PDP MINI PROJECT
BY SADAKOPA RAMAKRISHNAN IT C
3122 22 5002 109

Problem Statement: NetBanking System

1. Strategy Design Pattern:

Problem:
Currently, the `NetBankingApp` class directly executes deposit and withdrawal operations based on user input. If you want to support different transaction strategies (e.g., transaction fees, interest rates), you can use the Strategy pattern.

Implementation:
Create a `TransactionStrategy` interface with methods like `execute_deposit` and `execute_withdraw`. Implement concrete strategies (e.g., `RegularTransactionStrategy`, `FeeTransactionStrategy`). Modify the `NetBankingApp` to use a strategy object for transactions.

2. Decorator Design Pattern:

Problem:
Extend the functionality of the `BankAccount` class dynamically without modifying its code directly.

Implementation:
Create decorators (e.g., `TransactionLoggerDecorator`, `TransactionValidatorDecorator`) that wrap the `BankAccount`

class, adding logging or validation functionality. This allows you to compose different behaviors.

3. State Design Pattern:

Problem:
The current code handles the deposit and withdrawal operations in the same class. If you want to represent the state of an account (e.g., frozen, active), you can use the State pattern.

Implementation:
Define a `AccountState` interface with methods like `deposit` and `withdraw`. Implement concrete states (e.g., `ActiveState`, `FrozenState`). Modify the `BankAccount` class to have a state attribute and delegate operations to the current state.

4. Chain of Responsibility Design Pattern:

Problem:
You might want to handle requests (e.g., transaction approvals) in a chain where each handler in the chain can either process the request or pass it to the next handler.

Implementation:
Create a chain of handlers (e.g., `TransactionApproverHandler`, `TransactionLoggerHandler`). Each handler decides whether to handle the request or pass it to the next handler. Modify the `NetBankingApp` to use this chain.

5. Composite Design Pattern:

Problem:
If you want to treat individual transactions and the entire account balance uniformly, consider using the Composite pattern.

Implementation:
Create a `TransactionComponent` interface with methods like `get_balance`. Implement leaf components (e.g., `SingleTransaction`, `AccountBalance`) and composite components (e.g., `TransactionGroup`). This way, you can treat a single transaction or the entire account balance uniformly.

Code:

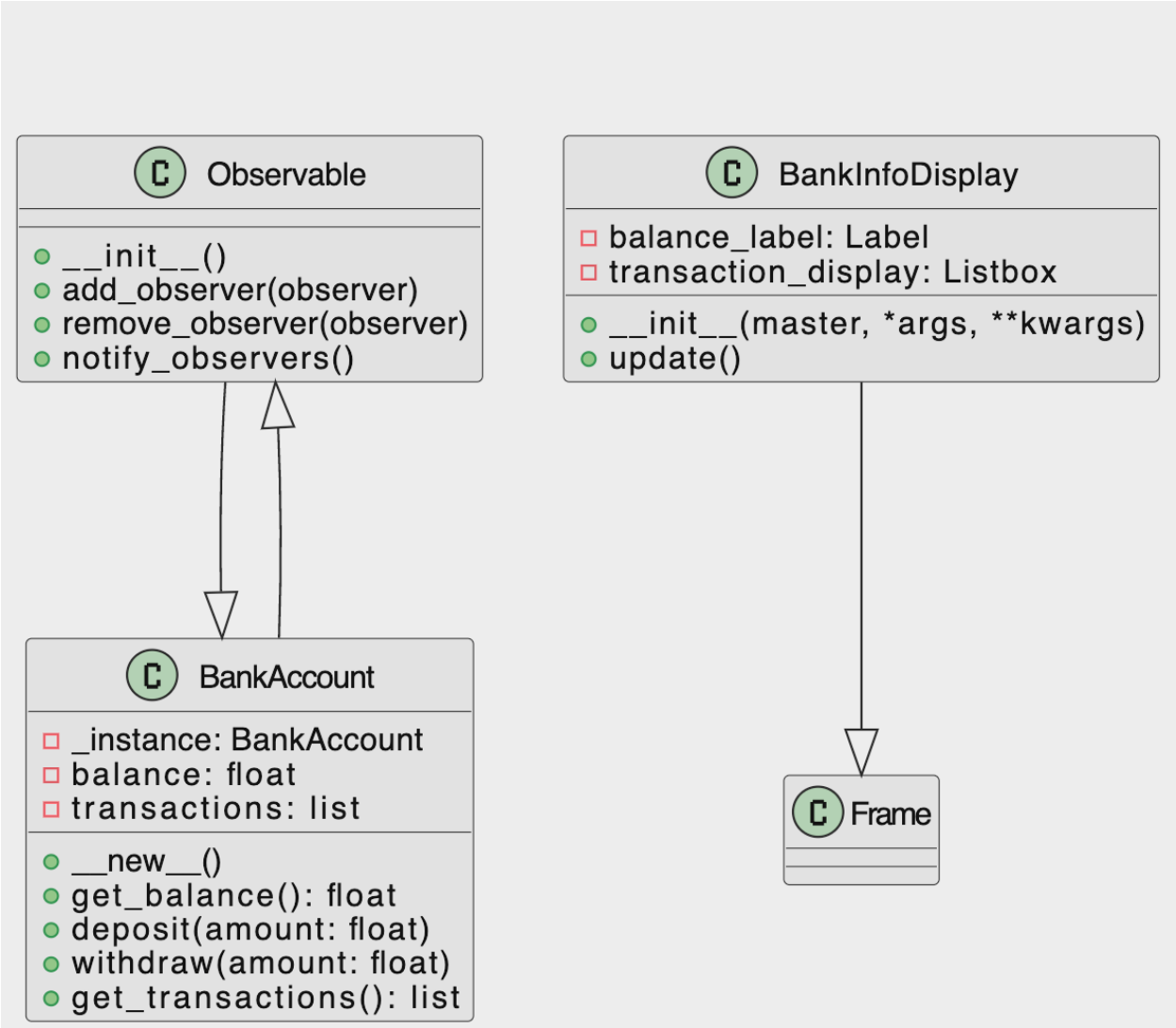
```python
import tkinter as tk
from tkinter import messagebox
from abc import ABC, abstractmethod
from datetime import datetime

# Observer Design Pattern
class Observable(ABC):
  def __init__(self):
    self._observers = []

  def add_observer(self, observer):
    self._observers.append(observer)

  def remove_observer(self, observer):
```

