Lab 06 Report

By Josh Scheitler and Ethan Meyer

## 1) A schematic diagram of the three subsystems (Bank Checking, Bank Saving, and Utility

INSERT DIAGRAM

## 2) Source Code

a. Use exception handling and add appropriate comments to your code.

b. Use appropriate persistent storage structures for the bank and the utility company accounts.

c. Interaction I/O will be done using command line.

d. Add a Read-Me file of how to deploy and use your assignment.

e. Use your own software code components, do not use any existing application

## 3) Screenshots showing steps

# Part 4 – Code Testing Overview

This test plan verifies the functionality, persistence, and error handling of a distributed ATM system consisting of three core subsystems: Checking Account, Savings Account, and Utility Account. Tests are written in JUnit 4 and cover both valid and invalid scenarios across data transactions and persistence.

# Test Categories & Designs

## 1. Account Functional Tests

Test Objectives:

- Validate balance tracking, deposit/withdraw/transfer limits  
- Enforce overdraft and daily limits  
- Confirm invalid inputs are handled

|  |  |  |
| --- | --- | --- |
| Test Name | Method | Purpose |
| Account Constructor Test | testAccountConstructor() | Ensures default values (balance = 0, limits set) are initialized properly |
| Valid Deposit | testAccountDeposit() | Verifies correct balance updates after deposits |
| Valid Withdraw | testAccountWithdraw() | Confirms withdrawals reduce balance correctly |
| Valid Transfers (both ways) | testAccountTransfer() | Tests transfer logic from checking → savings and savings → checking |
| Invalid Deposits | testAccountDepositInvalid() | Tests deposit amount exceeding daily limit |
| Invalid Withdrawals | testAccountWithdrawInvalid() | Tests withdraw amount exceeding daily limit |
| Invalid Transfers | testAccountTransferInvalid() | Tests over-limit and negative transfer amounts |
| Negative Value Inputs | testAccountDepositNegative(), etc. | Prevents negative values from affecting accounts |
| Daily Limit Reset | testAccountResetDailyLimits() | Verifies correct reset of daily limits for deposits, withdrawals, and transfers |
| Limit Trackers | testAccountGetDepositLimit(), etc. | Checks updates to limits after valid transactions |
| Balance Management | testAccountGetBalance(), etc. | Validates get/set functionality with edge case handling |

## 2. User Functional Tests

Test Objectives:

- Validate retrieval of user’s linked accounts  
- Confirm file-based persistence of user accounts

|  |  |  |
| --- | --- | --- |
| Test Name | Method | Purpose |
| Account Access | testGetCheckings(), etc. | Ensures all account references are properly initialized |
| Account Save/Load | testSaveAccounts(), testLoadAccounts() | Verifies persistence of account state using storage files |

## 3. Payment & Utility Tests

Test Objectives:

- Validate payment recording and history retrieval  
- Confirm login methods  
- Test data formatting and error paths

|  |  |  |
| --- | --- | --- |
| Test Name | Method | Purpose |
| Payment Constructor | testGetPaidAmount(), etc. | Confirms object fields are returned correctly |
| Username + Account Number Login | testCorrectUserNameLogin(), etc. | Confirms both username and account number logins work |
| Login Failures | testWrongLogin() | Ensures bad credentials are rejected |
| Save Payment & History | testSavePayment(), testGetPaymentHistory() | Validates saving and retrieving of payments |
| Payment Formatting | testPaymentFormat() | Confirms format of stored payment record |
| Payment Display | testDisplayPayment(), testGetNextBillPayment() | Ensures display format is correct and reflects accurate data |
| Account Number | testGetAccountNumber() | Validates uniqueness and retrievability of the account number |

## Persistence Storage Tests

Covered Through:  
- testSaveAccounts() and testLoadAccounts() in testUser.java  
- testSavePayment() and testGetPaymentHistory() in testUtilityAccount.java  
  
