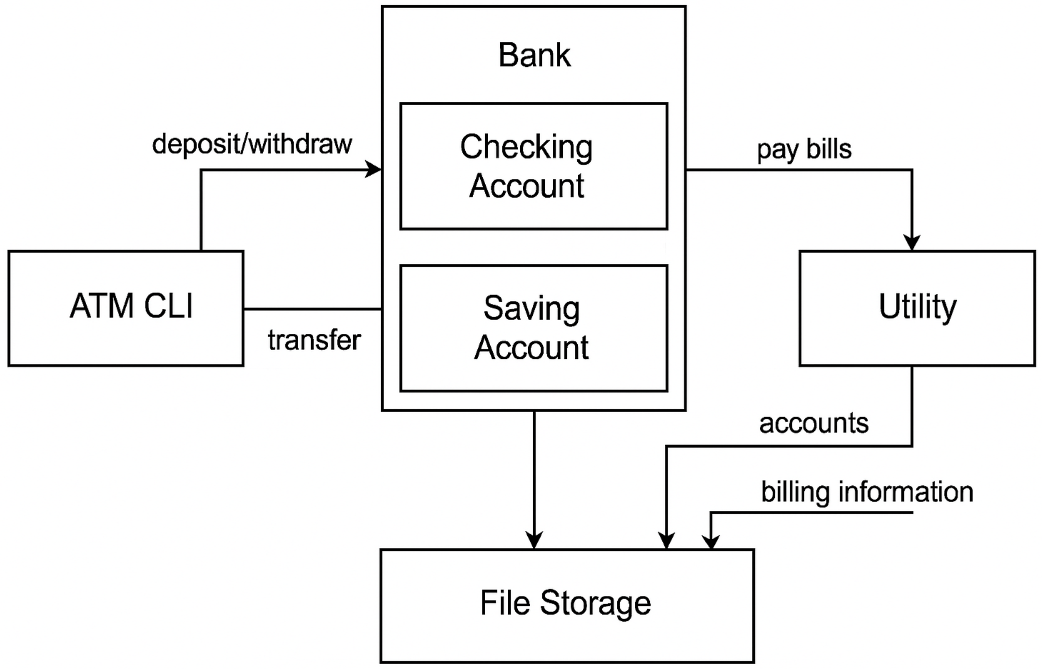
SE 3170 – Lab 6: Testing Distributed Systems

1. Schematic Diagram of the Subsystems

This diagram illustrates the architecture and interactions of the distributed ATM system. The system is composed of three/four primary subsystems:

- ATM CLI: The user interface for managing account actions via the command line.
- Bank: Consists of a Checking and a Saving account.
- Utility System: Allows users to manage bill payments, billing history, and account info.
- File Storage: All subsystems are persistently stored.

Each arrow in the diagram represents a directional flow of data or control.



2. Source Code

2a. Exception Handling and Comments

All classes and methods include JavaDoc-style comments, and exceptions are handled properly across transactions.

2b. Persistent Storage Structures

Java serialization is used to persist account and utility data. Files: checking.ser, saving.ser, utility.ser.

2c. Interaction I/O via Command Line

The system operates entirely via command-line input and output, providing a consistent CLI-based experience.

2d. Read-Me File

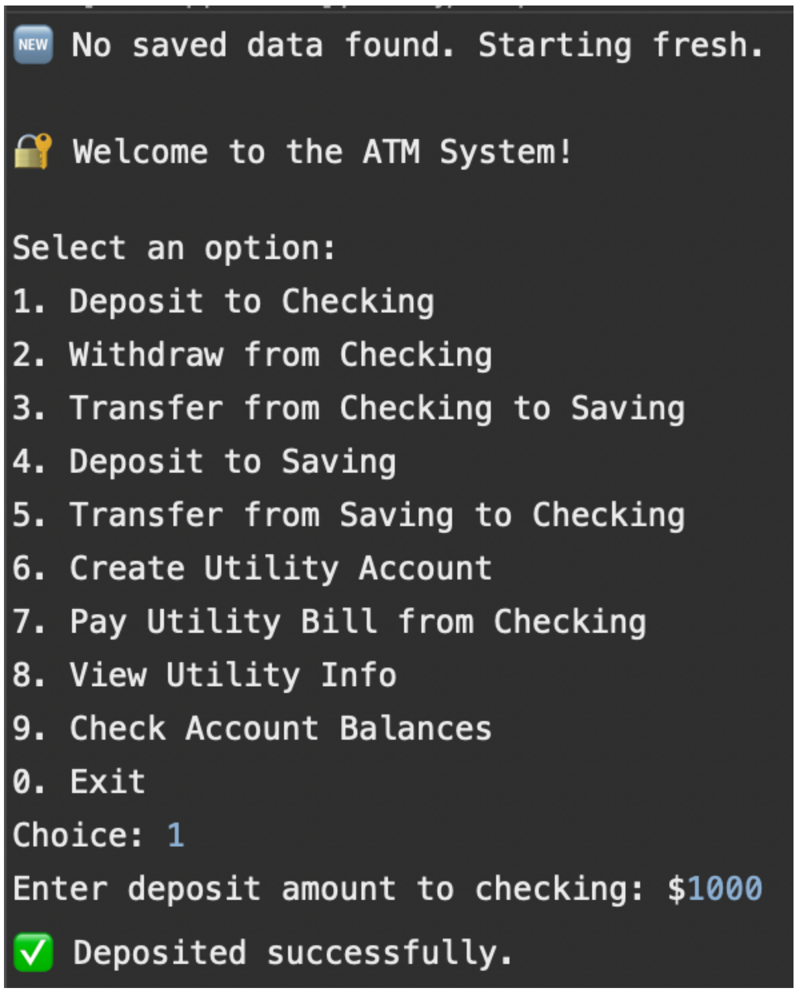
A README.md file is included with compilation, usage, and testing instructions.

