Code Review of The Software Project:
Money Manager application

Course Title: Software Development Project
Course ID: CSE- 3106

Project By:
SK Miraz Rahman Ani
Student ID: 210211
Mohommad IbnSina
Student ID: 200210

Reviewed By:

M.d. Ashiquzzaman Rahad
Student ID: 210201
Jannatul Ferdous Nijhum
Student ID: 210239

Submitted To:

Amit Kumar Mondal
Associate Professor
Computer Science & Engineering Discipline
Khulna University,
Khulna.

Introduction

This code review evaluates the Money Manager application, including the login, registration, and dashboard, summary, income and expense sections. The review identifies areas for improvement, adherence to best practices, and suggestions for enhancing maintainability, security, and overall code quality.In this code review, bad smells of the code, architecture evaluation, modularity check, condition statements of the code & other related sections are evaluated.

Code Smells

1. **Large or complex methods:**
   There are not many large or complex methods which can be difficult to read and understand. The average method size is close to 10 lines of code which is close to the standard size of methods. The highest size of method is in expense.py module with around 24 lines of code (submit_button_press method).

2. **Long parameter lists:**
   There is no method with long parameter lists.

3. **Excessive comments:**
   In some modules there are excessive comments and some doesnot have any comment at all. In the dashboard.py, there are enough comments. But in the expense.py, income.py, registration.py and summary.py modules, there is no comment at all. So, there are inconsistencies in using comments.

4. **Duplicate code:**
   There are some duplicate code in the similar kind of methods. But other than that code reusibility has been done quite effectively in most of the modules.

5. **Inconsistent naming conventions:**
   In most cases, naming conventions are standardized. For example: switch_income(), switch_login(), relative_to_assets(), etc.

6. **Incomplete error handling:**
   Errors are mostly handled in each of the modules. There are mainly scope of running into error while doing file operations which seems to be handled carefully.

7. **Too many if/else statements:**
   In the dashboard.py, expense.py and income.py modules, there are some excessive use of if/else statements. But in other modules, the usage of if/else statements are moderated.

8. **Poor use of inheritance:**
   There is no poor use of inheritance in this project which is particularly causing any problem.

9. **Unnecessary dependencies:**
   No third party libraries or frameworks are used here.So there is no unnecessary dependencies on the code.

10. **Magic numbers or hard-coded values:**
    Some magic numbers are used in the code for different functional works.For example,in dashboard.py ,magic numbers are used for drawing pie chart purpose.

Proposed Architecture Evaluation

The proposed architecture of the project is **“Repository Architecture”**. In the project, we can see the reflection of the proposed architecture. There are different layers or modules in the project.

1. Centralized Data Management: A repository architecture allows for centralizing all financial data related to accounts, transactions, budgets, and other relevant information. This centralized approach simplifies data management, ensuring consistency and reducing the likelihood of data discrepancies.

2. Abstraction of Data Access: By employing a repository pattern, the data access logic is abstracted away from the rest of the application. This separation of concerns makes the codebase more maintainable and flexible, as changes to the underlying data storage technology can be made without impacting the business logic.

3. Scalability and Flexibility: A well-designed repository architecture facilitates scalability by providing a modular structure that can accommodate changes in data volume and system complexity. As the money management system grows, additional repositories can be added or existing ones modified to support new features and requirements.

4. Interoperability and Integration: Repositories provide a standardized interface for accessing and manipulating data, making it easier to integrate the money management system.

So, we can say that the project reflects the proposed **Repository Architecture** more or less.

Modularity Check

The project is divided into 6 modules which are **dashboard.py, income.py, expense.py ,registration.py,login.py & summery.py.**

login.py module has the main window and frames of the user interface where user can login with their email and password.If the user is not registered yet,the system moves to the user to **registration.py** module which serves user to registered with their name, email and password. In

dashboard.py module,the user can see balance, the overview of the system of their income and expense details in pie chart. **income.py** module serves the user to input income source and income amount and **expense.py** serves to take input of expense source and expense amount from the user.In **summery.py** modules,the user can see the summery of his income and expense details according to date.

If/else Condition to Switch statement

There is no built-in switch case statement in python. As a result, there is no other way to implement the conditions required in the code other than using if/else statement.