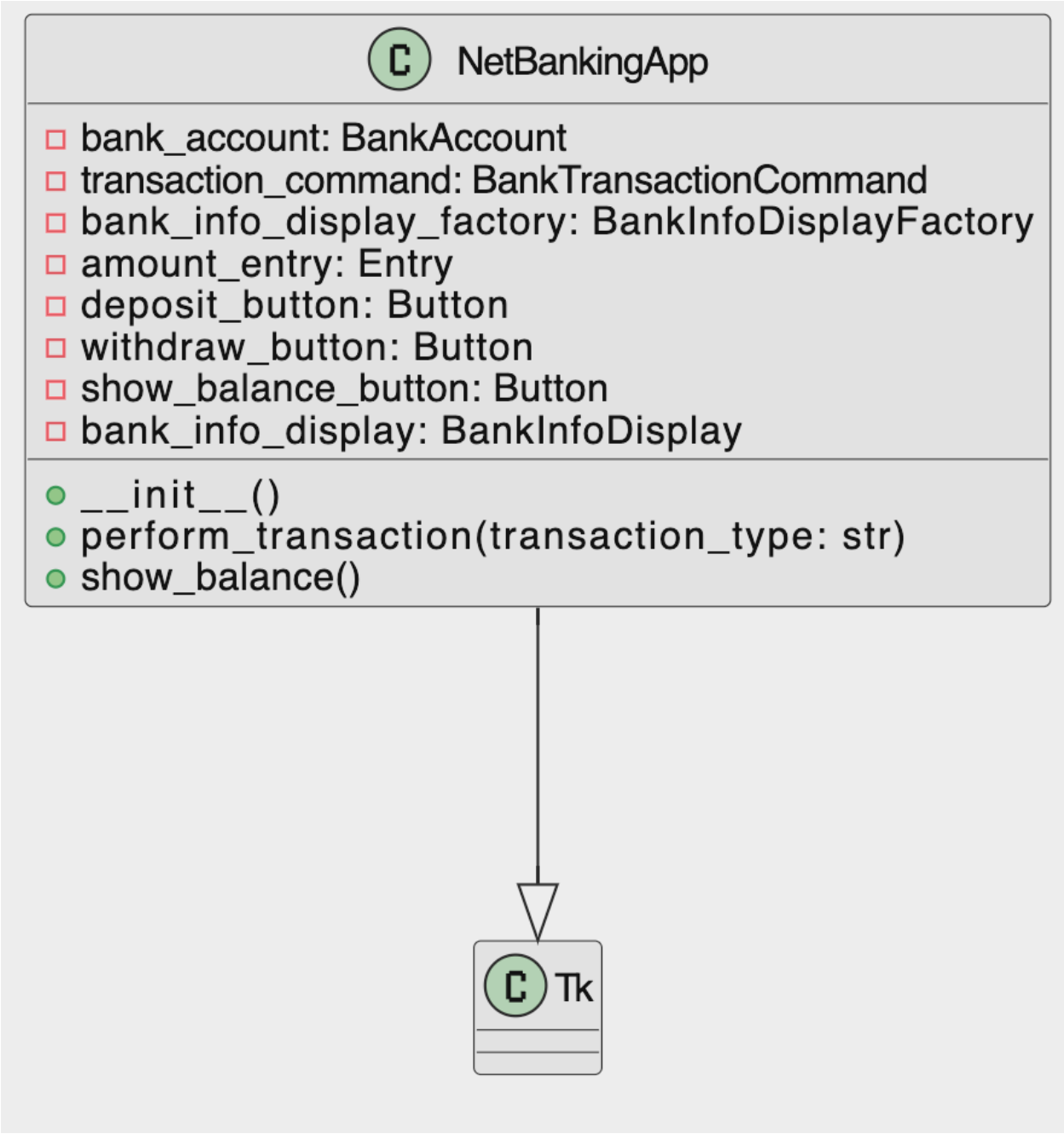
```
    self._observers.remove(observer)

  def notify_observers(self):
    for observer in self._observers:
      observer.update()

# Singleton Design Pattern
class BankAccount(Observable):
  _instance = None

  def __new__(cls):
    if not cls._instance:
      cls._instance = super(BankAccount, cls).__new__(cls)
      cls._instance.balance = 0
      cls._instance.transactions = []
    return cls._instance

  def get_balance(self):
    return self.balance

  def deposit(self, amount):
    self.balance += amount
    self.transactions.append(f"Deposit: +{amount}
({datetime.now().strftime('%Y-%m-%d %H:%M:%S')})")
        self.notify_observers()

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            self.transactions.append(f"Withdrawal: -{amount}
({datetime.now().strftime('%Y-%m-%d %H:%M:%S')})")
            self.notify_observers()
        else:
            messagebox.showwarning("Insufficient Balance", "You do not have
sufficient balance.")

  def get_transactions(self):
    return self.transactions

# Concrete Observer
class BankInfoDisplay(tk.Frame):
  def __init__(self, master, *args, **kwargs):
    tk.Frame.__init__(self, master, *args, **kwargs)
    self.balance_label = tk.Label(self, text="Balance: $0.00", font=("Helvetica",
16))
```

