ATM Simulation Documentation

Object-Oriented Programming Project 2021

Mentor:
Yash Chandak

Team Members:
Rishabh Barnwal (2020A7PS1677P)
Meet Patel (2020A7PS0973P)

ATM

An automated teller machine (ATM) is an electronic banking outlet that allows customers to complete basic transactions without the aid of a branch representative or teller. Anyone with a valid authenticated card can access ATMs.

GUIDE

The project is based on working an ATM with a database generated with random inputs to allow dynamics to the database. The approach for this project is primarily based on how the ATM functions when a known user enters their credentials to fulfill the requirements.
Four main methods are implemented in this project.

1. Withdrawal of cash from the account
2. Deposit of money to the account
3. Balance check on account
4. Changing Card Pin

With this generic topic, finding something unique was definitely a tough job. To which, the approach taken towards this project was with fresh mind and completely made out of scratch.

STRUCTURE

All these functions are encapsulated into two significant classes.

1. **User():** This class defines the structure of the user as provided to an ATM system, which is a composition of two subclasses, sharing the balance amount.

   1.1. **Account():** This class defines the methods for transactions and changes with the account balance of the particular user it is nested in with. It updates the shared balance according to the procedures called within.

   1.2. **Card():** This class defines the methods for verifying a user's card.

2. **ATM():** This is a driver class for the complete program that defines functions and consists of the database generated and saved during the execution of the code. Saving the databse is to signify the functionality of storing a hard copy of transactions and respective time for manual debug.

CLASSES

A brief description of all the methods and a few essential variables in classes.

1. **User():**
   1.1. It has its variable for storing balance, a getter method and a setter method for the same. The setter method uses a MATH floor library to set to a random value.

2. **Account():**
   2.1. Nested inside User(), this class defines all the methods to manipulate the balance variable of the user class using methods to withdraw, deposit and display.

3. **Card():**
   3.1. Nested inside User(), this class defines the card's number and pin associated with the user it is nested within. It implements a getter and setter. The getter method implements the MATH floor library to set numbers and pins as random values.
   3.2. It also has a method to override the initial pin that was randomly generated.

4. **ATM():**
   4.1. This is the driver class where all magic happens. This class implements the major switch case that is displayed in front of the user.
   4.2. The database is generated when the class runs, thus showing that Bank already has users and no user is new for the ATM. The size of the database can be changed in the code.

4.3. This class also implements writing the object data in a file to show that an ATM has its feature of storing transactions with dates to maintain the latest record of all user data in real-time.

4.4. This class has methods that set the machine balance, verify the user using the card number and card pin and check if withdrawals are in multiples of 100 or 500.

4.5. It also implements a method that in itself implements a thread to delay output. This is to make the terminal UI feel interactive to users.

(There are many helper variables and methods which are not necessary to be defined, they are just to fulfil the need of booleans in while loops or temporary storage)

Walkthrough of i/p and o/p:

At every simulation instance, the user is asked to provide some input that defines the next state. The complete simulation is based on what the user wants. The code never terminates abruptly without the permission of the user.

1. The very first input that is asked is whether the user is an admin or not. There is no use of this input for the future, and it's just a hidden element to show the privacy of the database, thus if the user prints the database from the bank in a thread.

2. Next, it asks the user for the card number and card pin. It checks whether the card number exists in the database or not.
    2.1. If not, the user is asked to input a number that is present. Otherwise, the user can exit.
    2.2. If the card pin entered doesn't match the respective card number, the system asks for the correct pin or exit the simulation again.

