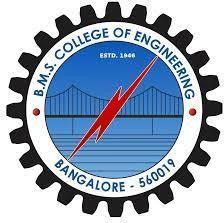
**B M S COLLEGE OF ENGINEERING**

*(An Autonomous Institution Affiliated to VTU, Belagavi)*

### Post Box No.: 1908, Bull Temple Road, Bengaluru – 560 019

**DEPARTMENT OF MACHINE LEARNING**

### Academic Year: 2023-2024 (Session: Nov 2023 - Mar 2024)



**OBJECT ORIENTED PROGRAMMING**

**(23AM5PCOOP)**

**ALTERNATIVE ASSESSMENT TOOL (AAT)**

# SIMPLE BANKING SYSTEM

### Submitted by

|  |  |  |  |
| --- | --- | --- | --- |
| Student Name: | **Harsh Vishnu** | **Harshit Mutalik** | **Hemanth Raghu** |
| USN: | **1BM21AI049** | **1BM21AI050** | **1BM21AI053** |
| Date: | **31-01-2024** | |  |
| Semester & Section: | **V-B** | | |
| Total Pages: | **24** | | |
| Student Signature: |  |  |  |

**Valuation Report (to be filled by the faculty)**

|  |  |
| --- | --- |
| Score: |  |
| Comments: |  |
| Faculty In-charge: | **Dr. Soumyalakshmi B.S** |
| Faculty Signature:  with date |  |

**Introduction To Object-Oriented Programming (OOP):**

In the realm of software development, Object-Oriented Programming (OOP) stands as a powerful paradigm that revolutionized the way we conceive, design, and implement complex systems. Rooted in the principles of abstraction, encapsulation, inheritance, and polymorphism, OOP provides a robust and modular approach to software development, fostering code reuse, scalability, and ease of maintenance.

**Key Concepts of OOP:**

Objects and Classes:

At the heart of OOP lies the concept of objects and classes. A class serves as a blueprint, defining the structure and behavior of objects. Objects, in turn, are instances of classes, embodying specific data and functionality.

Encapsulation:

Encapsulation involves bundling data and methods that operate on the data into a single unit, often referred to as a class. By encapsulating, we achieve data hiding, reducing complexity and enhancing the security and integrity of our code.

Inheritance:

Inheritance allows a class to inherit attributes and methods from another class, promoting code reuse and the creation of hierarchies. This hierarchical structure enables the modeling of relationships between entities in a more intuitive and organized manner.

Polymorphism:

Polymorphism allows objects of different classes to be treated as objects of a common base class. This flexibility simplifies code implementation and fosters the creation of more versatile and extensible systems.

Abstraction:

Abstraction involves hiding the complex implementation details while exposing only the necessary functionalities. Abstract classes and interfaces provide a blueprint for creating concrete classes, facilitating the creation of modular and scalable systems.

Composition:

Composition involves constructing classes using instances of other classes. This allows for the creation of more complex objects by combining simpler ones, promoting a modular and flexible design.

**Advantages of OOP:**

Modularity:

OOP encourages breaking down a system into smaller, manageable modules (classes), making it easier to understand, develop, and maintain.

Code Reusability:

Through inheritance and composition, OOP enables the reuse of existing code, reducing redundancy and promoting a more efficient development process.

Scalability:

OOP facilitates the creation of scalable systems by providing a structured and organized framework. New features or modifications can be added without significantly affecting existing code.

Maintainability:

OOP's modular and encapsulated nature makes code maintenance more straightforward. Changes can be isolated to specific classes, reducing the risk of unintended side effects.

Flexibility:

OOP allows for flexible and extensible designs, accommodating changes in requirements and supporting the evolution of software systems over time.

Real-world Modeling:

OOP allows developers to model real-world entities more closely, aligning software design with the structure of the problem domain. This results in a more intuitive and natural representation of complex systems, making it easier for developers to conceptualize and implement solutions.

Code Organization and Readability:

OOP promotes a clear and organized structure for code, making it more readable and understandable. The use of classes and objects aligns well with how humans naturally think about and categorize things, leading to code that is easier to navigate and maintain.

