Finance Management System

CASE STUDY

-------HEXAWARE TRAINING -------

-

PYTHON-BATCH 4

KEERTHIKA C

OVERVIEW

The **Finance Management System** is a Python-based application built to simplify and streamline personal expense tracking and management. It allows users to securely manage their financial data, including adding, viewing, updating, and deleting expenses while categorizing them for better analysis. Users can also generate reports to gain insights into their spending habits over specific periods.

Developed with strong adherence to **object-oriented principles**, the system ensures a clean and modular design, promoting scalability and maintainability. The backend leverages **MySQL** for persistent data storage, ensuring data integrity and fast access to user and expense records.

The application is structured into clearly defined packages for:

- **Entities** (to model users, expenses, and categories),
- **Data Access Objects (DAO)** for database interaction,
- **Utilities** for connection handling,
- **Exception Handling** for robust and user-friendly error management.

INTRODUCTION

With the increasing pace of life, we consider effective personal finances management an important component for individuals to remain financially stable and make better decisions. However, tracking daily expenditure manually can be time-consuming, has the potential for error, and does not allow for any analysis of spending patterns.

The Finance Management System was created to solve this problem by providing a systematic, user-friendly, and secure way of managing personal finances. The user can undertake the basic functionality, serval methods of adding, editing, categorizing expenses, and creating reports to conserve and analyze financial trends over time.The application was created using Python and mySQL and presents various software engineering best practices, including object-oriented programming, modular programming, exception handling, and unit testing. In short, this report provides a basis for the design, implementation, and basic functionality of the system, as well as the overview of the architecture and real-world context.

SQL Schema:

Schema Design:

CREATE DATABASE:

CREATE DATABASE finance_db;

USE finance_db;

Users:

• user_id (Primary Key)

• username

• password

• email

CREATE TABLE users (user_id INT AUTO_INCREMENT PRIMARY KEY, username VARCHAR(50) NOT NULL, password VARCHAR(50) NOT NULL, email VARCHAR(100) NOT NULL);

INSERT INTO users (user_id, username, password, email) VALUES

(1, 'rekha', 'pass123', 'rekha@gmail.com'),

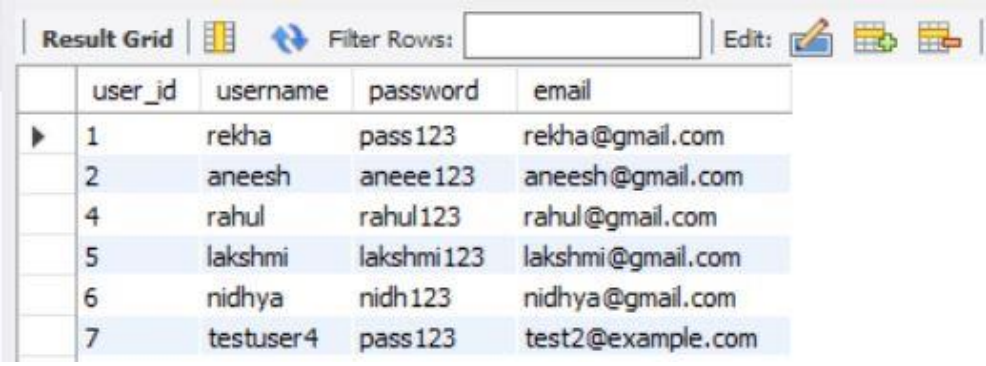
(2, 'aneesh', 'aneee123', 'aneesh@gmail.com'),

(4, 'rahul', 'rahul123', 'rahul@gmail.com'),

(5, 'lakshmi', 'lakshmi123', 'lakshmi@gmail.com'),

(6, 'nidhya', 'nidh123', 'nidhya@gmail.com'),

(7, 'testuser4', 'pass123', 'test2@example.com');



