add(new Payee("TMU", 1000));
    payees.add(new Payee("NETFLIX", 15));
    payees.add(new Payee("AMAZON PRIME", 5));
    payees.add(new Payee("APPLE MUSIC", 6));
    PayBill pay = new PayBill(users, payees);
    users.get(0).setSavingsBalance(1000);
    users.get(1).setSavingsBalance(1000);
}
```

8. TransferToAccount

Methods

- SavToAccCheck()
- CheckToAccSav()
- CheckToAccCheck()
- UserSavisCorrect()
- UserCheckisCorrect()

Purpose -

To test the functionality of various methods in the TransferToAccount class using JUnit testing framework. The TransferToAccountTest class tests different functionalities of the TransferToAccount class through various test methods. The assertEquals or assertEquals method is used to compare the expected and actual output of a method in the TransferToAccount class, with a pass or fail outcome. Each test method initializes a list of User objects, creates an instance of the TransferToAccount class, and calls a method to test, before comparing the expected and actual output. The test methods cover functions such as isUserCheckCorrect, isUserSavCorrect, checkToAccCheck, checkToAccSav, savToAccCheck, and savToAccSav.

Expected Results -

The expected result likely involves verifying that the correct amount of funds have been transferred and that both accounts reflect the transaction accurately.

Implementation -

```

Edit View Window Help PayBillTest.java - ~/Temp/Temp1_UnitTests.zip
PayBillTest.java
import ...

class PayBillTest {
    List<User> users = new ArrayList<>();
    List<Payee> payees = new ArrayList<>();

    @Test
    public void testCheckIfInsufficient(){...}
    @Test
    public void testPayByCheckings(){
        users.add(new User("Sidra", "Musheer", "123 Main St", "1234"));
        users.add(new User("Nicanor", "Obasi", "456 Elm St", "5678"));
        payees.add(new Payee("TMU", 1000));
        payees.add(new Payee("NETFLIX", 15));
        payees.add(new Payee("AMAZON PRIME", 5));
        payees.add(new Payee("APPLE MUSIC", 6));
        PayBill pay = new PayBill(users, payees);
        users.get(0).setCheckingsBalance(1000);
        users.get(1).setCheckingsBalance(400);

        assertEquals(new String[] {"0","0","0",""}, pay.payByCheckings(users.get(0), "payee1", 1000));
        assertEquals(new String[] {"994","0","3",""}, pay.payByCheckings(users.get(0), "payee4", 6));
        assertEquals(new String[] {"","","","Wrong Amount"}, pay.payByCheckings(users.get(1), "payee3", 500));
        assertEquals(new String[] {"","","","Wrong Amount"}, pay.payByCheckings(users.get(1), "payee2", 20));
    }
    @Test
    public void testPayBySavings(){
        users.add(new User("Sidra", "Musheer", "123 Main St", "1234"));
        users.add(new User("Nicanor", "Obasi", "456 Elm St", "5678"));
        payees.add(new Payee("TMU", 1000));
        payees.add(new Payee("NETFLIX", 15));
        payees.add(new Payee("AMAZON PRIME", 5));
        payees.add(new Payee("APPLE MUSIC", 6));
        PayBill pay = new PayBill(users, payees);
        users.get(0).setSavingsBalance(1000);
        users.get(1).setSavingsBalance(1000);
    }
}

```

```

@Test
public void testUserCheckIsCorrect() {
    users.add(new User( firstName: "Sidra", lastName: "Musheer", address: "123 Main St", pin: "1234"));
    users.add(new User( firstName: "Nicanor", lastName: "Obasi", address: "456 Elm St", pin: "5678"));
    TransferToAccount transfer = new TransferToAccount(users);

    users.get(0).setCheckingsAccountNumber("33829");
    users.get(1).setCheckingsAccountNumber("50766");

    assertEquals(users.get(0), transfer.isUserCheckCorrect(users, accNum: "33829"));
    assertEquals(null, transfer.isUserCheckCorrect(users, accNum: "78905"));
}

@Test
public void testUserSavIsCorrect() {
    users.add(new User( firstName: "Sidra", lastName: "Musheer", address: "123 Main St", pin: "1234"));
    users.add(new User( firstName: "Nicanor", lastName: "Obasi", address: "456 Elm St", pin: "5678"));
    TransferToAccount transfer = new TransferToAccount(users);

    users.get(0).setSavingsAccountNumber("33830");
    users.get(1).setSavingsAccountNumber("50765");

    assertEquals(null, transfer.isUserSavCorrect(users, accNum: "33829"));
    assertEquals(users.get(1), transfer.isUserSavCorrect(users, accNum: "50765"));
}

```