The Code Review Checklist provides a company guideline for checking code including pass/fail parameters and recording any comments when the test fails.

During a project, this document is used by team members as follows:

1. During project planning, it is utilized as a reminder for how much review time should be allocated during the project for the software being developed
2. During design and development (coding) portion of the project, the checklists are used to conduct code reviews.

The list of test items is representative and should be modified prior to use to reflect your development environment and standards.

Generic Checklist for Code Reviews

Structure

	Description of Item	Pass	Fail	Comments
☐	Does the code completely and correctly implement the design?	☑	☐	
☐	Does the code conform to any pertinent coding standards?	☑	☐	
☐	Is the code well-structured, consistent in style, and consistently formatted?	☑	☐	
☐	Are there any uncalled or unneeded procedures or any unreachable code?	☐	☑	
☐	Are there any leftover stubs or test routines in the code?	☐	☑	
☐	Can any code be replaced by calls to external reusable components or library functions?	☑	☐	
☐	Are there any blocks of repeated code that could be condensed into a single procedure?	☐	☑	
☐	Is storage use efficient?	☑	☐	
☐	Are symbolics used rather than “magic number” constants or string constants?	☐	☑	
☐	Are any modules excessively complex and should be restructured or split into multiple routines?	☐	☑	

Documentation

	Description of Item	Pass	Fail	Comments
☐	Is the code clearly and adequately documented	☑	☐	

	with an easy-to-maintain commenting style?			
☐	Are all comments consistent with the code?	☐	☑	

Variables

	Description of Item	Pass	Fail	Comments
☐	Are all variables properly defined with meaningful, consistent, and clear names?	☑	☐	
☐	Do all assigned variables have proper type consistency or casting?	☑	☐	
☐	Are there any redundant or unused variables?	☐	☑	

Style

	Description of Item	Pass	Fail	Comments
☐	Does the code follow the style guide for this project?	☑	☐	
☐	Is the header information for each file and each function descriptive enough?	☑	☐	
☐	Is there an appropriate amount of comments? (frequency, location, and level of detail)	☐	☑	
☐	Is the code well structured? (typographically and functionally)	☑	☐	
☐	Are the variable and function names descriptive and consistent in style?	☑	☐	
☐	Are "magic numbers" avoided? (use named constants rather than numbers)	☐	☑	
☐	Is there any “dead code” (commented out code or unreachable code) that should be removed?	☐	☑	
☐	Is it possible to remove any of the assembly language code, if present?	☐	☑	
☐	Is the code too tricky? (Did you have to think hard to understand what it does?)	☐	☑	
☐	Did you have to ask the author what the code does? (code should be self-explanatory)	☑	☐	

Architecture

	Description of Item	Pass	Fail	Comments
☐	Is the function too long? (e.g., longer than fits on one printed page)	☐	☑	
☐	Can this code be reused? Should it be reusing something else?	☑	☐	

☐	Is there minimal use of global variables? Do all variables have minimum scope?	☑	☐	
☐	Are classes and functions that are doing related things grouped appropriately? (cohesion)	☐	☑	
☐	Is the code portable? (especially variable sizes, e.g., “int32” instead of “long”)	☑	☐	
☐	Are specific types used when possible? (e.g., “unsigned” and typedef, not just "int")	☑	☐	
☐	Are there any if/else structures nested more than two deep? (consecutive “else if” is OK)	☐	☑	
☐	Are there nested switch or case statements? (they should never be nested)	☑	☐	

Arithmetic Operations

	Description of Item	Pass	Fail	Comments
☐	Does the code avoid comparing floating-point numbers for equality?	☐	☑	
☐	Does the code systematically prevent rounding errors?	☑	☐	
☐	Does the code avoid additions and subtractions on numbers with greatly different magnitudes?	☑	☐	
☐	Are divisors tested for zero or noise?	☐	☑	

Loops and Branches

	Description of Item	Pass	Fail	Comments
☐	Are all loops, branches, and logic constructs complete, correct, and properly nested?	☑	☐	
☐	Are the most common cases tested first in IF- -ELSEIF chains?	☑	☐	
☐	Are all cases covered in an IF- -ELSEIF or CASE block, including ELSE or DEFAULT clauses?	☐	☑	
☐	Does every case statement have a default?	☑	☐	
☐	Are loop termination conditions obvious and invariably achievable?	☑	☐	
☐	Are indexes or subscripts properly initialized, just prior to the loop?	☑	☐	
☐	Can any statements that are enclosed within loops be placed outside the loops?	☐	☑	
☐	Does the code in the loop avoid manipulating the index variable or using it upon exit from the loop?	☑	☐	