Result Grid | Filter Rows: | Edit:

user_id	username	password	email
1	rekha	pass123	rekha@gmail.com
2	aneesh	aneee123	aneesh@gmail.com
4	rahul	rahul123	rahul@gmail.com
5	lakshmi	lakshmi123	lakshmi@gmail.com
6	nidhya	nidh123	nidhya@gmail.com
7	testuser4	pass123	test2@example.com

Expenses:

• expense_id (Primary Key)

• user_id (Foreign Key referencing Users table)

• amount

• category_id (Foreign Key referencing ExpenseCategories table)

• date

• description

CREATE TABLE expenses (expense_id INT AUTO_INCREMENT PRIMARY KEY, user_id INT, amount DECIMAL(10, 2) NOT NULL, category_id INT, date DATE NOT NULL, description TEXT, FOREIGN KEY (user_id) REFERENCES Users(user_id),FOREIGN KEY(category_id) REFERENCES ExpenseCategories(category_id));

INSERT INTO expenses (expense_id, user_id, amount, category_id, date, description) VALUES (101, 1, 300.00, 1, '2025-12-12', 'movie ticket'), (102, 2, 500.00, 2, '2025-05-22', 'groceries'), (103, 3, 400.00, 3, '2025-03-05', 'medicines'), (104, 4, 100.50, 4, '2025-04-10', 'Taxi fare'), (107, 5, 100.50, 5, '2025-04-05', 'Test expense');

Result Grid | Filter Rows: | Edit: | Export/Imp

expense_id	user_id	amount	category_id	date	description
101	1	300.00	1	2025-12-12	movie ticket
102	2	500.00	2	2025-05-22	groceries
103	3	400.00	3	2025-03-05	medicines
104	4	100.50	4	2025-04-10	taxi fare
107	5	100.50	5	2025-04-05	Test expense

ExpenseCategories:

• category_id (Primary Key)

• category_name

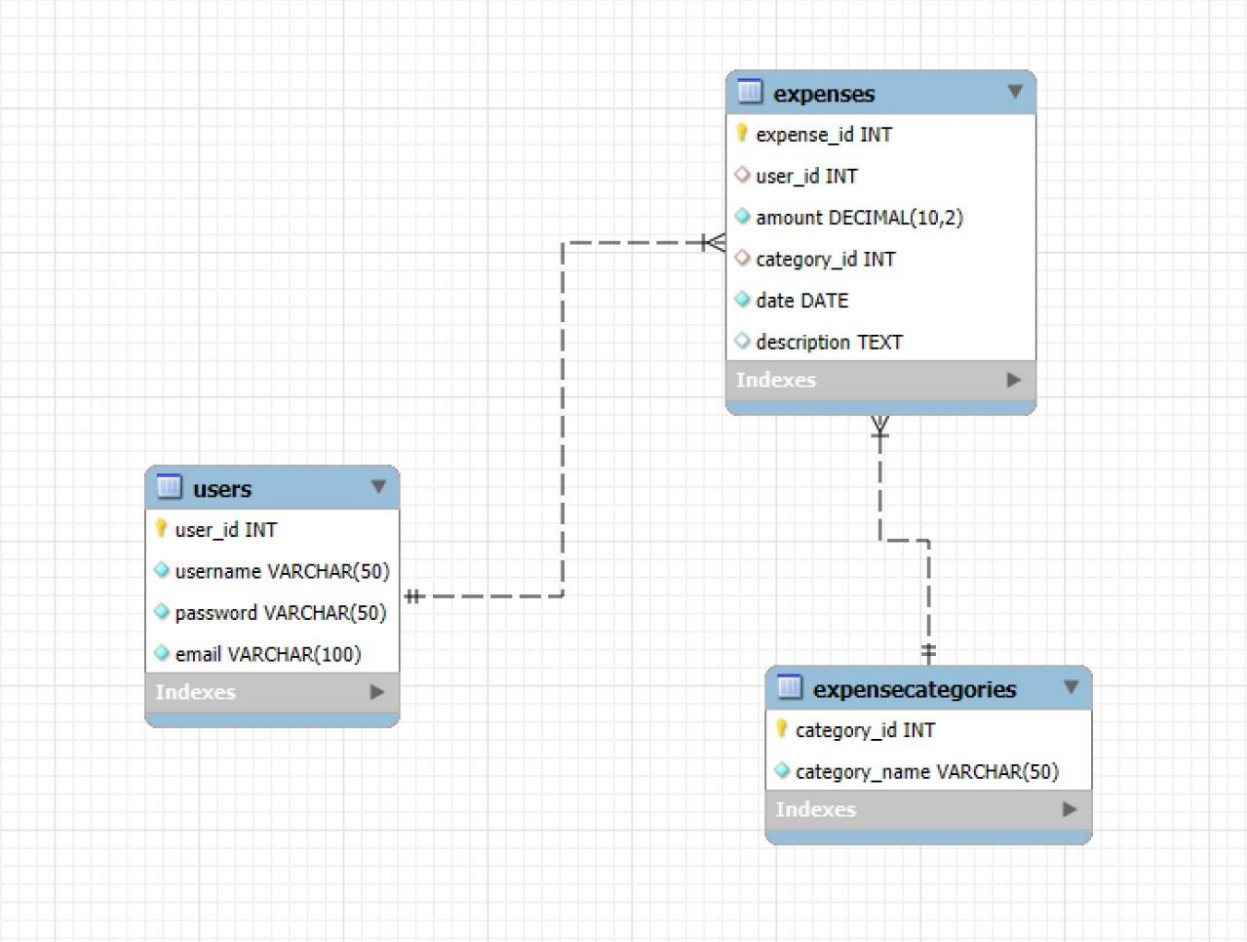
CREATE TABLE expensecategories (category_id INT AUTO_INCREMENT PRIMARY KEY, category_name VARCHAR(50) NOT NULL);