```

@Test
public void testCheckToAccCheck(){
    users.add(new User( firstName: "Sidra",   lastName: "Musheer",   address: "123 Main St",   pin: "1234"));
    users.add(new User( firstName: "Nicanor",  lastName: "Obasi",    address: "456 Elm St",   pin: "5678"));
    TransferToAccount transfer = new TransferToAccount(users);
    users.get(0).setCheckingsBalance(450);
    users.get(1).setCheckingsBalance(100);

    assertEquals(new String[] {"400","150",""}, transfer.checkToAccCheck(users.get(0),users.get(1), money: 50));
    assertEquals(new String[] {"","","Insufficient Funds"}, transfer.checkToAccCheck(users.get(1),users.get(0), money: 250));
    assertEquals(new String[] {"","","Insufficient Funds"}, transfer.checkToAccCheck(users.get(0),users.get(1), money: 500));
    assertEquals(new String[] {"50","500",""}, transfer.checkToAccCheck(users.get(1),users.get(0), money: 50));
}

@Test
public void testCheckToAccSav(){
    users.add(new User( firstName: "Sidra",   lastName: "Musheer",   address: "123 Main St",   pin: "1234"));
    users.add(new User( firstName: "Nicanor",  lastName: "Obasi",    address: "456 Elm St",   pin: "5678"));
    TransferToAccount transfer = new TransferToAccount(users);
    users.get(0).setCheckingsBalance(90);
    users.get(0).setSavingsBalance(50);
    users.get(1).setCheckingsBalance(100);
    users.get(1).setSavingsBalance(230);

    assertEquals(new String[] {"40","280",""}, transfer.checkToAccSav(users.get(0),users.get(1), money: 50));
    assertEquals(new String[] {"","","Insufficient Funds"}, transfer.checkToAccSav(users.get(1),users.get(0), money: 150));
    assertEquals(new String[] {"","","Insufficient Funds"}, transfer.checkToAccSav(users.get(0),users.get(1), money: 500));
    assertEquals(new String[] {"50","100",""}, transfer.checkToAccSav(users.get(1),users.get(0), money: 50));
}

@Test
public void testSavToAccCheck(){
    users.add(new User( firstName: "Sidra",   lastName: "Musheer",   address: "123 Main St",   pin: "1234"));
    users.add(new User( firstName: "Nicanor",  lastName: "Obasi",    address: "456 Elm St",   pin: "5678"));

```

```

    users.get(0).setSavingsBalance(50);
    users.get(1).setCheckingsBalance(100);
    users.get(1).setSavingsBalance(230);

    assertEquals(new String[] {"40","280",""}, transfer.checkToAccSav(users.get(0),users.get(1), money: 50));
    assertEquals(new String[] {"","","Insufficient Funds"}, transfer.checkToAccSav(users.get(1),users.get(0), money: 150));
    assertEquals(new String[] {"","","Insufficient Funds"}, transfer.checkToAccSav(users.get(0),users.get(1), money: 500));
    assertEquals(new String[] {"50","100",""}, transfer.checkToAccSav(users.get(1),users.get(0), money: 50));
}

@Test
public void testSavToAccCheck(){
    users.add(new User( firstName: "Sidra",   lastName: "Musheer",   address: "123 Main St",   pin: "1234"));
    users.add(new User( firstName: "Nicanor",  lastName: "Obasi",    address: "456 Elm St",   pin: "5678"));
    TransferToAccount transfer = new TransferToAccount(users);
    users.get(0).setCheckingsBalance(90);
    users.get(0).setSavingsBalance(50);
    users.get(1).setCheckingsBalance(100);
    users.get(1).setSavingsBalance(230);