2.3. If both these inputs are correct, the verification is complete.

3. A welcome message is printed with the user's options to manipulate its account balance or account pin.
    3.1. Input 0 exits the simulation
    3.2. Input 1 Goes for cash withdrawal
        3.2.1. If the cash is more than the machine balance, a message with the balance of the machine will be displayed.
        3.2.2. If the cash is more minor than the user balance, a message with the user balance will be displayed.
        3.2.3. If the cash is not multiples of 100 or 500, an error message pointing this out will be displayed.
        3.2.4. Else if everything goes fine, ATM asks for receipt printing or not. If yes, it prints, else it doesn't.
    3.3. Input 2 Goes for cash deposit
        3.3.1. There is no restriction on assurance of cash.
        3.3.2. ATM asks to print a receipt or not. If yes, it does. Else, it doesn't.
    3.4. Input 3 Goes for display balance of user that prints current balance.
    3.5. Input 4 Goes for pin change.
        3.5.1. It asks for double verification to re-enter the card number and pin.
        3.5.2. Since it's a sensitive process, any error in input leads to an ERROR display and takes the user back to the main menu as a punishment.
        3.5.3. If the inputs are well, it asks for a new card pin and checks the range of three digits. If not, it corrects the user to enter a pin in this range only

3.5.4. If the new pin is the same as the old, the input is not valid, and it loops on asking for a valid new pin.

Limitations:

1. The ATM can have a logout feature instead of exit simulation.
2. There are no internal account transfers.
3. There are no different types of users or cards in the database.
4. There are instances where Collections would have helped a lot, like finding user for verification, but the same is implemented with proper methods. The only fact that was unknown while making this project is encapsulating nested classes in Collections. Thus normal arrays were used.

Fortunately, there are no bugs that the Simulation runs into as far as its tested, and every case is caught with a while loop until it satisfies a necessary condition. Every negative number is considered invalid input whenever necessary and taken care of by taking its absolute value.

Future Work:

1. The logout feature can be implemented to ask users to log out and enter as new users into the system.
2. Would add an internal fund transfer system from one user to another. This will indeed create more flexibility in transactions connecting all users in the database.
3. Different types of users and even cards could be implemented to show hierarchy. Unique features and new methods will be shown or kept hidden accordingly.
4. Could learn GUI in the future and make the interface more interactive for the user. I have tried to do so in the terminal itself which is close but it is no match to GUI.
5. Implementing Collections for storing my database instead of normal arrays. I linked list could have worked perfect here.

Design Patterns:

1. **Favour composition over Inheritence:**

   The principle defines that the classes must be polymorphic in nature, less redundant and more flexible towards changes. Definitely its a good practice to implement. For the future developments, in order to implement different type of users and cards system, composition would have helped me a lot over inheritance after a certain level where the system would have achieved a lot of variations.

   Ex: A "gold card" could be a "card" with 3 digit pin, or A "gold card" could be a "card" also with a 3 digit pin.

2. **Program to an interface not implementation:**

   An interface can be seen as a contract between an object and its clients.
   This pattern can be implemented again when there are different instances of the same class of feature in the class that can be overridden throughout different types of the base class. Its a powerful method to show what is required and what is not required. Majority of libraries are made using this pattern.

3. **Strive for loose coupling between objects that interact**

With respect to my project, I believe that this pattern wouldn't have been much in practice given in my knowledge base, since this project required less flexibility in between sensitive objects that should be linked perfectly. I would definitely like to know about its implementation in future for this project, if given a chance and time.