Defensive Programming

	Description of Item	Pass	Fail	Comments
☐	Are indexes, pointers, and subscripts tested against array, record, or file bounds?	☐	☐	
☐	Are imported data and input arguments tested for validity and completeness?	☐	☐	
☐	Are all output variables assigned?	☐	☐	
☐	Are the correct data operated on in each statement?	☐	☐	
☐	Is every memory allocation deallocated?	☐	☐	
☐	Are timeouts or error traps used for external device accesses?	☐	☐	
☐	Are files checked for existence before attempting to access them?	☐	☐	
☐	Are all files and devices left in the correct state upon program termination?	☐	☐	

Maintainability

	Description of Item	Pass	Fail	Comments
☐	Does the code make sense?	☑	☐	
☐	Does the code comply with the accepted Coding Conventions?	☑	☐	
☐	Does the code comply with the accepted Best Practices?	☑	☐	
☐	Does the code comply with the accepted Comment Conventions?	☑	☐	
☐	Is the commenting clear and adequate? (As a guide, each file will have a comment at the start, explaining what the code does, possibly a comment at the start of each function, and comments as needed to explain complex or obfuscated code.)			
☐	Are ideas presented clearly in the code?	☑	☐	
☐	Is encapsulation done properly?	☐	☑	
☐	Is the code not too complex?	☑	☐	
☐	Are there no unnecessary global variables?	☑	☐	
☐	Is the reading order in source code from top to bottom?	☑	☐	
☐	Are there unused variables or functions?	☐	☑	

Requirements and Functionality

	Description of Item	Pass	Fail	Comments
☐	Does the code match the requirements, specifications and standards?	☐	☐	
☐	Is the logic proper? Does the code function as needed?	☐	☐	

System and Library Calls

	Description of Item	Pass	Fail	Comments
☐	Do all system calls have their return status checked?	☐	☐	
☐	Are all possible errors from system or library calls handled?	☐	☐	
☐	Are signals caught and handled?	☐	☐	
☐	Is mutex() used on multithreaded code on global variables?	☐	☐	

Reusability

	Description of Item	Pass	Fail	Comments
☐	Are all available libraries being used effectively?	☐	☐	
☐	Are available openmrs util methods known and used?	☐	☐	
☐	Is the code as generalized/abstracted as it could be?	☐	☐	
☐	Is the code a candidate for reusability?	☐	☐	

Robustness

	Description of Item	Pass	Fail	Comments
	Are all parameters checked?	✓		
	Are error conditions caught?		✓	
	Is there a default case in all switch statements?	✓		
	Is there non-reentrant code in dangerous places?		✓	
	Is the usage of macros proper? (Readability, complexity, portability…)	✓		
	Is there unnecessary optimization that may hinder maintainability?		✓	

Security

	Description of Item	Pass	Fail	Comments
☐	Does the code appear to pose a security concern?	☐	☐	
☐	Do Service methods have an @Authorize annotation on them	☐	☐	
☐	Does the application use an inclusion list (known, valid, and safe input) rather than an inclusion list (rejecting known malicious or dangerous input)	☐	☐	
☐	Is all user input encoding set by the server?	☐	☐	
☐	Is all character encoding set by the server?	☐	☐	
☐	If cookies contain sensitive data, are they marked secure?	☐	☐	
☐	Do input surfaces in Web parts and other customizations include boundary checks, input data integrity checks, and appropriate exception handling to protect from cross-site scripting and SQL injection.	☐	☐	
☐	Does the design address potential canonicalization issues?	☐	☐	