INSERT INTO expense_categories (category_id, category_name) VALUES (1, 'Food'), (2, 'Transportation'), (3, 'Utilities'), (4, 'Entertainment'), (5, 'Healthcare');

Result Grid | Filter Rows: | Edit:

category_id	category_name
1	Food
2	Transportation
3	Utilities
4	Entertainment
5	healthcare
NULL	NULL

ER DIAGRAM


expenses
expense_id INT
user_id INT
amount DECIMAL(10,2)
category_id INT
date DATE
description TEXT
Indexes
users
user_id INT
username VARCHAR(50)
password VARCHAR(50)
email VARCHAR(100)
Indexes
expensecategories
category_id INT
category_name VARCHAR(50)
Indexes

PYTHON PROGRAM

ENTITY:

The entity package contains the core model classes representing real-world objects within the Finance Management System. These include User, Expense, and Category classes. Each class encapsulates private attributes with appropriate constructors and accessor (getter/setter) methods. These classes form the foundation for mapping data between the application and the database.

Key Classes:

- User: Represents a registered user in the system.
- Expense: Represents an individual expense record, including amount, date, and category.
- Category: Represents the type or nature of an expense (e.g., Food, Healthcare).

expense.py

```
class Expense:
   def __init__(self, expense_id=None, user_id=None, amount=None,
category_id=None, date=None, description=None):
   self._expense_id = expense_id
      self._user_id = user_id
       self._amount = amount
       self._category_id = category_id
       self._date = date
       self._description = description

 def expense_id(self):
       return self._expense_id

   def expense_id(self, value):
       self._expense_id = value
```

```
    def user_id(self):
        return self._user_id

    def user_id(self, value):
        self._user_id = value

    def amount(self):
        return self._amount

    def amount(self, value):
        self._amount = value


    def category_id(self):
        return self._category_id

    def category_id(self, value):
        self._category_id = value

    def date(self):
        return self._date

    def date(self, value):
        self._date = value

    def description(self):
        return self._description

    def description(self, value):
        self._description = value
```

expense_category.py

```
class ExpenseCategory:
    def __init__(self, category_id=None, category_name=None):
        self._category_id = category_id
```

```
        self._category_name = category_name

    def category_id(self):
        return self._category_id

    def category_id(self, value):
        self._category_id = value

    def category_name(self):
        return self._category_name

    def category_name(self, value):
        self._category_name = value
```