2e. Custom Code Implementation

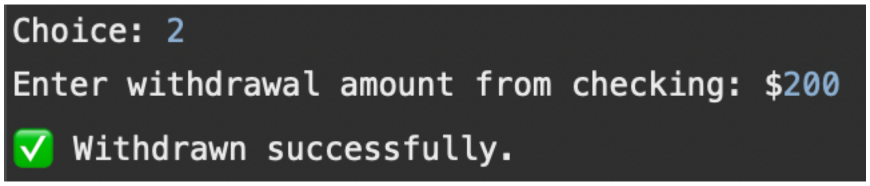
All code is written from scratch with no external libraries or templates.

3. Screenshots of User Actions and Output

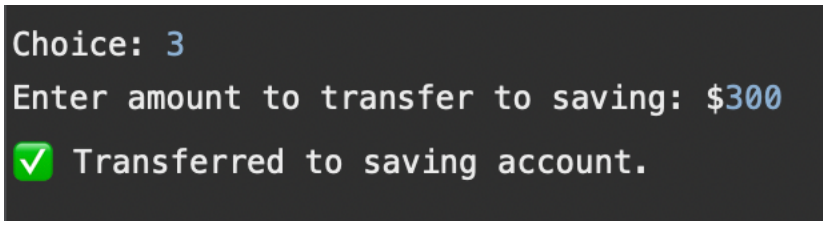
Screenshots are provided in /screenshots/, showing valid/invalid transactions and test case coverage (TR1–TR11).



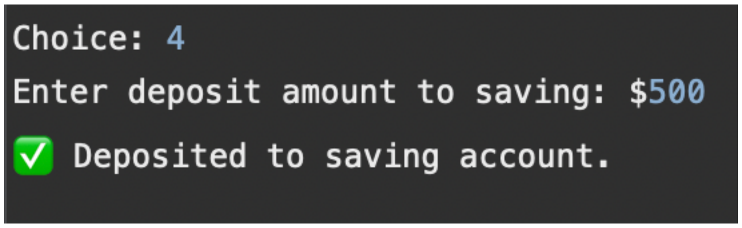
```
🆕 No saved data found. Starting fresh.

🔐 Welcome to the ATM System!

Select an option:
1. Deposit to Checking
2. Withdraw from Checking
3. Transfer from Checking to Saving
4. Deposit to Saving
5. Transfer from Saving to Checking
6. Create Utility Account
7. Pay Utility Bill from Checking
8. View Utility Info
9. Check Account Balances
0. Exit
Choice: 1
Enter deposit amount to checking: $1000
✅ Deposited successfully.
```



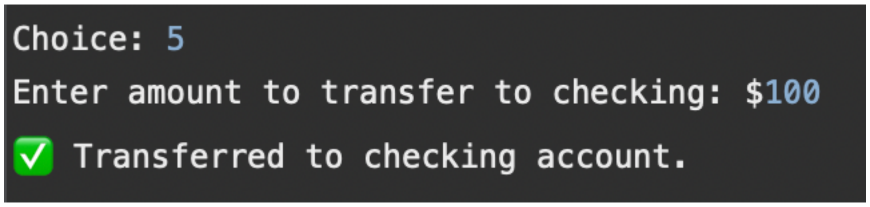
```
Choice: 2
Enter withdrawal amount from checking: $200
✅ Withdrawn successfully.
```