Control Structures

	Description of Item	Pass	Fail	Comments
☐	Does the application log sensitive data in clear text.	☐	☐	
☐	Sensitive data is not stored in cookies.	☐	☐	
☐	Sensitive data is not stored in unencrypted, hidden form fields or query strings. It is maintained by using server-side state management.	☐	☐	
☐	SSL, IPSEC with encryption, or application layer encryption prior to transmittal is used to protect sensitive data during transmission.	☐	☐	
☐	Sensitive data is not cached. Output caching is off by default.	☐	☐	
☐	Sensitive data that is transferred via email uses S/MIME encryption or Information Rights Management (IRM), depending upon the intended recipient.	☐	☐	
☐	Does the code make use of infinite loops?	☐	☐	
☐	Does the loop iterate the correct number of times?	☐	☐	

Resource Leaks

	Description of Item	Pass	Fail	Comments
☐	Does the code release resources?	☐	☐	
☐	Does the code release resources more than once?	☐	☐	
☐	Does the code use the most efficient class when dealing with certain resources?	☐	☐	

Error Handling

	Description of Item	Pass	Fail	Comments
☐	Does the code comply with the accepted Exception Handling Conventions?	☐	☑	
☐	Does the code make use of exception handling?	☐	☑	
☐	Does the code simply catch exceptions and log them?	☐	☑	
☐	Does the code catch general exception (java.lang.Exception)?	☐	☑	
☐	Does the code correctly impose conditions for "expected" values?	☑	☐	
☐	Are input parameters checked for proper values (sanity checking)?	☑	☐	
☐	Are error return codes/exception generated and passed back to the calling function?	☑	☐	
☐	Are error return codes/exceptions handled by the calling function?	☑	☐	
☐	Are null pointers and negative numbers handled properly?	☑	☐	
☐	Do switch statements have a default clause used for error detection?	☐	☑	
☐	Are arrays checked for out of range indexing? Are pointers similarly checked?	☑	☐	
☐	Is garbage collection being done properly, especially for errors/exceptions?	☑	☐	
☐	Is there a chance of mathematical overflow/underflow?	☐	☑	
☐	Are error conditions checked and logged? Are the error messages/codes meaningful?	☑	☐	
☐	Would an error handling structure such as try/catch be useful? (depends upon language)	☐	☑	

Timing

	Description of Item	Pass	Fail	Comments
☐	Is the worst case timing bounded? (no unbounded loops, no recursion)	☐	☐	
☐	Are there any race conditions? (especially multi-byte variables modified by an interrupt)	☐	☐	
☐	Is appropriate code thread safe and reentrant?	☐	☐	
☐	Are there any long-running ISRs? Are interrupts masked for more than a few clocks?	☐	☐	
☐	Is priority inversion avoided or handled by the RTOS?	☐	☐	
☐	Is the watchdog timer turned on? Is the watchdog kicked only if every task is executing?	☐	☐	
☐	Has code readability been sacrificed for unnecessary optimization?	☐	☐	

Validation & Test

	Description of Item	Pass	Fail	Comments
☐	Is the code easy to test? (How many paths are there through the code?)	☐	☐	
☐	Do unit tests have 100% branch coverage? (code should be written to make this easy)	☐	☐	
☐	Are the compilation and/or lint checks 100% warning-free? (Are warnings enabled?)	☐	☐	
☐	Is special attention given to corner cases, boundaries, and negative test cases?	☐	☐	
☐	Does the code provide convenient ways to inject faulty conditions for testing?	☐	☐	
	Are all interfaces tested, including all exceptions?	☐	☐	
☐	Has the worst case resource use been validated? (stack space, memory allocation)	☐	☐	
☐	Are run-time assertions being used? Are assertion violations logged?	☐	☐	
☐	Is there commented out code (for testing) that should be removed?	☐	☐	

Hardware				
	Description of Item	**Pass**	**Fail**	**Comments**
☐	Do I/O operations put the hardware in a correct state?	☐	☐	
☐	Are min/max timing requirements met for the hardware interface?	☐	☐	
☐	Is it ensured that multi-byte hardware registers can't change during read/write?	☐	☐	
☐	Does the software ensure that the system resets to a well defined hardware system state?	☐	☐	
☐	Have brownout and power loss been handled?	☐	☐	
☐	Is the system correctly configured for entering/leaving sleep mode (e.g. timers)?	☐	☐	
☐	Have unused interrupt vectors been directed to an error handler?	☐	☐	
☐	Has care been taken to avoid EEPROM corruption? (e.g., power loss during write)	☐	☐	