Lab 06 Report

By Josh Scheitler and Ethan Meyer

# 4. 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 schematic diagram of the three subsystems (Bank Checking, Bank Saving, and Utility

Regenerate??

## A diagram of a machine AI-generated content may be incorrect.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

# 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.

# 4. 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() |

# 5. UI Testing

The following UI tests evaluate user interactions with the command-line interface (CLI) for the ATM system. Two testing methods were used: functionality testing and boundary testing.

## i. Testing Methods and Criteria

1. Functionality Testing – Ensures all user options and flows operate as expected.

2. Boundary Testing – Tests edge cases for deposit, withdrawal, and transfer limits.

## ii. Test Requirements (TR)

TR1 (Functionality): User can successfully login and navigate all menu options.

TR2 (Boundary): Deposits, withdrawals, and transfers must not exceed limits or accept invalid values.

## iii. Test Cases and Results

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Method | Criterion | Input Values | Expected Output | Actual Output | Success/Fail |
| Functionality | Login & Navigate | 1 → username → password | User logged in, sees account options | Matched expected flow | Success |
| Functionality | Checking Deposit | 500 | Balance increases by $500 | Balance increased correctly | Success |
| Functionality | Utility History | 3 → 1 | Displays payment history or empty | Payment history shown correctly | Success |
| Boundary | Over-limit Withdraw | Withdraw $600 (limit = $500) | Error: withdraw limit exceeded | Error message printed | Success |
| Boundary | Negative Deposit | Deposit -100 | Error: invalid amount | Error message printed | Success |
| Boundary | Savings Transfer Limit | Transfer $150 (limit = $100) | Error: transfer limit exceeded | Error message printed | Success |

## iv. Analysis of Results

All test cases executed as expected. Functionality tests validated correct flow from login to account operations. Boundary tests confirmed that the system rejects invalid inputs and enforces transaction limits. No unexpected behavior was observed during the CLI interaction tests.

## v. Interpretation Report

The CLI interface correctly implements all major functionalities required by the ATM system. Transactions involving deposits, withdrawals, and transfers are guarded by daily limits and input validations. Login methods function correctly using either username or account number. User feedback through console output is clear and consistent.