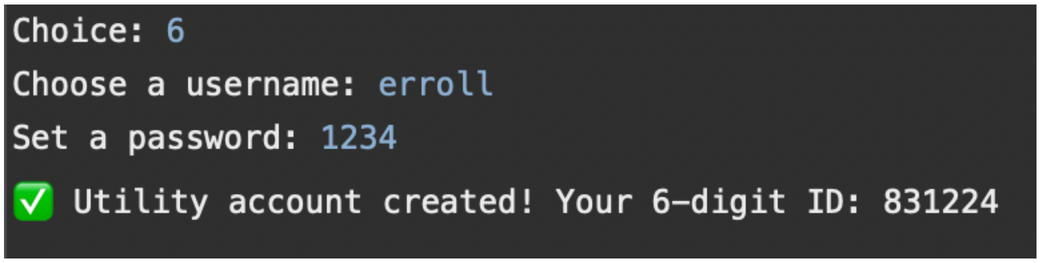
```
Choice: 3
Enter amount to transfer to saving: $300
✅ Transferred to saving account.
```



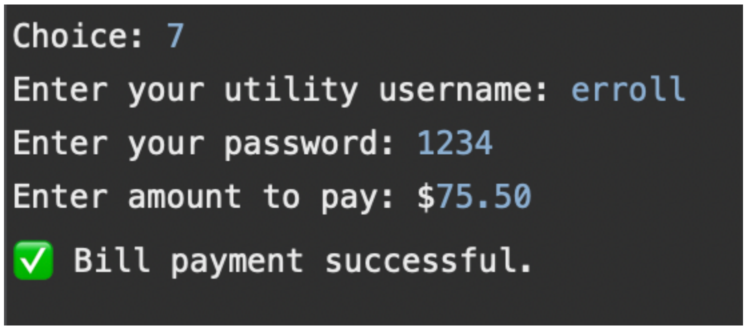
```
Choice: 4
Enter deposit amount to saving: $500
✅ Deposited to saving account.
```



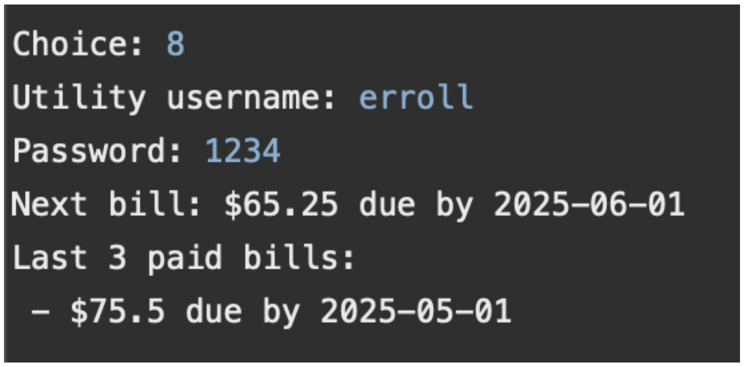
```
Choice: 5
Enter amount to transfer to checking: $100
✅ Transferred to checking account.
```



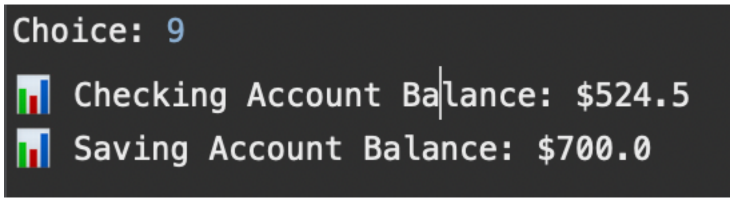
```
Choice: 6
Choose a username: erroll
Set a password: 1234
✅ Utility account created! Your 6-digit ID: 831224
```



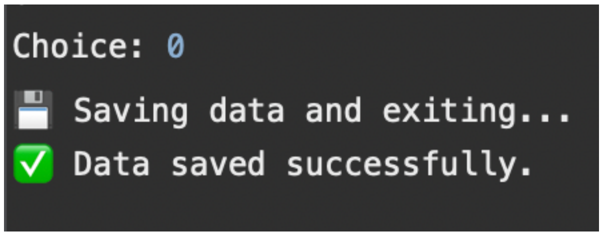
```
Choice: 7
Enter your utility username: erroll
Enter your password: 1234
Enter amount to pay: $75.50
✅ Bill payment successful.
```