Collaboration and Team Development:

OOP facilitates collaboration among development teams. By breaking down a system into modular components, different team members can work on distinct parts of the application independently, fostering parallel development and collaboration.

Improved Software Design:

OOP encourages the creation of well-designed software architectures. Design patterns, such as Singleton, Factory, and Observer, are often employed to solve common problems, providing tested and proven solutions to recurring design challenges.

Software Reusability:

The concept of inheritance and composition in OOP promotes code reusability. Developers can leverage existing classes and components to build new functionality, saving time and effort while ensuring reliability through the reuse of well-tested code.

Enhanced Maintenance and Updates:

OOP simplifies the maintenance and updating of software systems. Modifications or enhancements can be localized to specific classes, reducing the risk of unintended side effects and making it easier to roll out updates without affecting the entire codebase.

We will delve into a case study focused on the creation of a simplified banking system. This case study will demonstrate the implementation of OOP concepts, providing insights into the structured organization of code, encapsulation of data, utilization of inheritance and polymorphism for specialization, and the importance of exception handling for robustness. Through this exploration, we aim to showcase the real-world applicability of OOP in building scalable and maintainable software systems.

**Case Study: Building a Basic Banking System**

Background:

Consider a scenario where a local bank wants to automate its operations, including managing customer accounts, processing transactions, and providing basic banking services. This case study focuses on the development of a simplified banking system using OOP principles.

1. Identify Entities:

In our banking system, key entities include customers, accounts, and transactions.

class Customer:

def \_\_init\_\_(self, customer\_id, name, address):

self.customer\_id = customer\_id

self.name = name

self.address = address

class Account:

def \_\_init\_\_(self, account\_number, customer, balance=0.0):

self.account\_number = account\_number

self.customer = customer

self.balance = balance

class Transaction:

def \_\_init\_\_(self, source\_account, destination\_account, amount, transaction\_type, transaction\_date):

self.source\_account = source\_account

self.destination\_account = destination\_account

self.amount = amount

self.transaction\_type = transaction\_type # 'deposit' or 'withdrawal'

self.transaction\_date = transaction\_date

from datetime import datetime

# Create customers

customer1 = Customer(1, "Alice", "123 Main St")

customer2 = Customer(2, "Bob", "456 Elm St")

# Create accounts for customers

account1 = Account(101, customer1, 1000.0)

account2 = Account(102, customer2, 500.0)

# Perform a transaction

transaction\_date = datetime.now()

transaction = Transaction(account1, account2, 200.0, 'deposit', transaction\_date)

# Update balances based on the transaction

if transaction.transaction\_type == 'deposit':

transaction.source\_account.balance -= transaction.amount

transaction.destination\_account.balance += transaction.amount

elif transaction.transaction\_type == 'withdrawal':

transaction.source\_account.balance += transaction.amount

transaction.destination\_account.balance -= transaction.amount

# Output account balances after transaction

print("Account balances after transaction:")

print(f"{account1.customer.name}'s balance: {account1.balance}")

print(f"{account2.customer.name}'s balance: {account2.balance}")

2. Encapsulation:

Encapsulation helps in hiding the internal details of an object. In our case, encapsulating the customer's details and account balance.

class Customer:

def \_\_init\_\_(self, customer\_id, name, address):

self.\_customer\_id = customer\_id

self.\_name = name

self.\_address = address

def get\_name(self):

return self.\_name

# Define the Account class

class Account:

def \_\_init\_\_(self, account\_number, customer, balance=0.0):

self.\_account\_number = account\_number

self.\_customer = customer

self.\_balance = balance

def get\_balance(self):

return self.\_balance

# Create a Customer instance

customer = Customer(1, "Alice", "123 Main St")

# Create an Account instance associated with the customer

account = Account(101, customer, 1000.0)

# Output

print("Customer Name:", customer.get\_name())

print("Account Balance:", account.get\_balance())