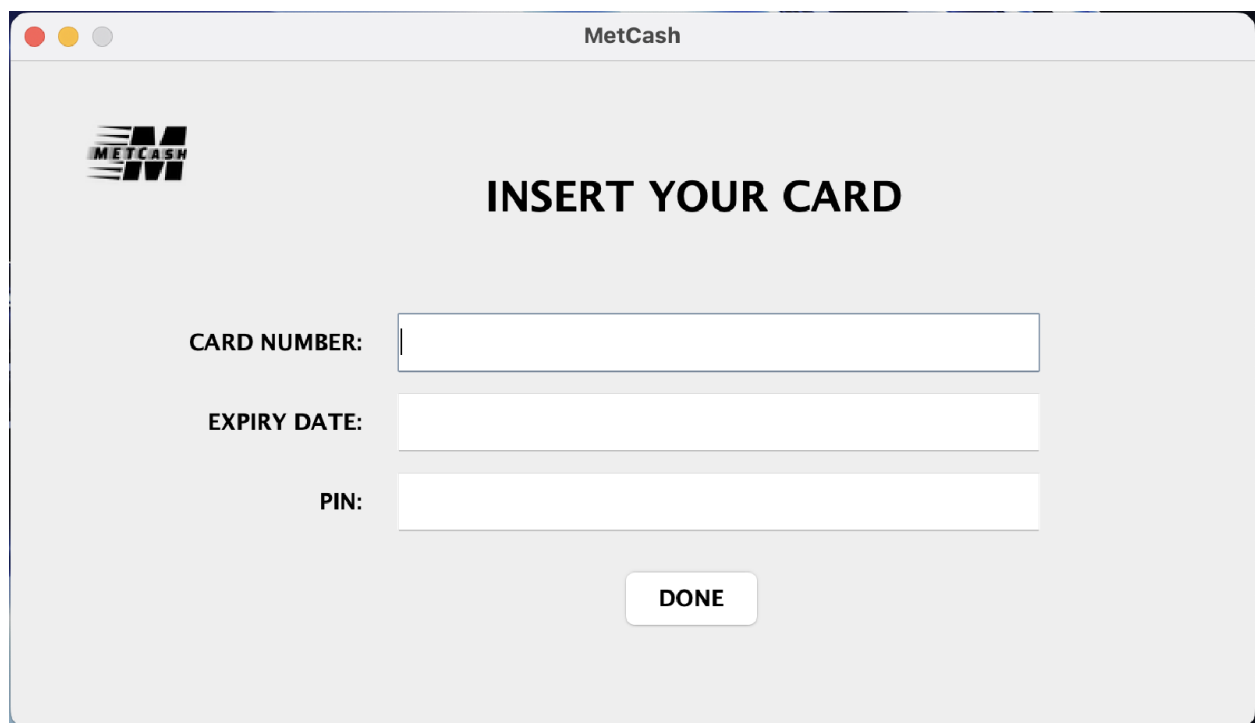
    assertEquals(new String[] {"0","150",""}, transfer.savToAccCheck(users.get(0),users.get(1), money: 50));
    assertEquals(new String[] {"","","Insufficient Funds"}, transfer.savToAccCheck(users.get(1),users.get(0), money: 300));
    assertEquals(new String[] {"","","Insufficient Funds"}, transfer.savToAccCheck(users.get(0),users.get(1), money: 60));
    assertEquals(new String[] {"180","140",""}, transfer.savToAccCheck(users.get(1),users.get(0), money: 50));
}