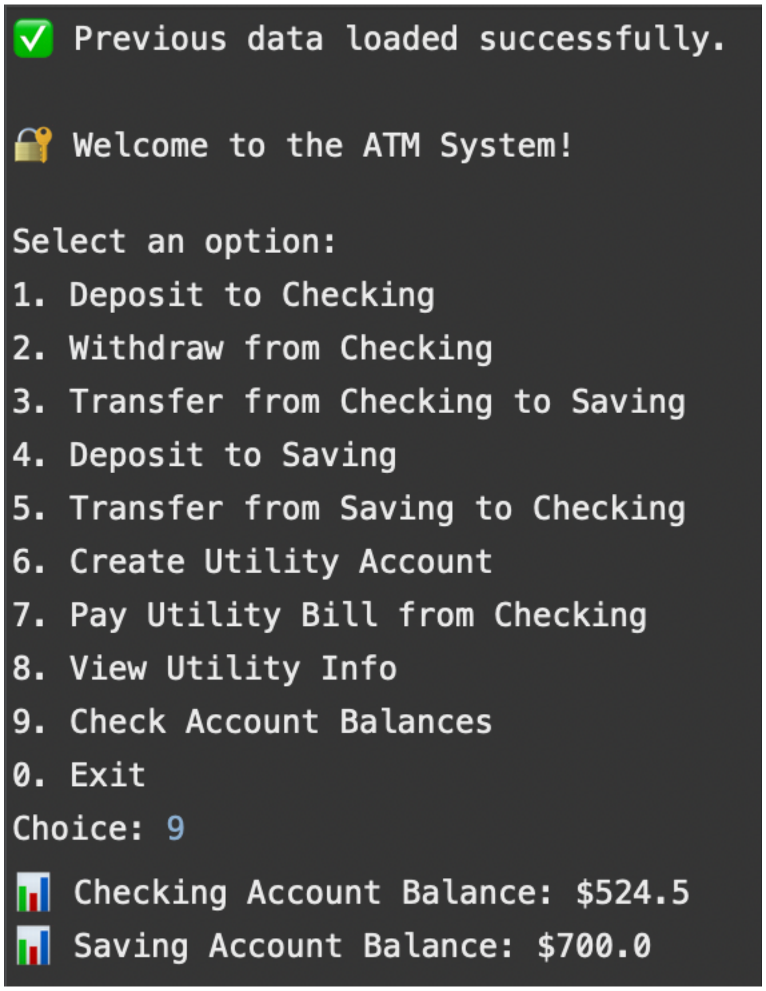
```
Choice: 8
Utility username: erroll
Password: 1234
Next bill: $65.25 due by 2025-06-01
Last 3 paid bills:
 - $75.5 due by 2025-05-01
```



```
Choice: 9
📊 Checking Account Balance: $524.5
📊 Saving Account Balance: $700.0
```



```
Choice: 0
💾 Saving data and exiting...
✅ Data saved successfully.
```



```
✅ Previous data loaded successfully.

🔐 Welcome to the ATM System!

Select an option:
1. Deposit to Checking
2. Withdraw from Checking
3. Transfer from Checking to Saving
4. Deposit to Saving
5. Transfer from Saving to Checking
6. Create Utility Account
7. Pay Utility Bill from Checking
8. View Utility Info
9. Check Account Balances
0. Exit
Choice: 9
📊 Checking Account Balance: $524.5
📊 Saving Account Balance: $700.0
```

4. Code Testing

4a. A Comprehensive Test Plan

JUnit test classes were written for checkingAcc, savingAcc, utilityComp, and dataManager. Each class tested valid and invalid inputs.

4a-i. Test Designs and Execution

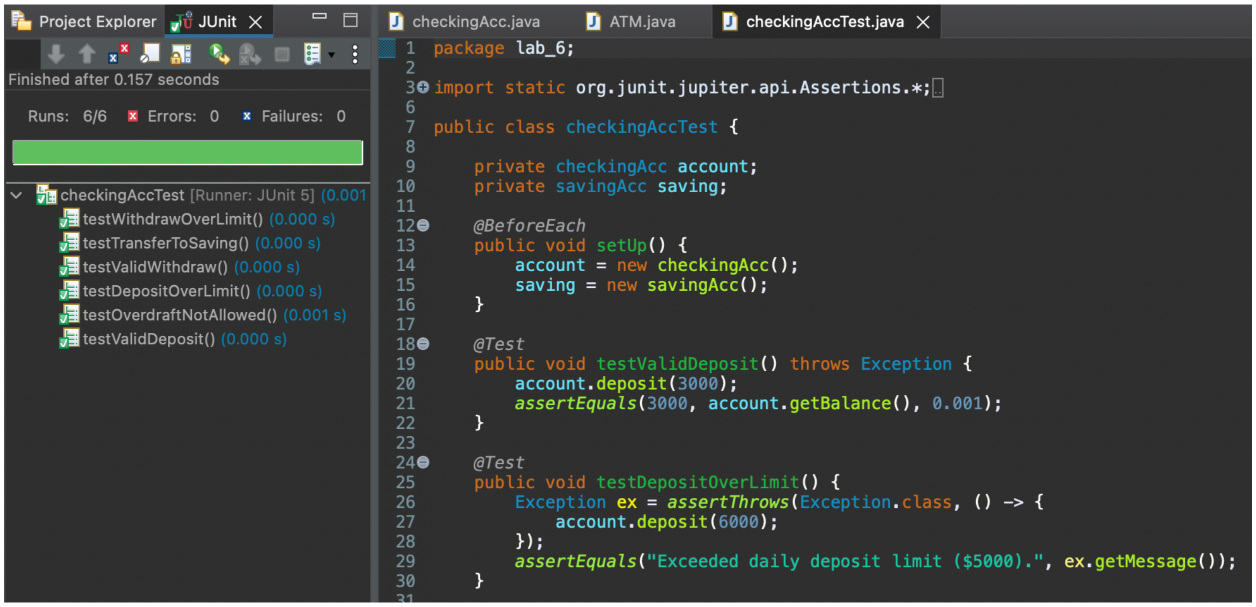
All essential functionality and boundary conditions were tested using JUnit 5.