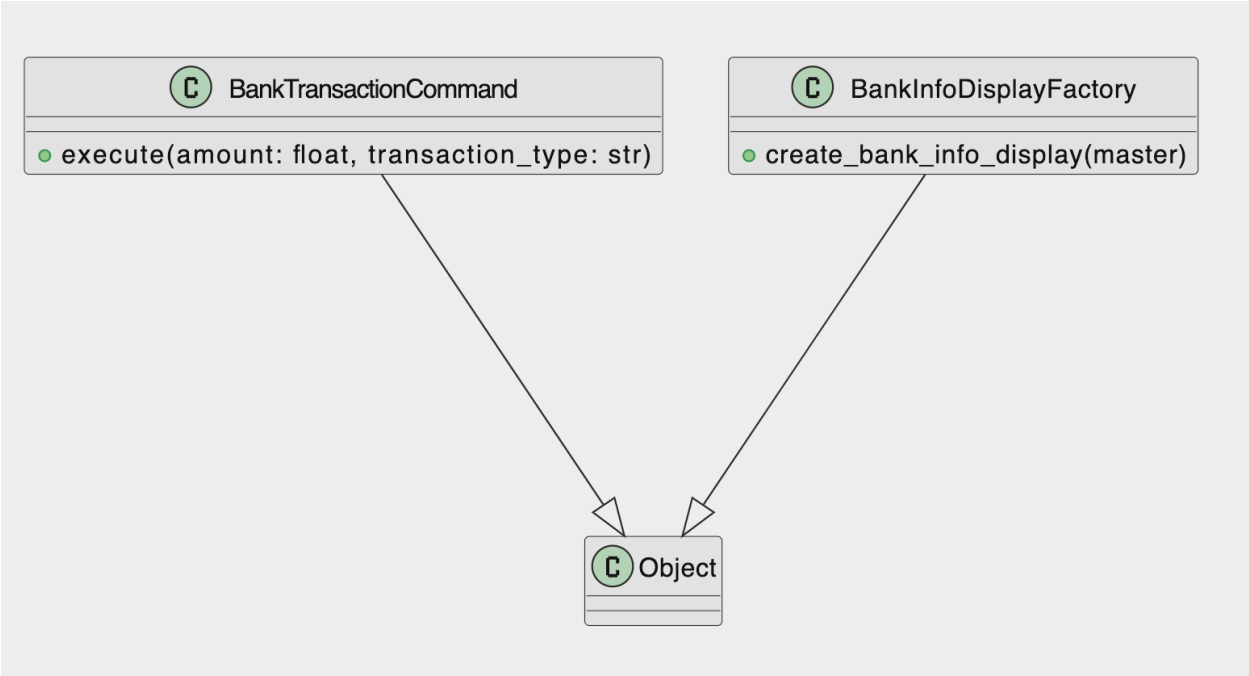
```
    self.balance_label.pack(pady=10)

    self.transaction_display = tk.Listbox(self, font=("Helvetica", 12))
    self.transaction_display.pack(pady=20)

  def update(self):
    account = BankAccount()._instance
    self.balance_label.config(text=f"Balance: ${account.get_balance():.2f}")
        self.transaction_display.delete(0, tk.END)
        transactions = account.get_transactions()
        for transaction in transactions:
            self.transaction_display.insert(tk.END, transaction)

# Command Design Pattern
class BankTransactionCommand:
    def execute(self, amount, transaction_type):
        account = BankAccount()._instance
        if transaction_type == "Deposit":
      account.deposit(amount)
    elif transaction_type == "Withdraw":
      account.withdraw(amount)

# Factory Design Pattern
class BankInfoDisplayFactory:
  def create_bank_info_display(self, master):
    return BankInfoDisplay(master)

# Application
class NetBankingApp(tk.Tk):
  def __init__(self):
    super().__init__()
    self.title("Net Banking System")
    self.geometry("1280x720")

    self.bank_account = BankAccount()._instance
    self.transaction_command = BankTransactionCommand()
    self.bank_info_display_factory = BankInfoDisplayFactory()

    # Entry for transaction amount
    self.amount_entry = tk.Entry(self, font=("Helvetica", 12))
    self.amount_entry.pack(pady=10)

    # Buttons for deposit and withdrawal
    self.deposit_button = tk.Button(self, text="Deposit", command=lambda:
self.perform_transaction("Deposit"))
```

```
    self.deposit_button.pack(side=tk.LEFT, padx=10)
    self.withdraw_button = tk.Button(self, text="Withdraw", command=lambda:
self.perform_transaction("Withdraw"))
    self.withdraw_button.pack(side=tk.RIGHT, padx=10)

    # Show Balance button
    self.show_balance_button = tk.Button(self, text="Show Balance",
command=self.show_balance)
    self.show_balance_button.pack(pady=10)

    # Bank information display
    self.bank_info_display =
self.bank_info_display_factory.create_bank_info_display(self)
    self.bank_account.add_observer(self.bank_info_display)

  def perform_transaction(self, transaction_type):
    amount_text = self.amount_entry.get()
    if not amount_text.isdigit():
      messagebox.showwarning("Invalid Amount", "Please enter a valid numeric
amount.")
      return

    amount = float(amount_text)
    if amount <= 0:
      messagebox.showwarning("Invalid Amount", "Please enter a positive
amount.")
      return

    self.transaction_command.execute(amount, transaction_type)
    self.amount_entry.delete(0, tk.END) # Clear the entry after performing the
transaction

  def show_balance(self):
    messagebox.showinfo("Current Balance", f"Your current balance is
${self.bank_account.get_balance():.2f}")

if __name__ == "__main__":
  app = NetBankingApp()
  app.mainloop()
```