User.py

```
Class User:
    def __init__(self, user_id=None, username=None, password=None,
email=None):
        self._user_id = user_id
        self._username = username
        self._password = password
        self._email = email

    def user_id(self):
        return self._user_id

      def user_id(self, value):
        self._user_id = value

    def username(self):
        return self._username

    def username(self, value):
        self._username = value

   def password(self):
```

```
        return self._password

    def password(self, value):
        self._password = value

    def email(self):
        return self._email


def email(self, value):
        self._email = value
```

DAO

The dao (Data Access Object) package handles all the interaction with the SQL database. It abstracts the database operations and provides methods to perform CRUD operations on users and expenses.

Interfaces and Implementations:

- IFinanceRepository: An interface that defines all required methods such as:
    - createUser(User user)
    - createExpense(Expense expense)
    - deleteUser(int userId)
    - deleteExpense(int expenseId)
    - getAllExpenses(int userId)
    - updateExpense(int userId, Expense expense)
- FinanceRepositoryImpl: Implements the above interface and executes SQL queries to perform the actual data manipulation in the MySQL database.

finance_repository_impl.py

```
import mysql.connector
from dao.ifinance_repository import IFinanceRepository
```

```
from entity.expense import Expense
from entity.user import User
from exception.myexceptions import UserNotFoundException,
ExpenseNotFoundException
from util.db_conn_util import DBConnUtilclass
FinanceRepositoryImpl(IFinanceRepository): def __init__(self):
        self.connection = DBConnUtil.get_connection()

    def create_user(self, user: User) -> bool:
        cursor = self.connection.cursor()
        query = "INSERT INTO Users (username, password, email) VALUES
(%s, %s, %s)"
        cursor.execute(query, (user.username, user.password, user.email))
        self.connection.commit()
        return True

    def create_expense(self, expense: Expense) -> bool:
        cursor = self.connection.cursor()
        query = "INSERT INTO Expenses (user_id, amount, category_id, date,
description) VALUES (%s, %s, %s, %s, %s)"
        cursor.execute(query, (expense.user_id, expense.amount,
expense.category_id, expense.date, expense.description))
        self.connection.commit()
        return True

    def delete_user(self, user_id: int) -> bool:
        cursor = self.connection.cursor()
        cursor.execute("SELECT * FROM Users WHERE user_id = %s", (user_id,))
        if not cursor.fetchone():
            raise UserNotFoundException(f"User with ID {user_id} not found")
        cursor.execute("DELETE FROM Expenses WHERE user_id = %s",
(user_id,))
        cursor.execute("DELETE FROM Users WHERE user_id = %s", (user_id,))
        self.connection.commit()
        return True
```

```
    def delete_expense(self, expense_id: int) -> bool:
        cursor = self.connection.cursor()
        cursor.execute("SELECT * FROM Expenses WHERE expense_id = %s",
(expense_id,))
        if not cursor.fetchone():
            raise ExpenseNotFoundException(f"Expense with ID {expense_id} not
found")
        cursor.execute("DELETE FROM Expenses WHERE expense_id = %s",
(expense_id,))
        self.connection.commit()
        return True

    def get_all_expenses(self, user_id: int) -> list:
        cursor = self.connection.cursor()
        cursor.execute("SELECT * FROM Expenses WHERE user_id = %s",
(user_id,))
        rows = cursor.fetchall()
        if not rows:
            raise UserNotFoundException(f"No expenses found for user ID {user_id}")
        expenses = []
        for row in rows:
            expense = Expense(row[0], row[1], row[2], row[3], row[4], row[5])
            expenses.append(expense)
        return expenses

    def update_expense(self, user_id: int, expense: Expense) -> bool:
        cursor = self.connection.cursor()
        cursor.execute("SELECT * FROM Expenses WHERE expense_id = %s AND
user_id = %s", (expense.expense_id, user_id))
        if not cursor.fetchone():
            raise ExpenseNotFoundException(f"Expense with ID
{expense.expense_id} not found for user {user_id}")
        query = "UPDATE Expenses SET amount = %s, category_id = %s, date
= %s, description = %s WHERE expense_id = %s"
        cursor.execute(query, (expense.amount, expense.category_id, expense.date,
expense.description, expense.expense_id))
```

```
        self.connection.commit()
        return True
```