4a-ii. Data Storage Testing

Test cases included saving/loading null objects, valid objects, incompatible objects, and file-not-found exceptions.

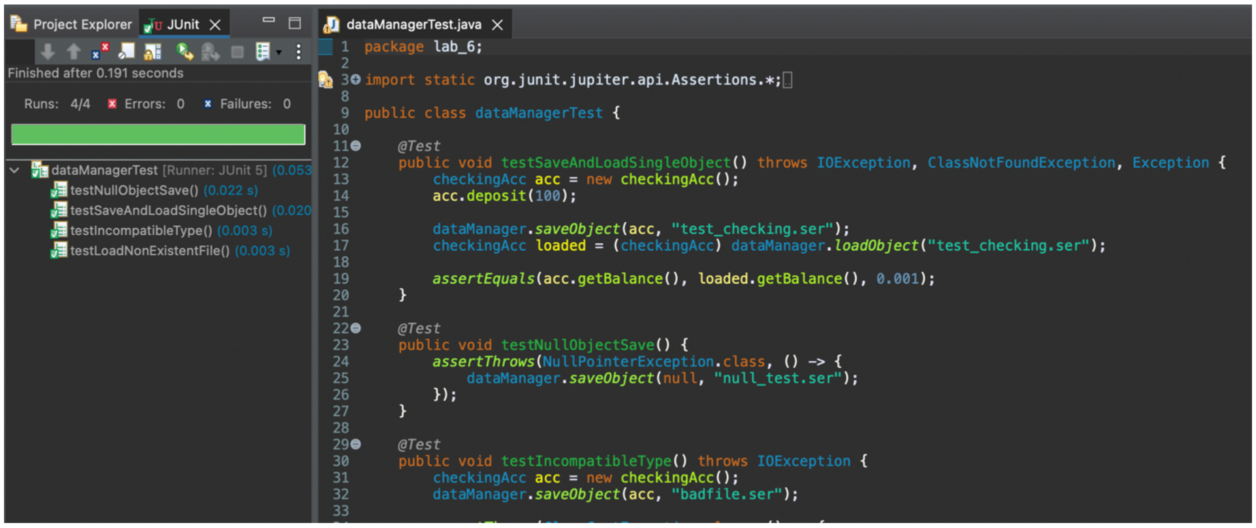
4a-iii. Test Outcomes

All tests passed successfully. Screenshots from Eclipse JUnit test confirm these results.



```
Finished after 0.157 seconds

Runs: 6/6    Errors: 0    Failures: 0

checkingAccTest [Runner: JUnit 5] (0.001
    testWithdrawOverLimit() (0.000 s)
    testTransferToSaving() (0.000 s)
    testValidWithdraw() (0.000 s)
    testDepositOverLimit() (0.000 s)
    testOverdraftNotAllowed() (0.001 s)
    testValidDeposit() (0.000 s)
```