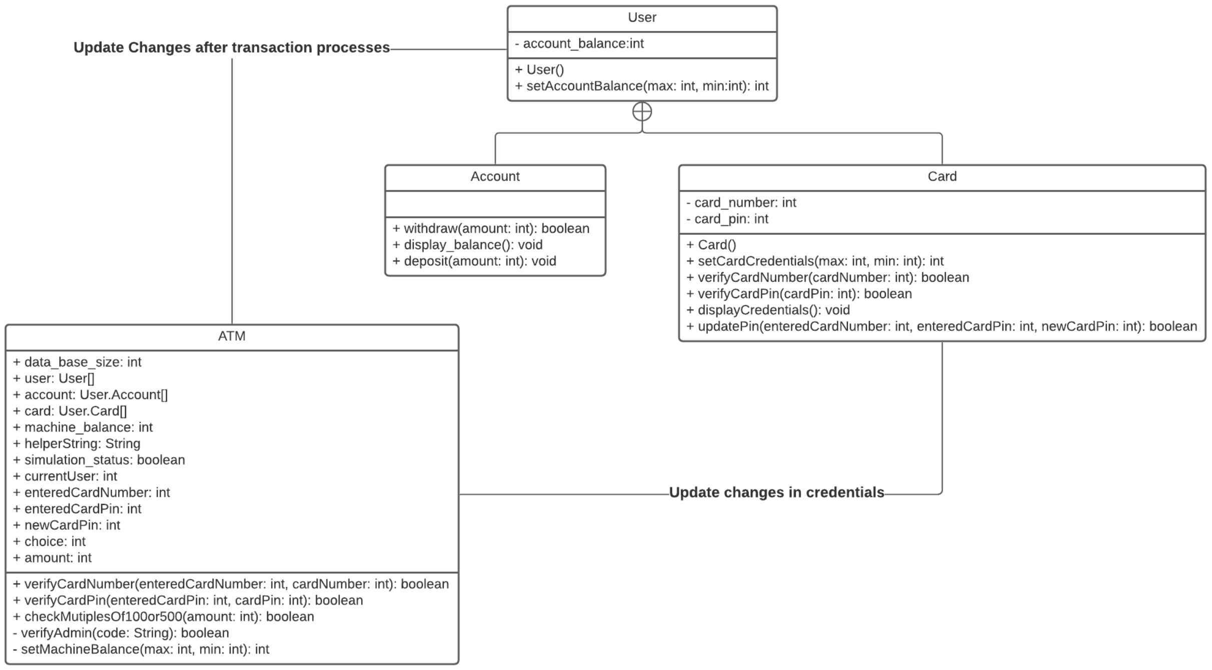
4. Classes should be open for extension and closed for modification

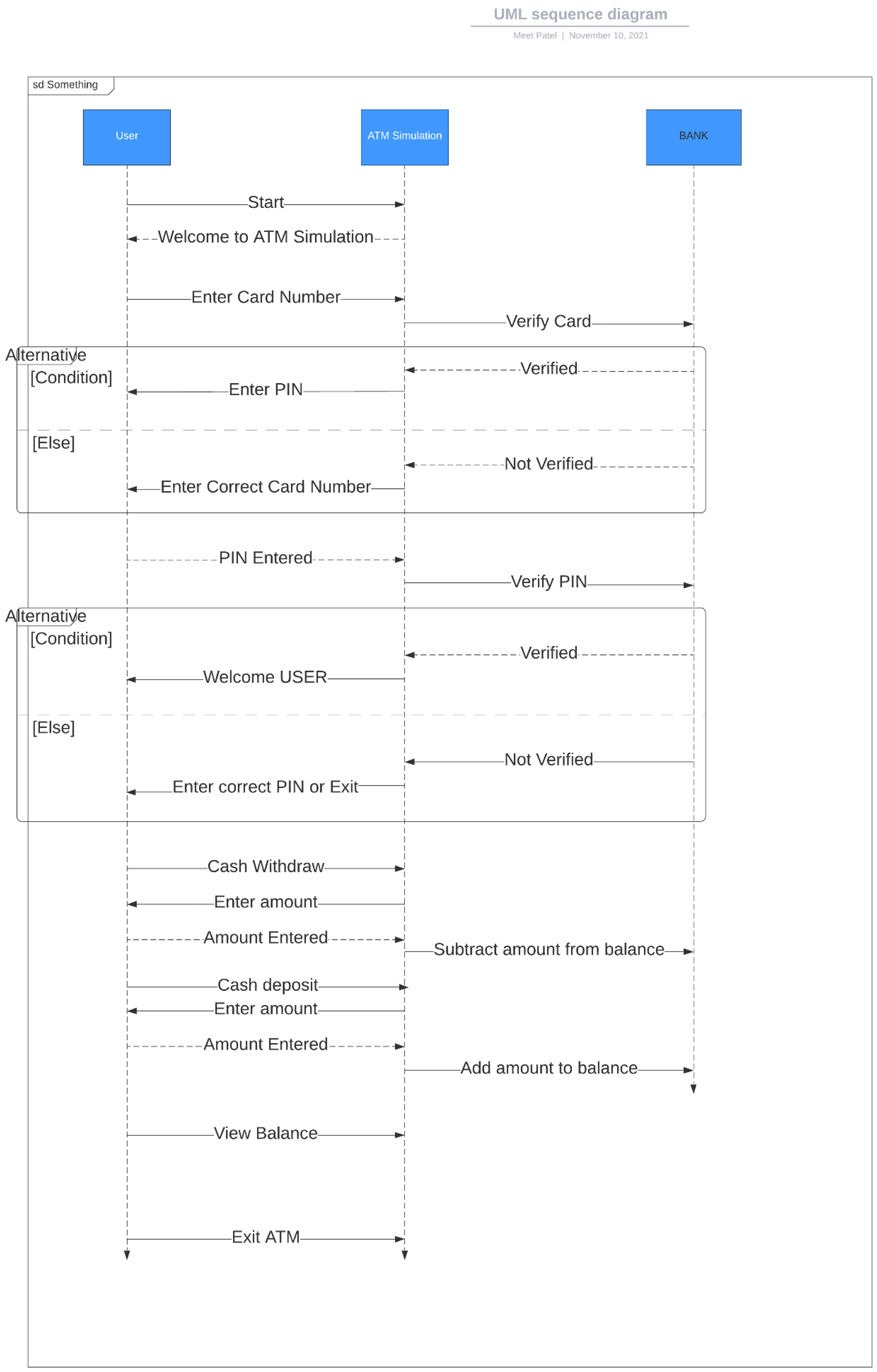
I really liked this this pattern for this project and even feel that it somehow encapsulates the idea as well. This pattern is really helpful in making secure systems where someone wouldn't like some of the data to get overridden by some manipulations.