@Test
public void testSavToAccSav(){...}

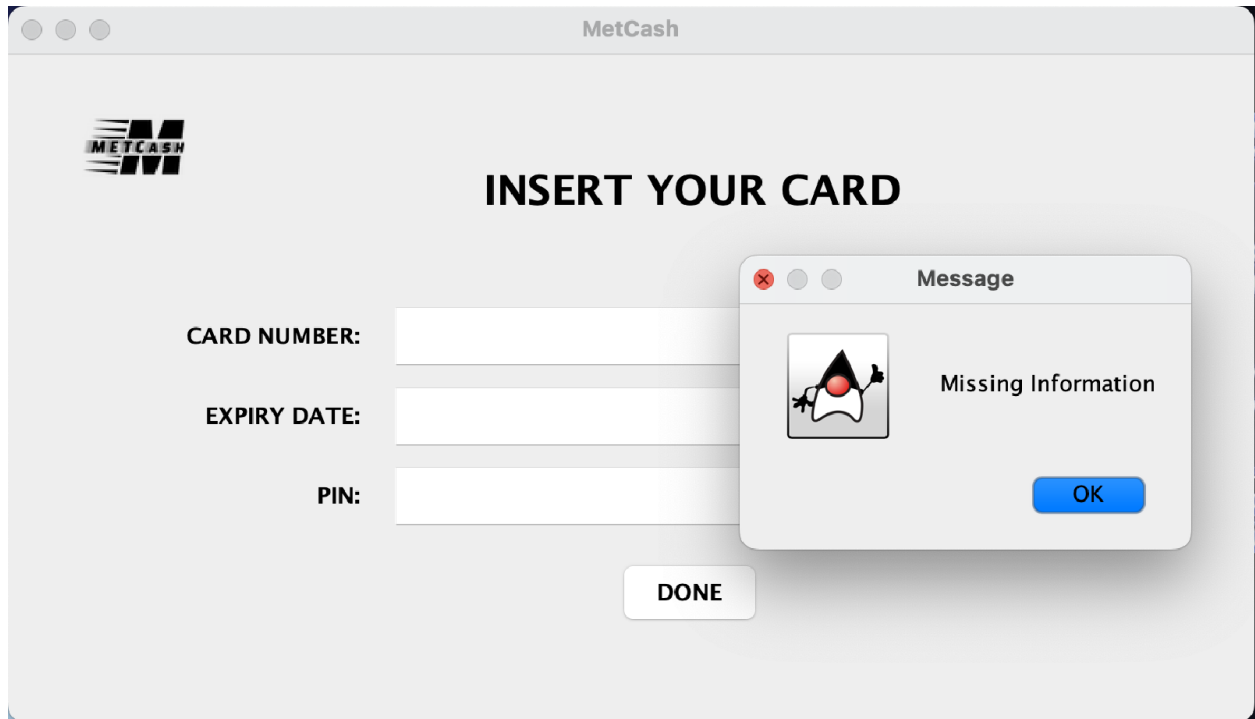
```

Testing

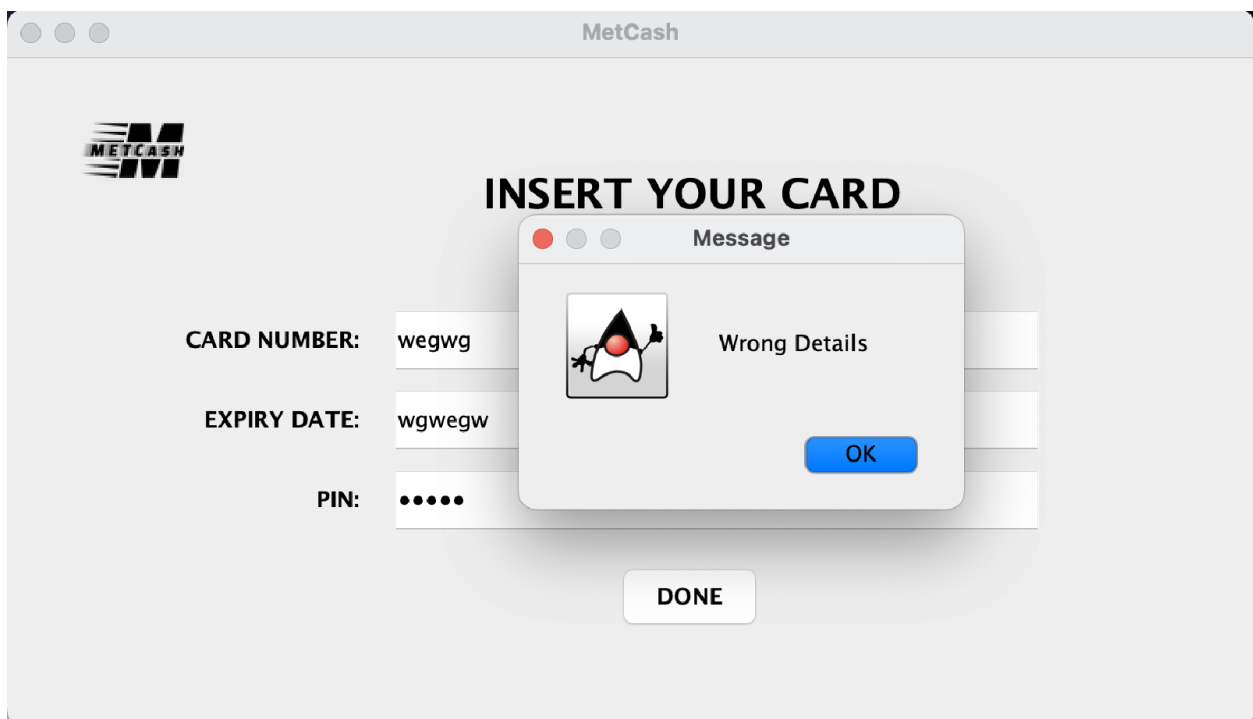
Loading screen

The image shows a screenshot of the MetCash application window. The window has a title bar with the text "MetCash" and three colored window control buttons (red, yellow, grey). The main content area has a light grey background. In the top left corner, there is a small version of the MetCash logo. In the center, the text "INSERT YOUR CARD" is displayed in a bold, black, sans-serif font. Below this text, there are three input fields. The first field is labeled "CARD NUMBER:" and is empty. The second field is labeled "EXPIRY DATE:" and is empty. The third field is labeled "PIN:" and is empty. At the bottom center, there is a white button with the text "DONE" in black.

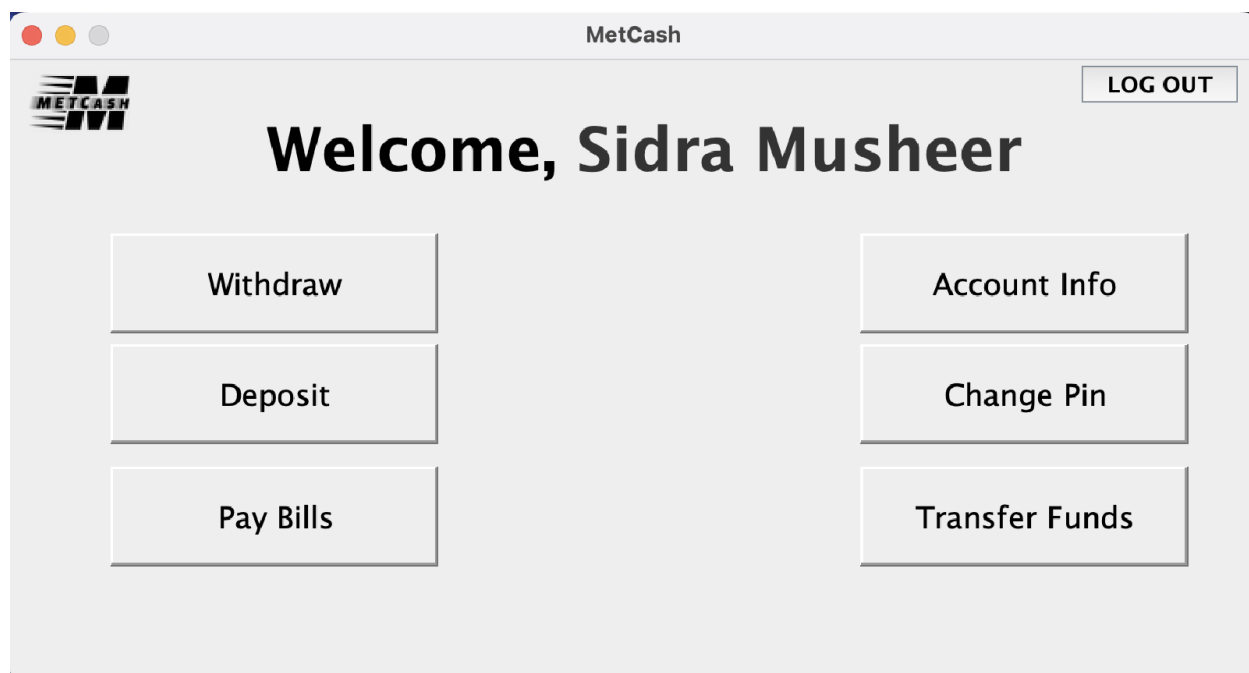