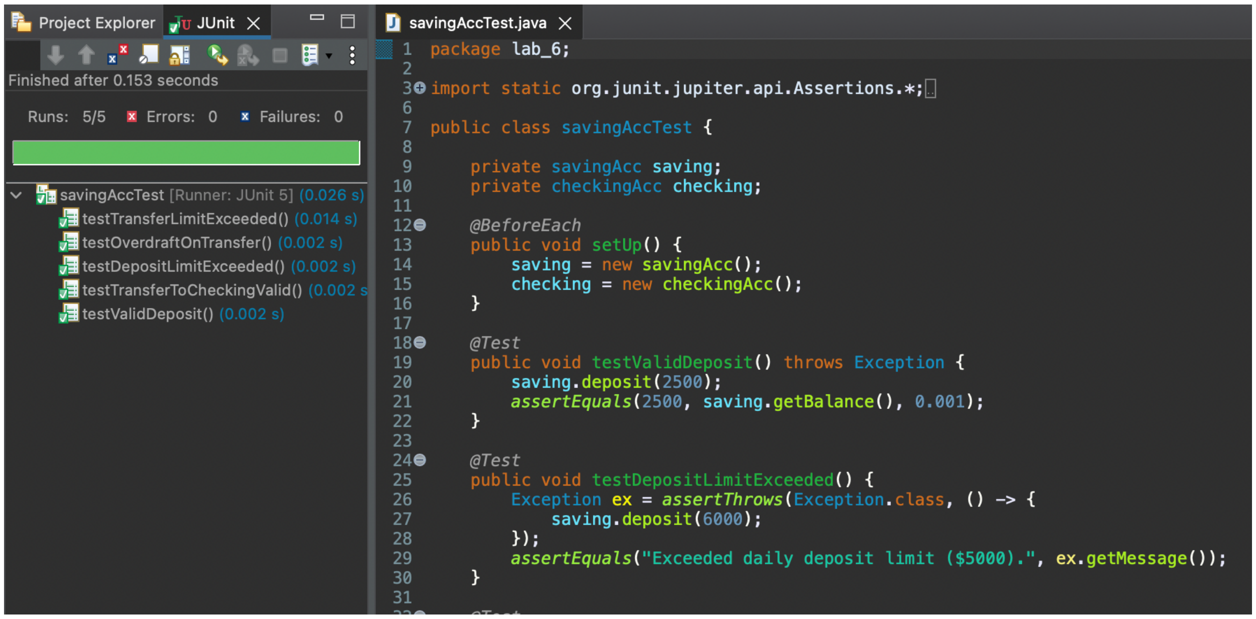
```java
package lab_6;

import static org.junit.jupiter.api.Assertions.*;

public class checkingAccTest {

    private checkingAcc account;
    private savingAcc saving;

    @BeforeEach
    public void setUp() {
        account = new checkingAcc();
        saving = new savingAcc();
    }

    @Test
    public void testValidDeposit() throws Exception {
        account.deposit(3000);
        assertEquals(3000, account.getBalance(), 0.001);
    }

    @Test
    public void testDepositOverLimit() {
        Exception ex = assertThrows(Exception.class, () -> {
            account.deposit(6000);
        });
        assertEquals("Exceeded daily deposit limit ($5000).", ex.getMessage());
    }
```



```
Project Explorer   JUnit
Finished after 0.191 seconds

Runs: 4/4    Errors: 0    Failures: 0

dataManagerTest [Runner: JUnit 5] (0.053
    testNullObjectSave() (0.022 s)
    testSaveAndLoadSingleObject() (0.020
    testIncompatibleType() (0.003 s)
    testLoadNonExistentFile() (0.003 s)
```