ifinance_repository.py

```
from abc import ABC, abstractmethod
from entity.expense import Expense
from entity.user import User

class IFinanceRepository(ABC):

   def create_user(self, user: User) -> bool:

pass

    def create_expense(self, expense: Expense) -> bool:
        pass

    def delete_user(self, user_id: int) -> bool:
        pass

    def delete_expense(self, expense_id: int) -> bool:
        pass

    def get_all_expenses(self, user_id: int) -> list:
        pass

    def update_expense(self, user_id: int, expense: Expense) -> bool:
        pass
```

EXCEPTION

The exception package includes custom exception classes to handle specific business rule violations and ensure robust error management. These exceptions help

differentiate application-specific errors from generic system errors, thereby improving debugging and user feedback.

Custom Exceptions:

- UserNotFoundException: Thrown when a user ID does not exist in the database.
- ExpenseNotFoundException: Thrown when an expense ID is not found or invalid.

myexceptions.py

```
class UserNotFoundException(Exception):
    pass

class ExpenseNotFoundException(Exception):
    pass
```

MAIN

The main package contains the entry point for the Finance Management System application. It is responsible for driving the application by providing a user interface (usually command-line) and orchestrating interactions between different components, including entities, service providers, and the database.

Key Classes:

- **MainModule:**
    - This class contains the main method, which acts as the entry point of the application. It displays a menu of options for users to interact with the system and allows them to choose the operations they wish to perform.
    - **Operations Available:**
        - Add a new user.
        - Add a new expense.
        - Delete an existing user.
        - Delete an existing expense.

- Update an expense record.
- View all expenses for a specific user.

- The user selects an operation, and the corresponding method from the service layer (FinanceRepositoryImpl) is called to interact with the database and perform the required operation. The results are then displayed to the user.

