System Requirements Specification Index

For

Banking System Error Handling Framework

Version 1.0

IIHT Pvt. Ltd.

fullstack@iiht.com

Table of Contents

Banking System Error Handling Framework

System Requirements Specification

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. Detailed Function Structure:

5.1 Exception Hierarchy Implementation

1. Write a Python base exception class for banking operations.

Define: **class BankingException(Exception):**

This class should:

- Accept parameters: message, error_code=None
- Initialize attributes: `self.message = message`, `self.error_code = error_code`
- Call parent constructor: `super().__init__(self.message)`
- Implement `__str__` method: Return `f"[{self.error_code}] {self.message}"` if error_code exists, otherwise return `self.message`
- Example usage: `BankingException("Test error", "B001")` should display as "[B001] Test error"

2. Write a Python exception class for invalid input errors.

Define: **class InvalidInputError(BankingException):**

This class should:

- Inherit from BankingException
- Used for format validation errors, type mismatches
- Accept message and error_code parameters
- Pass parameters to parent class constructor
- Example: Used when account ID format is invalid or input types are wrong

3. Write a Python exception class for invalid amount errors.

Define: **class InvalidAmountError(InvalidInputError):**

This function should:

- Inherit from InvalidInputError
- Accept amount parameter in constructor
- Create error message: `f"Invalid amount: {amount}. Amount must be positive."`
- Use error code "E001"
- Call parent constructor: `super().__init__(message, "E001")`
- Example: `InvalidAmountError("-100")` should show "Invalid amount: -100. Amount must be positive. [E001]"

4. Write a Python exception class for insufficient funds errors.

Define: **class InsufficientFundsError(BankingException):**

This function should:

- Inherit from BankingException
- Accept parameters: account_id, amount, balance
- Create descriptive message: `f"Insufficient funds in account {account_id}. Attempted to withdraw {amount}, but balance is {balance}."`
- Use error code "T001"
- Call parent constructor with message and "T001"
- Example: Shows exact amounts and account for debugging

5.2 INPUT VALIDATION IMPLEMENTATION

5. Write a Python class for input validation with static methods.

Define: **class InputValidator:**

This class should:

- Contain only static methods (use @staticmethod decorator)
- Provide validate_amount() and validate_account_id() methods
- Handle syntax errors early in the application flow
- Raise appropriate domain-specific exceptions
- Support validation of different input types

6. Write a static method to validate monetary amounts.

Define:**@staticmethod def validate_amount(amount):**

This function should:

- Accept amount parameter (string, int, float, or Decimal)
- Use try-except block to handle Decimal conversion: amount_decimal = Decimal(str(amount))
- Catch InvalidOperation exception and raise InvalidInputError with message and error code "E003"
- Check if amount is positive: if amount_decimal <= Decimal('0'): raise InvalidAmountError(amount)
- Return validated Decimal amount
- Example: validate_amount("100.50") returns Decimal("100.50")
- Example error: validate_amount("-10") raises InvalidAmountError

7. Write a static method to validate account ID format.

Define:**@staticmethod def validate_account_id(account_id):**

This method should:

- Check if account_id is string: if not isinstance(account_id, str): raise InvalidInputError("Account ID must be a string", "E004")
- Use regex validation: if not re.match(r'^[a-zA-Z0-9]{8,12}$', account_id):
- Raise InvalidInputError with descriptive message and error code "E005" for invalid format
- Return validated account_id if successful
- Example valid: "ACCT123456", "12345678"
- Example invalid: "ABC" (too short), "ACCT123&*" (invalid characters)

5.3 BankAccount Class Implementation

8. Write a Python class to represent a bank account with error handling.

Define: **class BankAccount:**

This class should:

- Accept parameters: account_id, owner_name, initial_balance=0
- Use InputValidator to validate account_id
- Validate owner_name is non-empty string: `if not isinstance(owner_name, str) or not owner_name.strip(): raise InvalidInputError("Owner name cannot be empty", "E006")`
- Validate initial_balance using InputValidator.validate_amount()
- Initialize transaction_history as empty list
- Store balance as Decimal type
- Handle and re-raise BankingException errors during initialization

9. Write a deposit method with comprehensive error handling.

Define: **def deposit(self, amount):**

This method should:

- get_balance(): Return current balance (Decimal type)
- get_transaction_history(): Return copy of transaction history list
- No error handling needed for simple property access
- Provide read-only access to account state
- Use try-except block for error handling
- Validate amount using InputValidator.validate_amount()
- Create transaction record: `{'type': 'deposit', 'amount': amount, 'status': 'pending'}`
- Store previous_balance before operation
- Update balance: `self.balance += amount`
- Verify logical correctness: `if self.balance <= previous_balance: raise BankingException("Logical`

`error in deposit: Balance didn't increase", "L001")`

- Update transaction status to 'completed' and append to history
- Return new balance
- Handle exceptions by logging failed transaction and re-raising

10 . Write a withdrawal method with fund checking and error handling.