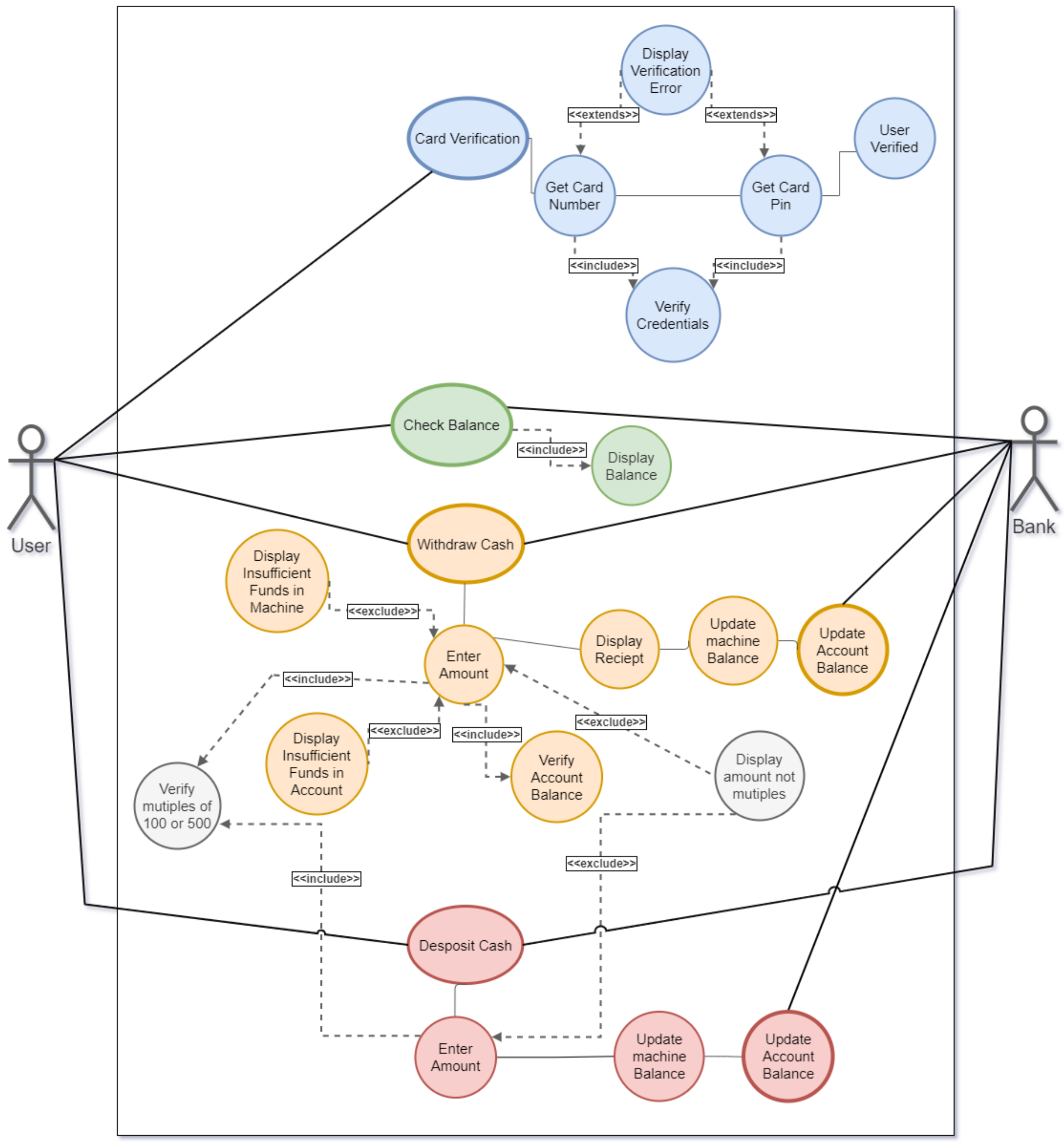
Ex: With different types of cards to be implemented, the risk of not implementing some important functions or overriding those functions is high.