finance_app.py

```
from dao.finance_repository_impl import FinanceRepositoryImpl
from entity.user import User
from entity.expense import Expense
from exception.myexceptions import
UserNotFoundException,ExpenseNotFoundExceptionclass FinanceApp:
def __init__(self):
      self.repo = FinanceRepositoryImpl()

   def menu(self):
      while True:
         print("\n=== Finance Management System ===")
         print("1. Add User")
         print("2. Add Expense")
         print("3. Delete User")
         print("4. Delete Expense")
         print("5. Update Expense")
         print("6. View All Expenses")
         print("7. Exit")
         choice = input("Enter your choice: ")

         try:
            if choice == "1":
               username = input("Enter username: ")
               password = input("Enter password: ")
               email = input("Enter email: ")
               user = User(username=username, password=password, email=email)
```

```
                if self.repo.create_user(user):
                    cursor = self.repo.connection.cursor()
                    cursor.execute("SELECT LAST_INSERT_ID()")
                    user_id = cursor.fetchone()[0]
                    print(f"User added successfully! User ID: {user_id}")

            elif choice == "2":
                user_id = int(input("Enter user ID: "))
                amount = float(input("Enter amount: "))
                category_id = int(input("Enter category ID: "))
                date = input("Enter date (YYYY-MM-DD): ")
                description = input("Enter description: ")
                expense = Expense(user_id=user_id, amount=amount,
category_id=category_id, date=date, description=description)
                if self.repo.create_expense(expense):
                    print("Expense added successfully!")

            elif choice == "3":
                user_id = int(input("Enter user ID to delete: "))
                if self.repo.delete_user(user_id):
                    print("User deleted successfully!")

            elif choice == "4":
                expense_id = int(input("Enter expense ID to delete: "))
                if self.repo.delete_expense(expense_id):
                    print("Expense deleted successfully!")

            elif choice == "5":
                user_id = int(input("Enter user ID: "))
                expense_id = int(input("Enter expense ID to update: "))
                amount = float(input("Enter new amount: "))
                category_id = int(input("Enter new category ID: "))
                date = input("Enter new date (YYYY-MM-DD): ")
                description = input("Enter new description: ")
                expense = Expense(expense_id=expense_id, user_id=user_id,
amount=amount, category_id=category_id, date=date, description=description)
```

```
                if self.repo.update_expense(user_id, expense):
                    print("Expense updated successfully!")

            elif choice == "6":
                user_id = int(input("Enter user ID to view expenses: "))
                expenses = self.repo.get_all_expenses(user_id)
                for exp in expenses:
                    print(f"ID: {exp.expense_id}, Amount: {exp.amount}, Date:
{exp.date}, Description: {exp.description}")

            elif choice == "7":
                print("Exiting...")
                break

            else:
                print("Invalid choice!")

        except UserNotFoundException as e:
            print(f"Error: {e}")
        except ExpenseNotFoundException as e:
            print(f"Error: {e}")
        except Exception as e:
            print(f"An error occurred: {e}")

if __name__ == "__main__":
    app = FinanceApp()
    app.menu()
```

UTIL

The util package contains utility classes to handle cross-cutting concerns like database connectivity and property management. These utilities centralize common logic for reuse and easier maintenance.

db_conn_util.py

```
import mysql.connector
from util.db_property_util import DBPropertyUtil

class DBConnUtil:
   connection = None

      def get_connection():
      if DBConnUtil.connection is None:
         #conn_string =
DBPropertyUtil.get_property_string("C:/Users/Admin/Desktop/FinanceManagemn
tSystem/db.properties")
         DBConnUtil.connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="NewPassword",
            database="finance_db",
            port=3306
         )
      return DBConnUtil.connection
```

db_property_util.py

```
import configparser


class DBPropertyUtil:

   def get_property_string(filename: str) -> str:
      config = configparser.ConfigParser()
```

```
    config.read(filename)
    db_config = config['DATABASE']
    return
f"mysql+mysqlconnector://{db_config['username']}:{db_config['password']}@{db
_config['hostname']}:{db_config['port']}/{db_config['dbname']}"
```

UNIT TESTING

Unit testing ensures the correctness of individual components in the Finance Management System. It helps verify that each method and class behaves as expected. Here's an overview of key unit testing concepts:

- **Test Cases:** Validate specific functionalities like adding users, creating expenses, or handling exceptions.
- **Assertions:** Used to compare actual outcomes with expected results.
- **Mocking:** Simulates external dependencies (e.g., database) to isolate and test components.
- **Test Coverage:** Ensures all code paths are tested for reliability.

Test Cases:

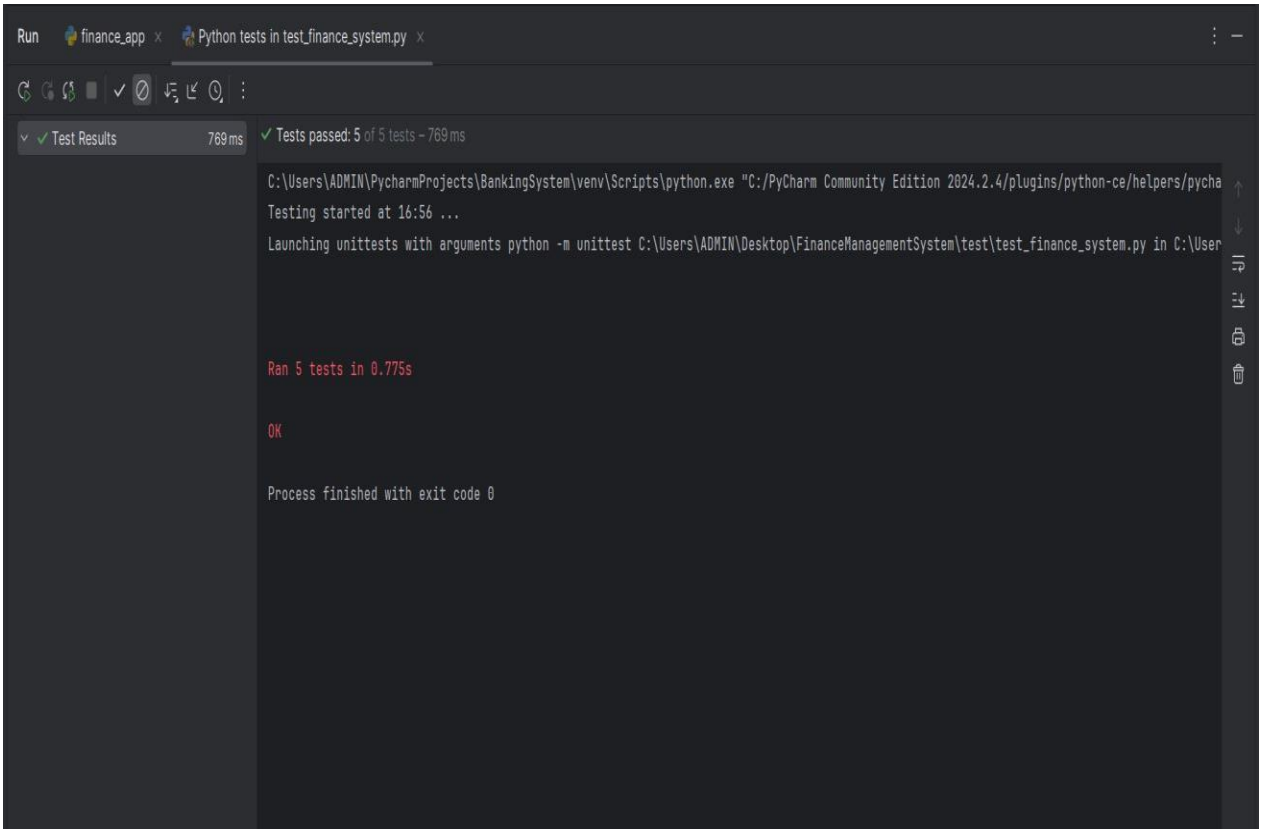
1. **Test User Creation:** Verifies that a new user is successfully added to the database.
2. **Test Expense Creation:** Ensures an expense is correctly inserted into the system.
3. **Test Expense Search:** Validates retrieving all expenses for a specific user.
4. **ExceptionHandling:**Confirmscustom exceptions (UserNotFoundException, ExpenseNotFoundException) are thrown when needed.
5. **Test Expense Update/Deletion:** Ensures expenses can be updated or removed from the database.