3. Inheritance:

Inheritance allows for the creation of specialized classes. In our banking system, we might have different types of accounts like savings and checking, which can inherit from a common Account class.

class SavingsAccount(Account):

def \_\_init\_\_(self, account\_number, customer, balance=0.0, interest\_rate=0.02):

super().\_\_init\_\_(account\_number, customer, balance)

self.interest\_rate = interest\_rate

4. Polymorphism:

Polymorphism allows for the use of a common interface across different classes. For example, implementing a process\_transaction() method that can handle transactions for both savings and checking accounts.

def process\_transaction(account, amount, transaction\_type):

if transaction\_type == 'deposit':

account.deposit(amount)

elif transaction\_type == 'withdrawal':

account.withdraw(amount)

5. Composition:

Composition involves building classes using instances of other classes. In our banking system, a Bank class could be composed of multiple Account objects.

class Bank:

def \_\_init\_\_(self):

self.accounts = []

def add\_account(self, account):

self.accounts.append(account)

def display\_accounts(self):

for account in self.accounts:

print(f"Account Number: {account.\_account\_number}, Balance: {account.\_balance}")

6. Exception Handling:

Implementing exception handling is crucial for a reliable banking system. For instance, handling cases where a withdrawal amount exceeds the account balance.

class InsufficientFundsError(Exception):

pass

class Account:

def \_init\_(self, account\_number, customer, balance=0.0):

self.\_account\_number = account\_number

self.\_customer = customer

self.\_balance = balance

def withdraw(self, amount):

if amount > self.\_balance:

raise InsufficientFundsError("Insufficient funds to complete the withdrawal.")

self.\_balance -= amount

7. Association:

Association represents relationships between classes, highlighting how they interact with each other. In the context of a banking system, associations could be established between classes like Customer, Account, and Transaction.

from datetime import datetime

# Define the Customer class

class Customer:

def \_\_init\_\_(self, customer\_id, name, address):

self.customer\_id = customer\_id

self.name = name

self.address = address

self.accounts = [] # Establishing association with accounts

def add\_account(self, account):

self.accounts.append(account)

# Define the Account class

class Account:

def \_\_init\_\_(self, account\_number, customer, balance=0.0):

self.account\_number = account\_number

self.customer = customer

self.balance = balance

self.transactions = [] # Establishing association with transactions

def add\_transaction(self, transaction):

self.transactions.append(transaction)

# Define the Transaction class

class Transaction:

def \_\_init\_\_(self, source\_account, destination\_account, amount, transaction\_type, transaction\_date):

self.source\_account = source\_account

self.destination\_account = destination\_account

self.amount = amount

self.transaction\_type = transaction\_type

self.transaction\_date = transaction\_date

# Create customers

customer1 = Customer(1, "Alice", "123 Main St")

customer2 = Customer(2, "Bob", "456 Elm St")

# Create accounts for customers

account1 = Account(101, customer1, 1000.0)

account2 = Account(102, customer2, 500.0)

# Add accounts to customers

customer1.add\_account(account1)

customer2.add\_account(account2)

# Perform a transaction

transaction\_date = datetime.now()

transaction = Transaction(account1, account2, 200.0, 'deposit', transaction\_date)

# Add transaction to accounts

account1.add\_transaction(transaction)

account2.add\_transaction(transaction)

# Output

print("Customer:", customer1.name)

print("Account Number:", account1.account\_number)

print("Balance:", account1.balance)

print("Transactions:")