Insert Card: Insert Card Menu Screen



Login Screen: Details are missing error



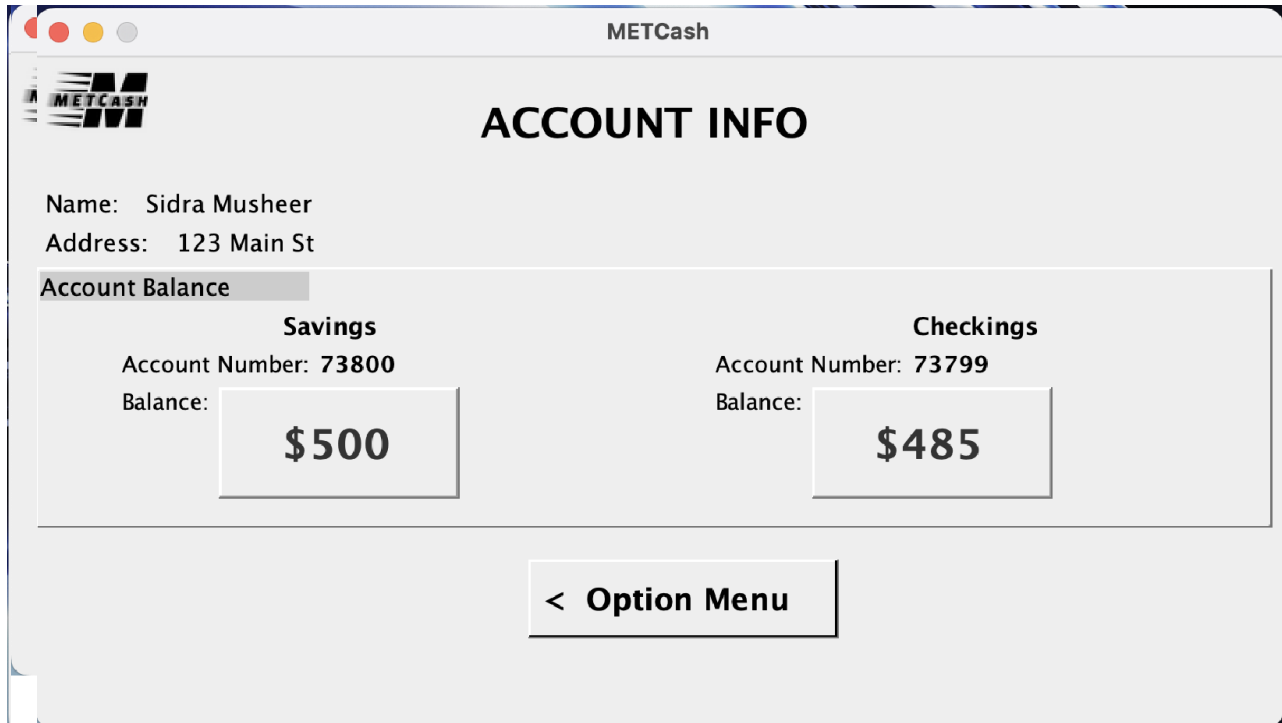
Insert Card: Error if Wrong Details entered



Transaction Menu



Withdraw: Error if Confirm button clicked without choosing amount



A screenshot of a web browser window titled "METCash". The page displays account information for "Sidra Musheer" at "123 Main St". Under the heading "ACCOUNT INFO", there is a section titled "Account Balance" which is divided into two columns: "Savings" and "Checkings". The Savings account (number 73800) has a balance of \$500, and the Checkings account (number 73799) has a balance of \$485. At the bottom of the page is a button labeled "< Option Menu".

METCash

ACCOUNT INFO

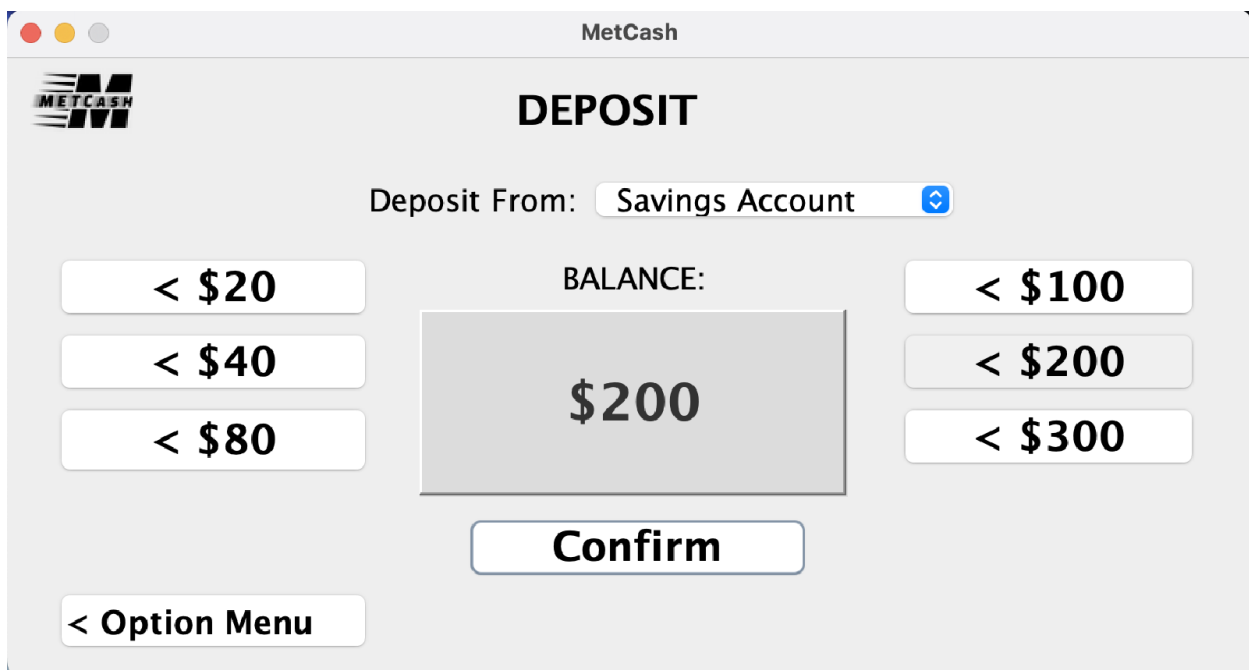
Name: Sidra Musheer
Address: 123 Main St

Account Balance

Savings	Checkings
