System Requirements Specification Index

For

Banking System Error Handling Framework

Version 1.0

IIHT Pvt. Ltd.
fullstack@iiht.com

TABLE OF CONTENTS

1 PROJECT ABSTRACT

The Banking System Error Handling Framework (BSEHF) demonstrates three main error types: syntax errors, runtime exceptions, and logical errors. This banking application showcases input validation, exception handling, and data integrity protection.

2 BUSINESS REQUIREMENTS:

Screen Name	Console input screen
Problem Statement	1. Handle syntax, runtime, and logical errors 2. Maintain transaction integrity during exceptions 3. Validate all user inputs with appropriate error messages 4. Implement custom exception hierarchy 5. Record error states in transaction history

3 CONSTRAINTS

3.1 CLASS REQUIREMENTS

1. `BankAccount` Class:
   - Methods for deposit, withdrawal, and balance inquiry
   - Error handling for insufficient funds and invalid amounts
   - Transaction tracking with error states
   - Exception propagation

2. `InputValidator` Class:
    - Validation methods for amounts and account IDs
    - Type conversion with error handling

3.2 Error Handling Requirement

1. Syntax Error Handling:
    - Validate numeric and string formats
    - Handle malformed inputs with custom exceptions
    - Catch decimal conversion errors
2. Runtime Exception Handling:
    - Use try-except blocks for operations
    - Catch specific exception types
    - Propagate exceptions appropriately
3. Logical Error Prevention:
    - Validate state before/after operations
    - Verify transaction integrity
    - Ensure balance changes are correct
4. Custom Exception Hierarchy:
    - Base `BankingException` class
    - Specialized exceptions with proper inheritance
    - Informative error messages and codes

3.3 Exception Types

1. `BankingException` - Base exception
    - Properties:
        - `message`: Descriptive error message
        - `error_code`: Unique identifier for error type
    - Methods:
        - Custom `__str__` implementation for formatting
2. `InvalidInputError` - For syntax errors
    - Use cases:
        - Invalid formats
        - Type mismatches
        - Out-of-range values

- Required information:
    - Input that failed validation
    - Expected format/type

3. `InvalidAmountError` - For negative/zero amounts

   - Use cases:
       - Zero amount transactions
       - Negative deposits/withdrawals
   - Required information:
       - Attempted amount
       - Constraint violation details

4. `InsufficientFundsError` - For failed withdrawals

   - Use cases:
       - Withdrawals exceeding balance
       - Transfers exceeding source balance
   - Required information:
       - Account ID
       - Requested amount
       - Current balance

3.4 Implementation Constraints

1. Exception handling patterns:

   - No bare except blocks
   - Specific exception catching
   - Proper exception propagation
   - No exception silencing

2. Transaction integrity:

   - Implement transaction rollback for failed transfers
   - Ensure consistent state after exceptions
   - Maintain money conservation principle

3. Testing support:

   - Support unit testing of error conditions
   - Provide clear error information for tests
   - Allow exception verification in test cases

4. Performance considerations:

   - Minimize exception throwing for expected cases
   - Use validation before operations when possible
   - Optimize error checking for critical paths

4. TEMPLATE CODE STRUCTURE:

1. Exception Classes:

- `BankingException` (base class)
- `InvalidInputError` (syntax errors)
- `InvalidAmountError` (specialized input error)
- `InsufficientFundsError` (runtime error)

2. Input Validation:

- `InputValidator` class with static methods:
  - `validate_amount(amount)`
  - `validate_account_id(account_id)`

3. Core Banking Classes:

- `BankAccount` class:
  - Initialization with validation
  - Account operations with error handling
  - Transaction history tracking

4. Transaction Functions:

- `transfer(from_account, to_account, amount)`:
  - Validation and error handling
  - Transaction integrity verification
  - Rollback capability for failures

5. Demonstration:

- `main()` function demonstrating all error types
- Example usage scenarios
- Error case demonstrations

5. EXECUTION STEPS TO FOLLOW:

1. Exception Hierarchy:
   - Define base `BankingException` class
   - Implement specialized exceptions
   - Add error codes and message formatting

2. Input Validation:
   - Create `InputValidator` class
   - Implement validation methods

   - Add comprehensive error detection

3. Account Operations:
   - Build `BankAccount` class with validation
   - Add transaction history tracking
   - Implement error handling in methods

4. Transaction Integrity:
   - Create transfer function with validation
   - Add rollback capability
   - Implement money conservation checks

5. Testing and Demonstrations:
   - Create main function with examples
   - Demonstrate all error types
   - Show recovery from errors