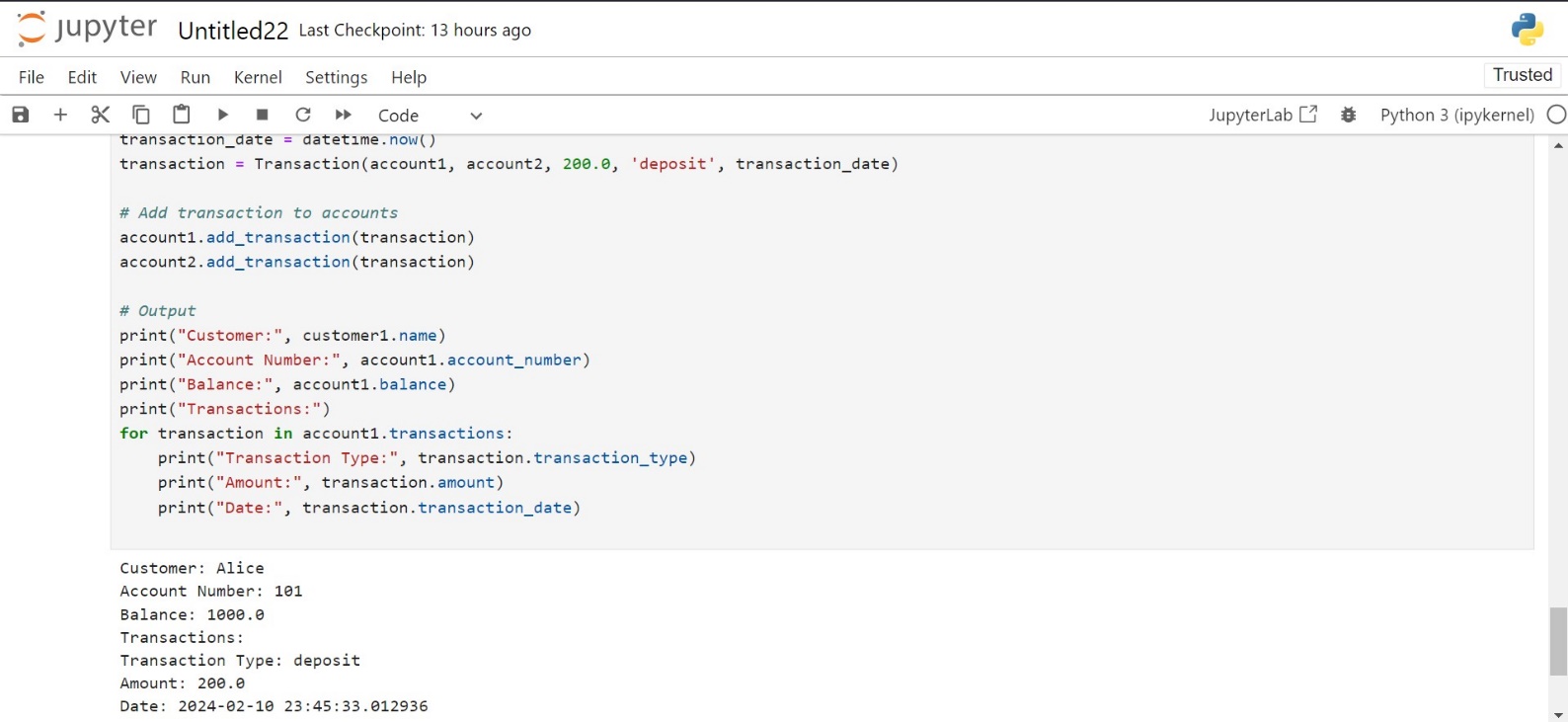
for transaction in account1.transactions:

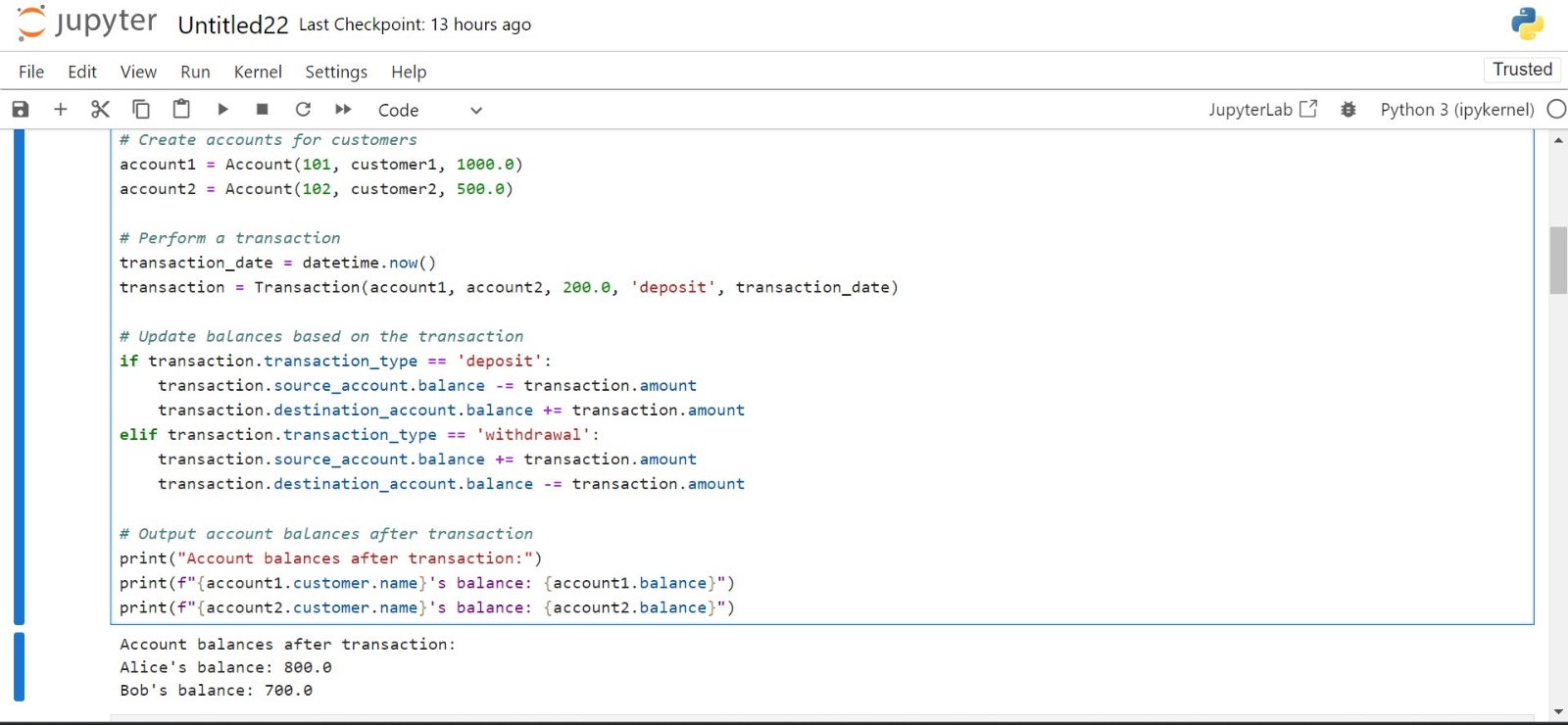
print("Transaction Type:", transaction.transaction\_type)