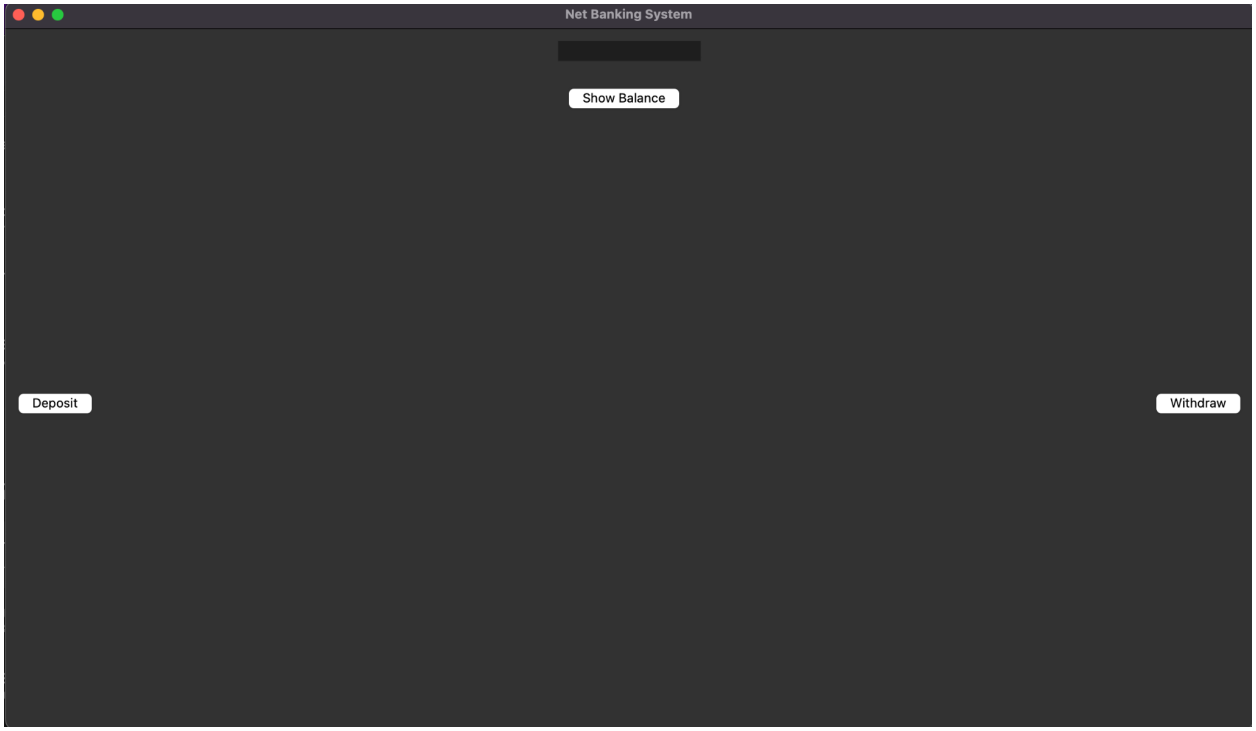
UML DIAGRAMS:



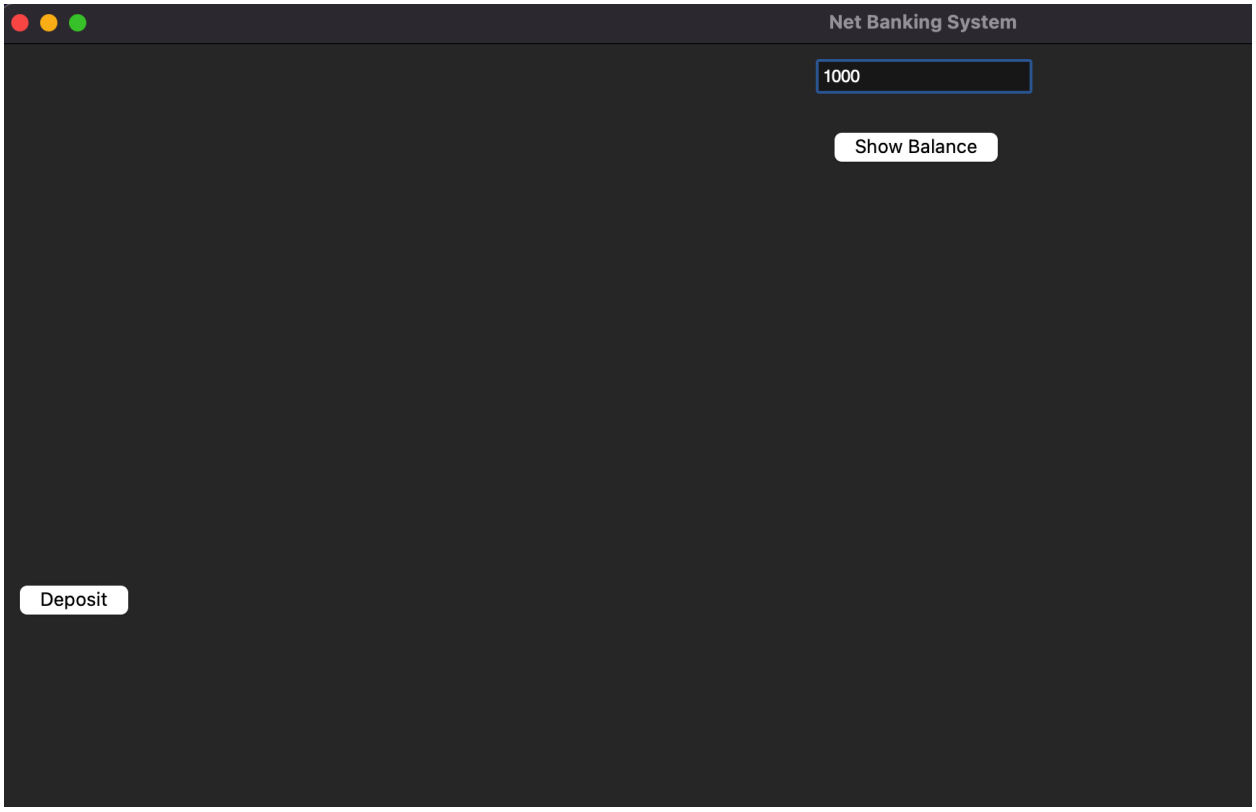

NetBankingApp
bank_account: BankAccount
transaction_command: BankTransactionCommand
bank_info_display_factory: BankInfoDisplayFactory
amount_entry: Entry
deposit_button: Button
withdraw_button: Button
show_balance_button: Button