TEST

Test_finance_system.py

```
import unittest
from dao.finance_repository_impl import FinanceRepositoryImpl
from entity.user import User
from entity.expense import Expense
from exception.myexceptions import UserNotFoundException,
ExpenseNotFoundException
class TestFinanceSystem(unittest.TestCase):
 def setUp(self):
        self.repo = FinanceRepositoryImpl()

    def test_create_user(self):
        user = User(username="testuser3", password="pass123",
email="test@example.com")
        self.assertTrue(self.repo.create_user(user))

    def test_create_expense(self):
        user = User(username="testuser4", password="pass123",
email="test2@example.com")
        self.repo.create_user(user)
        expense = Expense(user_id=1, amount=100.50, category_id=1, date="2025-
04-05", description="Test expense")
        self.assertTrue(self.repo.create_expense(expense))

    def test_get_all_expenses(self):
        expenses = self.repo.get_all_expenses(1)
        self.assertGreater(len(expenses), 0)

    def test_user_not_found_exception(self):
        with self.assertRaises(UserNotFoundException):
            self.repo.get_all_expenses(999)

    def test_expense_not_found_exception(self):
        with self.assertRaises(ExpenseNotFoundException):
```

```
        self.repo.delete_expense(999)

if __name__ == "__main__":
unittest.main()
```



OUTPUT

1.ADD USER

```
=== Finance Management System ===
1. Add User
2. Add Expense
3. Delete User
4. Delete Expense
5. Update Expense
6. View All Expenses
7. Exit
Enter your choice: 1
Enter username: keerthi
Enter password: keerthika
Enter email: keerthi@gmail.com
User added successfully! User ID: 3
```

2.ADD EXPENSES

```
=== Finance Management System ===
1. Add User
2. Add Expense
3. Delete User
4. Delete Expense
5. Update Expense
6. View All Expenses
7. Exit
Enter your choice: 2
Enter user ID: 1
Enter amount: 500
Enter category ID: 3
Enter date (YYYY-MM-DD): 2025-05-22
Enter description: abc
Expense added successfully!
```

3.DELETE USER

```
=== Finance Management System ===
1. Add User
2. Add Expense
3. Delete User
4. Delete Expense
5. Update Expense
6. View All Expenses
7. Exit
Enter your choice: 3
Enter user ID to delete: 3
User deleted successfully!
```

4.DELETE EXPENSE

```
=== Finance Management System ===
1. Add User
2. Add Expense
3. Delete User
4. Delete Expense
5. Update Expense
6. View All Expenses
7. Exit
Enter your choice: 4
Enter expense ID to delete: 5
Error: Expense with ID 5 not found
```

5.UPDATE EXPENSES

```
=== Finance Management System ===
1. Add User
2. Add Expense
3. Delete User
4. Delete Expense
5. Update Expense
6. View All Expenses
7. Exit
Enter your choice: 5
Enter user ID: 1
Enter expense ID to update: 1
Enter new amount: 300
Enter new category ID: 3
Enter new date (YYYY-MM-DD): 2025-12-12
Enter new description: a
Expense updated successfully!
```

6.VIEW ALL EXPENSES

```
=== Finance Management System ===
1. Add User
2. Add Expense
3. Delete User
4. Delete Expense
5. Update Expense
6. View All Expenses
7. Exit
Enter your choice: 6
Enter user ID to view expenses: 1
ID: 1, Amount: 100.50, Date: 2025-04-05, Description: Test expense
```

7.EXIT

```
=== Finance Management System ===
1. Add User
2. Add Expense
3. Delete User
4. Delete Expense
5. Update Expense
6. View All Expenses
7. Exit
Enter your choice: 7
Exiting...
```

CONCLUSION

The Finance Management System project successfully demonstrates the application of core software development principles, including object-oriented programming (OOP), database interaction, exception handling, and unit testing. Through a structured approach, the system manages key functionalities such as user authentication, expense tracking, categorization, and reporting, providing a seamless user experience.

By using a modular design with distinct packages for entities, data access, exceptions, utilities, and the main application, the project ensures maintainability and scalability. The integration with MySQL for data persistence ensures that all user-related information and expenses are securely stored and easily retrievable.

The use of custom exceptions helps maintain robust error handling, while unit testing ensures the system’s reliability and correctness by verifying each component's functionality.

Overall, this project not only enhances understanding of various software design patterns and principles but also builds a comprehensive solution for efficient financial management, demonstrating a high level of coding expertise and a keen focus on quality and performance.