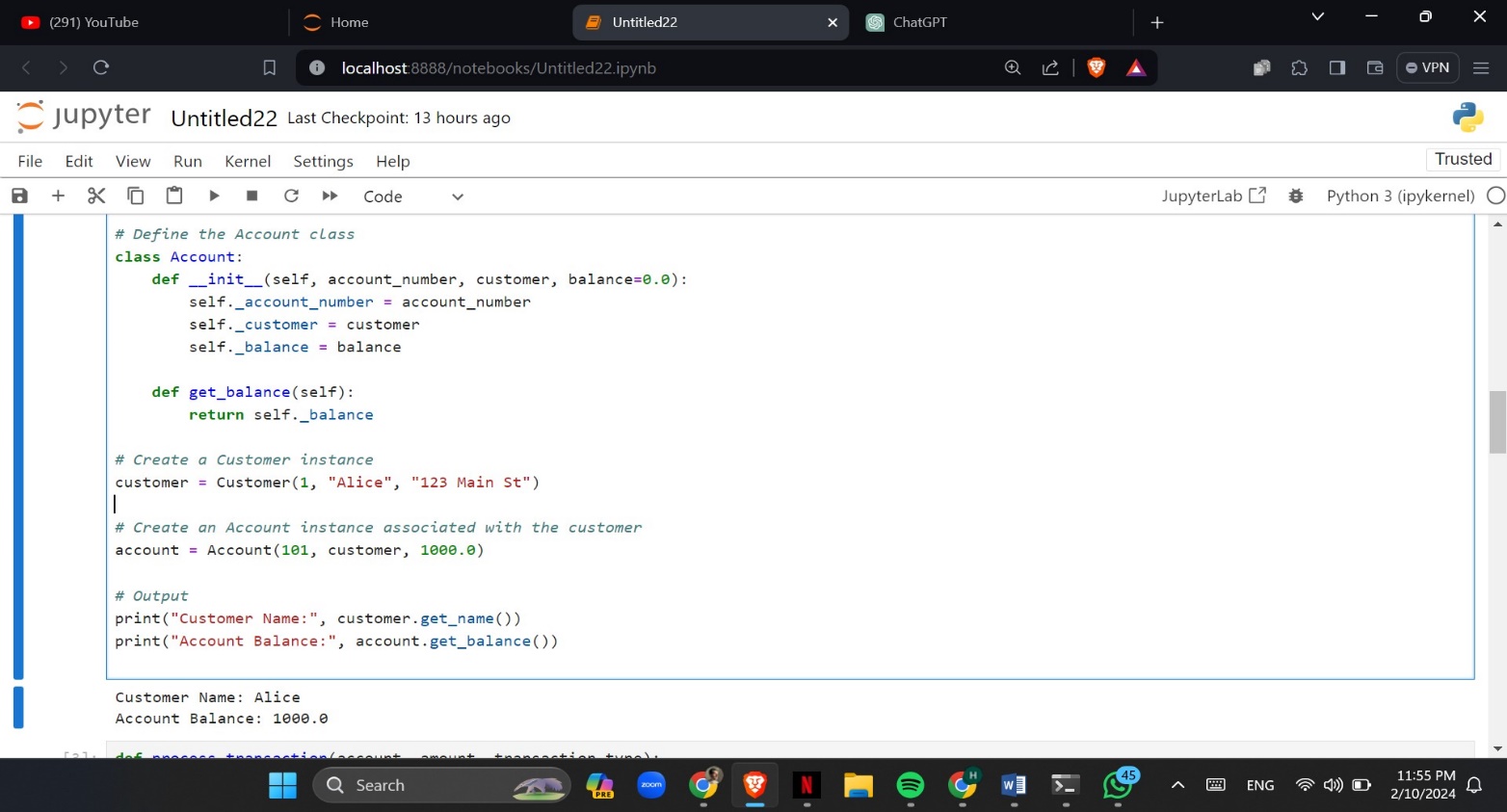
print("Amount:", transaction.amount)

print("Date:", transaction.transaction\_date)

**OUTPUT:**







**CONCLUSION:**

The adoption of Object-Oriented Programming (OOP) principles in the development of a basic banking system has proven to be instrumental in creating a structured, modular, and scalable solution. By employing encapsulation, inheritance, polymorphism, association, and abstraction, the banking system has evolved into a cohesive and adaptable framework.

OOP's emphasis on classes and objects facilitated the representation of real-world entities, enhancing the system's alignment with the problem domain. Encapsulation heightened data security, while inheritance allowed for the creation of specialized account types, promoting code reuse.

Associations captured vital relationships between classes, showcasing the significance of interconnected entities. Polymorphism in transaction processing illustrated the flexibility derived from a shared interface among different transaction types.

Abstraction provided a standardized interface for account operations, contributing to code maintainability. Collectively, these OOP concepts not only improved the banking system's design but also established a robust foundation for future enhancements.

In summary, the case study underscores the effectiveness of OOP in crafting resilient and adaptable software systems. The principles applied have resulted in a banking system that is not only technically sound but also aligns seamlessly with real-world problem-solving, emphasizing the enduring value of OOP in software development.