Scenarios:  
- Empty file state → getPaymentHistory() returns empty list  
- Normal file entries → confirms saved data loads correctly  
- Corrupted/wrong input (invalid account number) → savePayment() returns false

## Boundary & Negative Testing

|  |  |
| --- | --- |
| Scenario | Covered In |
| Deposit > daily limit | testAccountDepositInvalid() |
| Withdraw > daily limit | testAccountWithdrawInvalid() |
| Transfer > savings limit or overdraft | testAccountTransferInvalid() |
| Negative amounts | testAccountDepositNegative(), etc. |
| Setting balance to negative | testAccountSetBalanceInvalid() |
| Login with incorrect credentials | testWrongLogin() |
| Invalid account number in payment | testSavePayment() |

a. A comprehensive teat plan of your software, including:  
i. Test designs and execution of your project functions and algorithms, storage structures,  
and user actions for handling the functionalities of their two bank accounts and the utility  
company (valid and invalid transactions shall be covered in your tests)  
ii. For data storage: Test null storage, null element with multiple elements, null single  
element, incompatible types, empty elements as well as two normal cases (single and  
multiple elements).  
iii. Test outcomes (screen shots and description)  
iv. Sufficient transactions for testing

## 5) UI Testinig

For the requirements given above, develop an experiment to evaluate the UI.  
i. Use two testing methods: functionality and boundary.  
ii. Provide one test criterion for each method.  
iii. Design a suitable Test Requirement (TR) set for each criterion.  
iv. Provide all test cases for your TR set.  
v. Run your test cases and collect results of your UI tests using the following table:  
1. Test method, test criterion, test input values, test expected output, test actual  
output, success/fail.  
vi. Take screenshots of your test results  
vii. Analyze your results and identify any unexpected behavior or wrong results.  
viii. Provide a report interpreting your test results

**1. Overview and Testing Methods**

**A. Functionality Testing**

**Purpose:** Verify that the application works correctly for a well-defined, expected sequence of user inputs.  
**Test Criterion (FC):** *“ATM will continue to function without error during defined sequence of user inputs. The UI testing will contain a set of user inputs with the expected result of each step. An unexpected result is considered a fail.”*

**B. Boundary Testing**

**Purpose:** Validate the system’s behavior when inputs are at or just beyond defined limits.  
**Test Criterion (BC):** *“ATM should enforce the daily limits for deposit, withdraw and transfer on accounts. The system should not accept negative values or overdrafts. The UI testing will contain a set of invalid inputs as well as a set of inputs close to boundaries.*

**2. Test Requirement (TR) Sets**

**TR Set for Functionality Criterion (FC):**

* **TR-F1:** The user can properly use each part of the checking account. Deposit, withdrawal, transfer, pay bill, check balance.
* **TR-F2:** The user can properly use each part of the savings account. Deposit, transfer, check balance.
* **TR-F3:** The user can properly use each part of the utility account. Log on, payment history, next bill.

**TR Set for Boundary Criterion (BC):**

* **TR-B1:** The user should be limited by daily limits when using accounts.
* **TR-B2:** The user should be unable to use negative values or overdraft accounts.
* **TR-B3:** The user should be able to use values just up to the daily limits and overdraft limit without failure.

**3. Test Cases**

**A. Functionality Test Case (FC)**

* **Test Case FC1:** *Test checking account*
  + **Step 1:**
* **Test Case FC2:** *Test savings account*
* **Test Case FC3:** *Test utility account*

**B. Boundary Test Cases (BC)**

* **Test Case BC1:** *Test daily limits*
* **Test Case BC2:** *Test negative values and overdraft*
* **Test Case BC3:** *Test boundary values*

**5. Test Execution and Screenshots**

**Execution Process**

1. **Simulated Run:**  
   We executed each test case by providing the input values manually via the console.
2. **Result Collection:**  
   For each test case, the output on the console was recorded and compared against the expected behavior.