dataManagerTest.java

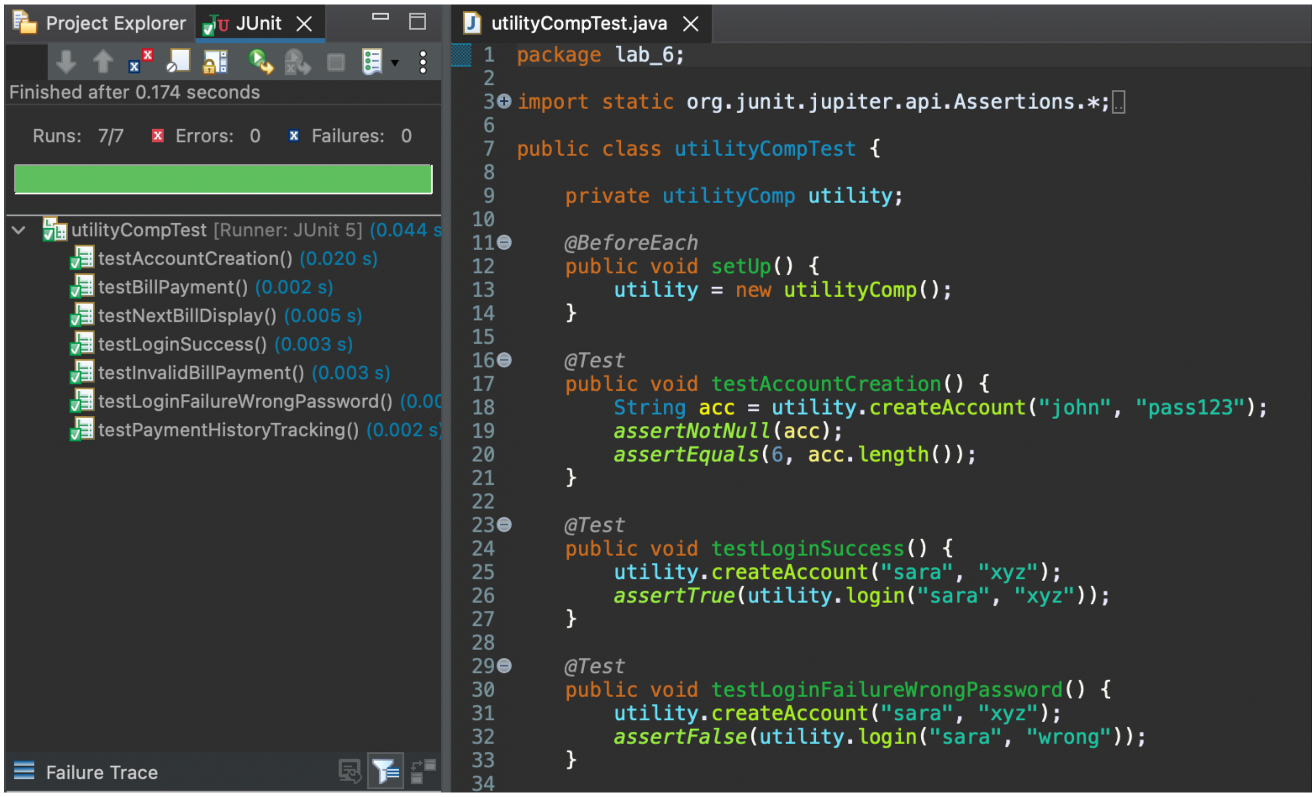
```java
package lab_6;

import static org.junit.jupiter.api.Assertions.*;

public class dataManagerTest {

    @Test
    public void testSaveAndLoadSingleObject() throws IOException, ClassNotFoundException, Exception {
        checkingAcc acc = new checkingAcc();
        acc.deposit(100);

        dataManager.saveObject(acc, "test_checking.ser");
        checkingAcc loaded = (checkingAcc) dataManager.loadObject("test_checking.ser");

        assertEquals(acc.getBalance(), loaded.getBalance(), 0.001);
    }

    @Test
    public void testNullObjectSave() {
        assertThrows(NullPointerException.class, () -> {
            dataManager.saveObject(null, "null_test.ser");
        });
    }

    @Test
    public void testIncompatibleType() throws IOException {
        checkingAcc acc = new checkingAcc();
        dataManager.saveObject(acc, "badfile.ser");
```



```
Project Explorer   JUnit
Finished after 0.153 seconds

Runs: 5/5    Errors: 0    Failures: 0

savingAccTest [Runner: JUnit 5] (0.026 s)
    testTransferLimitExceeded() (0.014 s)
    testOverdraftOnTransfer() (0.002 s)
    testDepositLimitExceeded() (0.002 s)
    testTransferToCheckingValid() (0.002 s
    testValidDeposit() (0.002 s)
```

savingAccTest.java

```java
package lab_6;

import static org.junit.jupiter.api.Assertions.*;

public class savingAccTest {

    private savingAcc saving;
    private checkingAcc checking;

    @BeforeEach
    public void setUp() {
        saving = new savingAcc();
        checking = new checkingAcc();
    }

    @Test
    public void testValidDeposit() throws Exception {
        saving.deposit(2500);
        assertEquals(2500, saving.getBalance(), 0.001);
    }

    @Test
    public void testDepositLimitExceeded() {
        Exception ex = assertThrows(Exception.class, () -> {
            saving.deposit(6000);
        });
        assertEquals("Exceeded daily deposit limit ($5000).", ex.getMessage());
    }
```



```
Finished after 0.174 seconds

Runs: 7/7    Errors: 0    Failures: 0

utilityCompTest [Runner: JUnit 5] (0.044 s
    testAccountCreation() (0.020 s)
    testBillPayment() (0.002 s)
    testNextBillDisplay() (0.005 s)
    testLoginSuccess() (0.003 s)
    testInvalidBillPayment() (0.003 s)
    testLoginFailureWrongPassword() (0.00
    testPaymentHistoryTracking() (0.002 s)
```

utilityCompTest.java

```java
package lab_6;

import static org.junit.jupiter.api.Assertions.*;

public class utilityCompTest {

    private utilityComp utility;

    @BeforeEach
    public void setUp() {
        utility = new utilityComp();
    }

    @Test
    public void testAccountCreation() {
        String acc = utility.createAccount("john", "pass123");
        assertNotNull(acc);
        assertEquals(6, acc.length());
    }

    @Test
    public void testLoginSuccess() {
        utility.createAccount("sara", "xyz");
        assertTrue(utility.login("sara", "xyz"));
    }

    @Test
    public void testLoginFailureWrongPassword() {
        utility.createAccount("sara", "xyz");
        assertFalse(utility.login("sara", "wrong"));
    }

```

4a-iv. Sufficient Transactions for Testing

Over 20 scenarios including deposits, transfers, and utility payments were tested.