5. Depend on abstraction, do not depend on concrete classes

As stated earlier, having one class or many classes for every different class is not a good practice. Rather than hardcoding each class, extending an abstract is a good approach to create many cards and users in this project. Definitely something that can be clubbed with other patterns to achieve something more flexible.


User
- account_balance:int
+ User()
+ setAccountBalance(max: int, min:int): int
Update Changes after transaction processes
Account
+ withdraw(amount: int): boolean
+ display_balance(): void
+ deposit(amount: int): void
Card
- card_number: int
- card_pin: int
+ Card()
+ setCardCredentials(max: int, min: int): int
+ verifyCardNumber(cardNumber: int): boolean
+ verifyCardPin(cardPin: int): boolean
+ displayCredentials(): void
+ updatePin(enteredCardNumber: int, enteredCardPin: int, newCardPin: int): boolean
ATM
+ data_base_size: int
+ user: User[]
+ account: User.Account[]
+ card: User.Card[]
+ machine_balance: int
+ helperString: String
+ simulation_status: boolean
+ currentUser: int
+ enteredCardNumber: int
+ enteredCardPin: int
+ newCardPin: int
+ choice: int
+ amount: int
+ verifyCardNumber(enteredCardNumber: int, cardNumber: int): boolean
+ verifyCardPin(enteredCardPin: int, cardPin: int): boolean
+ checkMutiplesOf100or500(amount: int): boolean
- verifyAdmin(code: String): boolean
- setMachineBalance(max: int, min: int): int
Update changes in credentials


UML sequence diagram
Meet Patel | November 10, 2021
sd Something
User
ATM Simulation
BANK
Start
Welcome to ATM Simulation
Enter Card Number
Verify Card
Alternative
[Condition]
Verified
Enter PIN
[Else]
Not Verified
Enter Correct Card Number
PIN Entered
Verify PIN
Alternative
[Condition]
Verified
Welcome USER
[Else]
Not Verified
Enter correct PIN or Exit
Cash Withdraw
Enter amount
Amount Entered
Subtract amount from balance
Cash deposit
Enter amount
Amount Entered
Add amount to balance
View Balance
Exit ATM


Display Verification Error
<<extends>>
<<extends>>
Card Verification
User Verified
Get Card Number
Get Card Pin
<<include>>
<<include>>
Verify Credentials
Check Balance
<<include>>
Display Balance
User
Bank
Withdraw Cash
Display Insufficient Funds in Machine
<<exclude>>
Enter Amount
Display Reciept
Update machine Balance
Update Account Balance
<<include>>
Display Insufficient Funds in Account
<<exclude>>
<<include>>
<<exclude>>
Verify Account Balance
Display amount not mutiples
Verify mutiples of 100 or 500
<<include>>
<<exclude>>
Desposit Cash
Enter Amount
Update machine Balance
Update Account Balance