bank_info_display: BankInfoDisplay
__init__()
perform_transaction(transaction_type: str)
show_balance()
Tk


BankTransactionCommand
execute(amount: float, transaction_type: str)
BankInfoDisplayFactory
create_bank_info_display(master)
Object

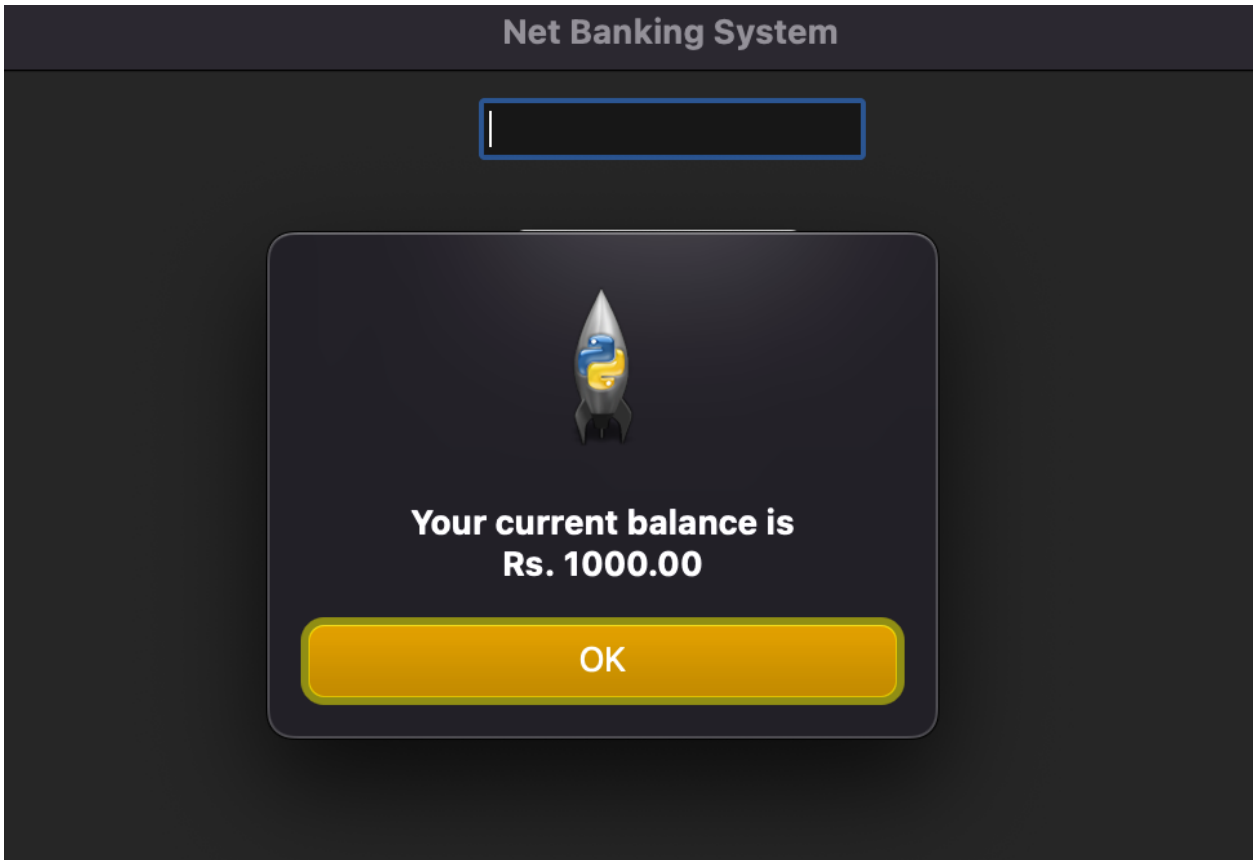
OUTPUTS:


Net Banking System
Show Balance
Deposit
Withdraw

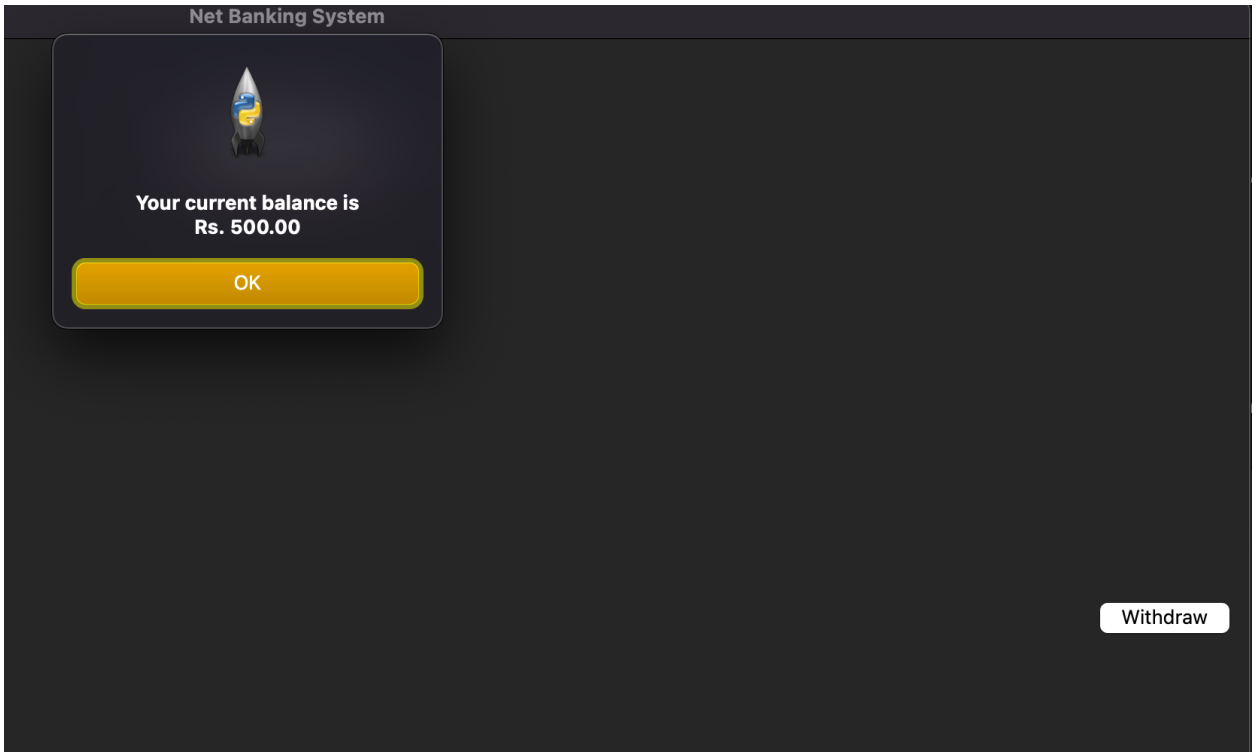
Depositing Rs. 1000



Showing Balance:



Withdrawing Rs. 500



Withdrawing More than Balance