Account Number: 73800	Account Number: 73799
Balance: \$500	Balance: \$485

< Option Menu

Account Information Page



A screenshot of a web browser window titled "MetCash". The page displays a "DEPOSIT" form. The "Deposit From:" dropdown menu is set to "Savings Account". In the center, the "BALANCE:" is shown as \$200. To the left of the balance are three buttons: "< \$20", "< \$40", and "< \$80". To the right are three buttons: "< \$100", "< \$200", and "< \$300". At the bottom center is a "Confirm" button, and at the bottom left is a "< Option Menu" button.

MetCash

DEPOSIT

Deposit From: Savings Account

BALANCE:

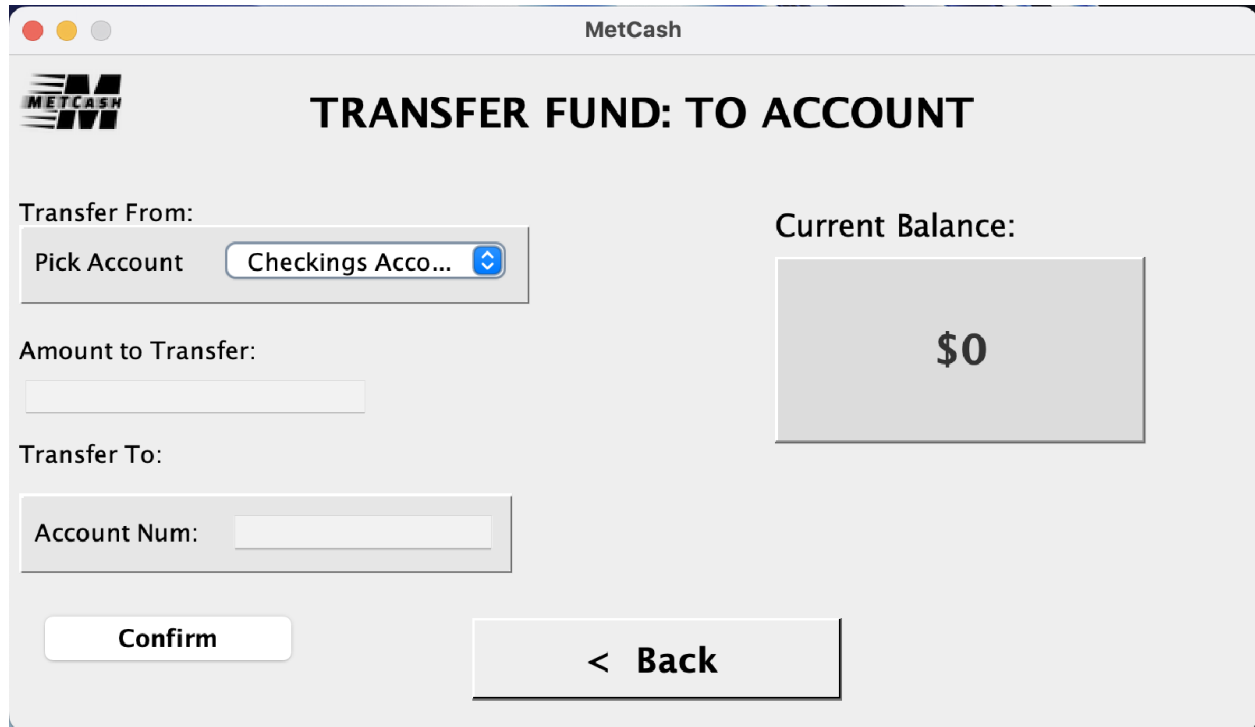
\$200

< \$20 **< \$40** **< \$80** **< \$100** **< \$200** **< \$300**

Confirm

< Option Menu

Deposit: Adding Money to Savings Account



MetCash

TRANSFER FUND: TO ACCOUNT

Transfer From:

Pick Account Checkings Acco...

Current Balance:

\$0

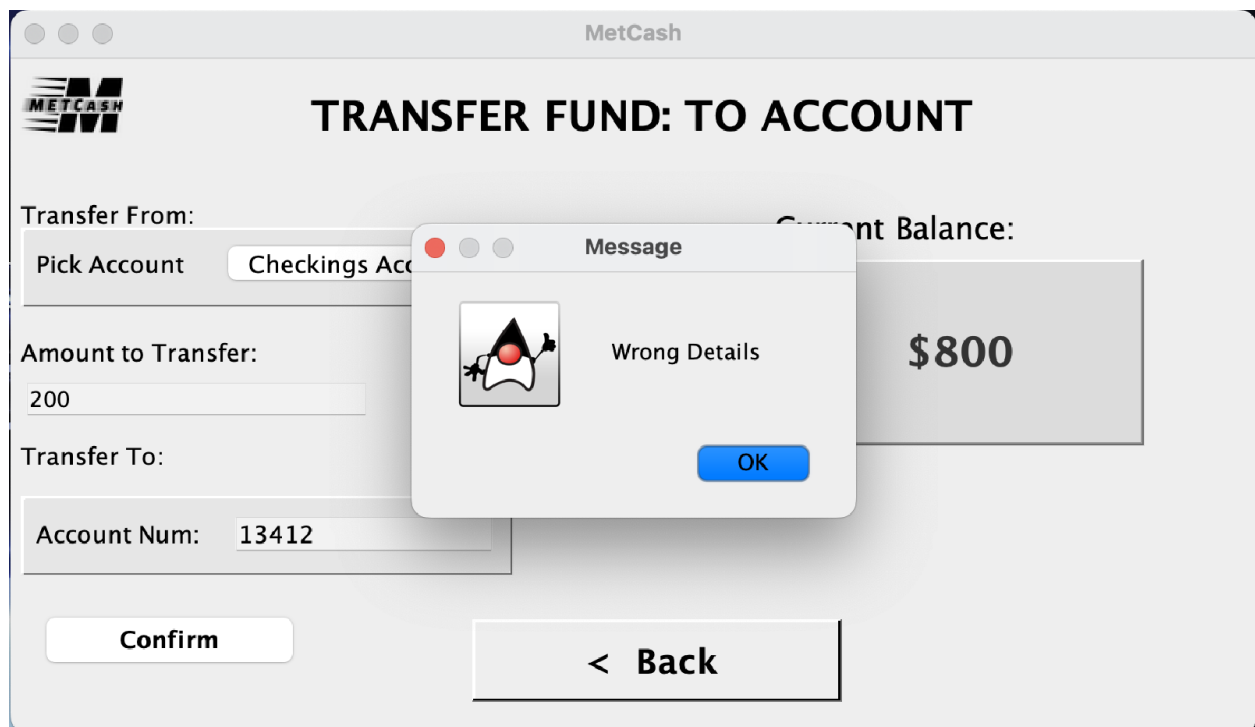
Amount to Transfer:

Transfer To:

Account Num:

Confirm < Back

Transfer Funds: Transfer Funds to different account screen



MetCash

TRANSFER FUND: TO ACCOUNT

Transfer From:

Pick Account Checkings Acco...

Current Balance:

\$800

Amount to Transfer:

200

Transfer To:

Account Num: 13412

Confirm < Back

Message

Wrong Details

OK

Transfer Funds: Error if Account num is less than needed values

MetCash

TRANSFER FUNDS: CHECKING/SAVINGS

Amount to Transfer:

Savings Balance

\$500

Checking Balance

\$400

Transfer From:

Checking To Savings >

Savings To Checking >

< Back

Transfer Funds: Checking and Savings Transfers screens

MetCash

TRANSFER FUNDS: CHECKING/SAVINGS

Amount to Transfer:

Message

Missing Information

OK

Transfer From:

Checking To Savings >

Savings To Checking >

< Back

Transfer Funds: If Amount to transfers field is empty

MetCash

TRANSFER FUNDS: CHECKING/SAVINGS

Amount to Transfer:
345

Savings Balance
\$155

Checking Balance
\$745

Transfer From:
Checking To Savings >
Savings To Checking >

< Back

Transfer Funds: Successful Checking/Saving Transfer

MetCash

TRANSFER FUND: TO ACCOUNT


Transfer From:
Pick Account Checkings Ac

Amount to Transfer:
200

Transfer To:
Account Num:

Current Balance:
\$800

Confirm **< Back**

Message
 Missing Information
OK

Transfer Funds: Error if Account num field is empty

Conclusion

The ATM Banking System is a critical application for banks to provide convenient banking services to their customers. The system's architecture and functional and non-functional requirements ensure that the system is secure, available, and reliable for users.