Define: **def withdraw(self, amount):**

This function should:

- Use try-except block for error handling
- Validate amount using InputValidator.validate_amount()
- Create transaction record: {'type': 'withdrawal', 'amount': amount, 'status': 'pending'}
- Check for sufficient funds: if amount > self.balance: raise InsufficientFundsError(self.account_id, amount, self.balance)
- Store previous_balance before operation
- Update balance: self.balance -= amount
- Verify logical correctness: if self.balance >= previous_balance: raise BankingException("Logical error in withdrawal: Balance didn't decrease", "L002")
- Update transaction status to 'completed' and append to history
- Return new balance
- Handle exceptions by logging failed transaction and re-raising

11. Write getter methods for account properties.

Define: **def get_balance(self): and def get_transaction_history(self):**

This method should:

- get_balance(): Return current balance (Decimal type)
- get_transaction_history(): Return copy of transaction history list
- No error handling needed for simple property access
- Provide read-only access to account state

5.4 Transfer Function Implementation

12. Write a function to transfer funds between accounts with transaction integrity.

Define: **def transfer(from_account, to_account, amount):**

This function should:

- Validate amount using InputValidator.validate_amount()
- Check sufficient funds: `if amount > from_account.balance: raise InsufficientFundsError(from_account.account_id, amount, from_account.balance)`
- Store initial_balance for rollback capability
- Withdraw from source: `from_account.balance -= amount`
- Use try-except block around destination deposit
- Call `to_account.deposit(amount)` inside try block
- Verify money conservation: check total money before and after
- If deposit fails or logical error detected, rollback: `from_account.balance = initial_balance`
- Return transaction details dictionary with balances and status
- Handle all exceptions with proper rollback

5.5 Main Program Function

13. Write a main function demonstrating all error handling types.

Define: **def main()**

This function should:

- Print descriptive headers for each demonstration section
- Create valid accounts to demonstrate success cases
- Demonstrate syntax error handling: Try creating account with invalid ID
- Demonstrate runtime error handling: Try withdrawal with insufficient funds
- Demonstrate logical error prevention: Perform transfer and verify money conservation
- Use try-except blocks around each demonstration
- Print caught exceptions with descriptive context
- Show initial and final system state
- Verify transaction integrity using assertions
- Example structure:
    - print("1. Creating account with proper values...")

- print("2. Creating account with invalid ID (syntax error)...")
- print("3. Withdrawing with insufficient funds (runtime error)...")
- print("4. Transfer with transaction integrity...")

6. 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

<u>Execution Steps to Follow:</u>

- All actions like build, compile, running application, running test cases will be through Command Terminal.
- To open the command terminal the test takers, need to go to Application menu (Three horizontal lines at left top) -> Terminal -> New Terminal
- This editor Auto Saves the code
- If you want to exit(logout) and continue the coding later anytime (using Save & Exit option on Assessment Landing Page) then you need to use CTRL+Shift+B -command compulsorily on code IDE. This will push or save the updated contents in the internal git/repository. Else the code will not be available in the next login.
- These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.
- To setup environment:
- Pip install --upgrade typing_extensions
- To launch application: Python3 filename.py
- To run Test cases: In the terminal type pytest
- Before Final Submission also, you need to use CTRL+Shift+B - command compulsorily on code IDE, before final submission as well. This will push or save the updated contents in the internal git/repository, and will be used to evaluate the code quality.

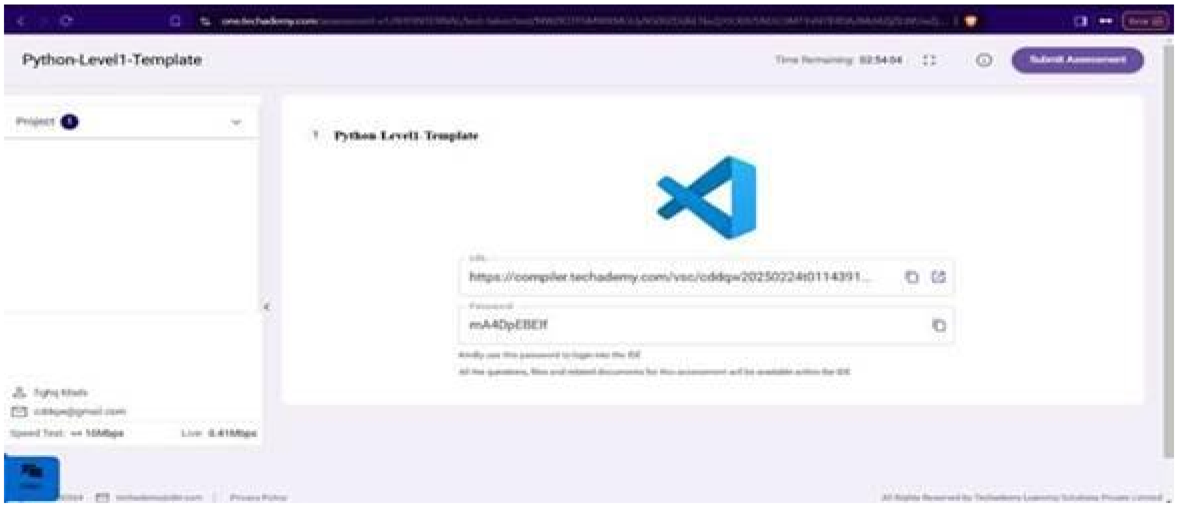
Screen shot to run the program

To run the application

Python3 filename.py

To run the testcase

type pytest in the terminal



https://compiler.techademy.com/vsc/cddqw20250224t0114391...

mA4DpEBEIf

● Once you are done with development and ready with submission, you may navigate to the previous tab and submit the workspace. It is mandatory to click on “Submit Assessment” after you are done with code.