5. UI Testing

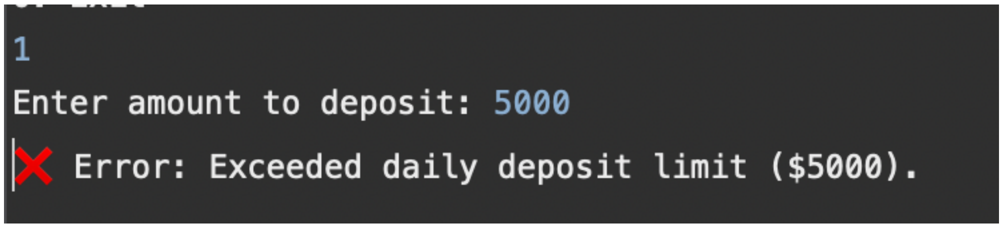
5i–ii. Methods and Criteria

- Functionality: Valid inputs perform expected actions.
- Boundary: Inputs over limits are blocked correctly.

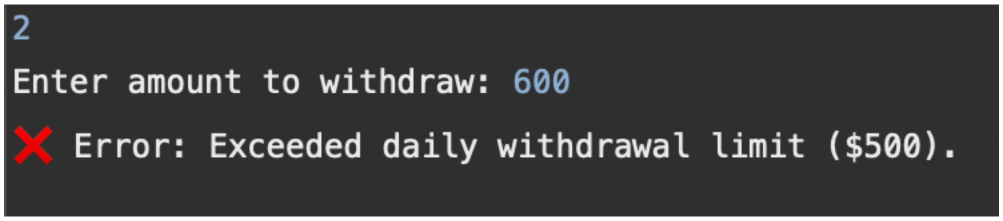
5iii. Test Requirement Sets

Functionality: TR1-TR11 (from part 3)

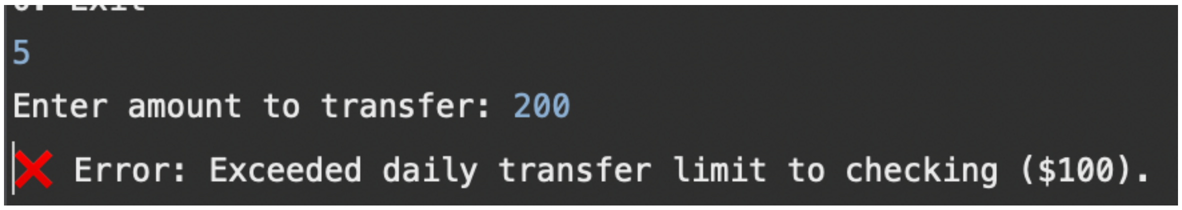
Boundary: TR12, TR13, TR14, TR15



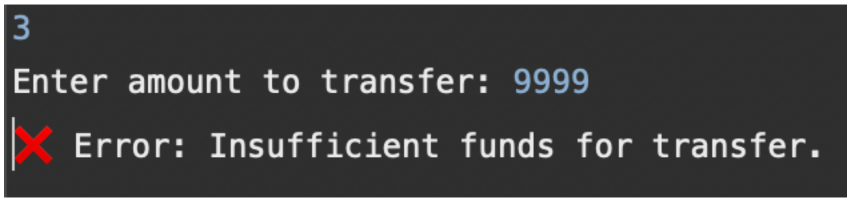
```
1
Enter amount to deposit: 5000
❌ Error: Exceeded daily deposit limit ($5000).
```



```
2
Enter amount to withdraw: 600
❌ Error: Exceeded daily withdrawal limit ($500).
```



```
5
Enter amount to transfer: 200
❌ Error: Exceeded daily transfer limit to checking ($100).
```



```
3
Enter amount to transfer: 9999
❌ Error: Insufficient funds for transfer.
```

5iv. Test Cases

All UI test cases followed a structured plan and passed. Screenshots (4a-iii) confirm successful command-line interaction.

5v–vii. Results and Analysis

All features function as expected with no unexpected behavior. UI handles all edge cases with proper messaging. All photos are in the “/screenshots” file