CSC207: Phase 2 Design Document

Group Members: Aliyan, Bernard, Ruiting, Tiffany, Miguel, Linh, Anita, Leya

Updated README: refer to readme file

Accessibility Report:

Principle 1: Equitable Use - All users have access to the same privacy, security and safety. Our program allows users to create username and password unique to them. All features are accessible to all users, regardless of their financial situation. However, we would have wished to implement designs for accessibility such as individuals with visual impairment.

Principle 2: Flexibility in Use - Our app allows the user to add as much or little precision as they wish. Accounts are options as well as budgeting features. Users can easily budget their monthly expenses if they wish to. Moving forward, we would have liked to add a "learn" page for the user so they can learn and adapt at their own pace.

Principle 3: Simple and Intuitive Use - The client interface never uses complex language and user questions are simple and short. Basic financial knowledge is required to use this app and this makes it accessible to anyone who wants to start tracking their finances.

Principle 4: Perceptible Information - Our financial planning app provides adequate contrast for the user to engage with the app. Text fonts are kept to a minimum to maximize legibility. Moving forward, we would have liked to implement different modes for accessibility.

Principle 5: Tolerance for Error - Our program minimizes errors by limiting user error. For example, when creating a budget, all categories are set to 0 by default and the user may only change amounts in valid categories (Otherwise the program will inform the user of a non valid category). Furthermore, the program assures that the user's account can't be in the negative.

Principle 6: Low Physical Effort - Our program can be used efficiently and comfortably with a minimum of fatigue. Our program requires an electronics device for use, hence it consistently allows the user to maintain a neutral body position and to use minimal operating forces.

Principle 7: Size and Space for Approach and Use - The user interface in console provides a clear line of sight to the instructions and displays for any seated or standing user. It makes reach to all components comfortable for any seated or standing user.

2. Our financial planner is marketed towards the average adult between the ages of 25-50. This group of people have everyday financial needs and long-term money goals and our financial planner can guide them to achieve these goals while making money chores and tasks easier. Our financial app can help users build their assets and assist them to make the most out of their investments to secure a better future for themselves and their family. By organizing and assessing their financial situation with the budget feature, our users will have a better understanding of what they want to achieve with their money and our program can help them create a plan to get there.

3. Our program is less likely to be used by people under the age of 20, who do not yet have any financial responsibilities. People who do not have access to an electronic device or are not familiar with using an electronic device are less likely to use our financial planner.

Updated Specification:

The domain for our project is a financial planning app. The app allows the user to create an account (or login to an existing account) which is linked to a total balance and their budget. Using this information the user is able to perform a variety of actions related to financial planning. For example, the users financial information can be updated by simulating withdrawals and deposits using the users total balance. Additionally, the user is able to view a history of their spending and deposit records, input and view their depositable and non-depositable assets. Each user also has the ability to set a monthly budget for themselves and they are able to view their budget as well as past performance in which they can see whether they were above or below budget. Users can also see how much interest they accumulate over x number of months based on compound interest on depositable assets and they are able to see the projected value of their non-depositable investments.

Phase 2 New Features: For phase 2 we added the view of deposit and withdrawal records and a series of budgeting tools that the user can use (for example, they can set monthly budgets for themselves and view their past budget performance)

List of All Actions:

- Adjust Budget
- Activate Budget
- View Budget
- Cash Out Non-depositable Investments

- Change Savings Balance
- Check Total Balance
- Calculate and see remaining total amount left in a budget
- Compare budget with Optimal Budget
- Create Non-depositable and Depositable Assets
- Deposit and Withdraw from Balance
- Simulate Depositable and Non-depositable Investment Values
- View Deposit and Withdrawal Record
- View User Profile
- View Transaction History
- Transfer Money from Balance to Depositable accounts (i.e credit card and savings)
- View Budget Performance
- View Budget
- View Investments

Major Design Decision:

- Changes to Budget
    - Removed remainingBudget (amount of the goal budget you have left to spend) and replaced it with actualBudget (amount you've spent so far in a certain category for the month) along with methods associated with remainingBudget
        - Fixes clean architecture issue where Budget is in charge of more than one responsibility (now Budget only has getters and setters
    - Simplified the entity so it would be easier to navigate and understand what it's supposed to actually do for the User (i.e. created a default list of categories users have and removed the option for how long a budget is for)
- Restructure of Factory Design Pattern in commands
    - Instead of multiple Action Requests, we've centralized it to a single ActionRequest class that gets accepted into all Action classes. The appropriate class is used by an enum class called Command using a switch statement
- Abandoning GUI
    - The recent changes made it such that GUI could not be implemented as it would cause issues with SRP and code smells, therefore it was decided that the GUI should not be implemented.
- Created our own Date class
    - We wrote our own date class helper class. Java.util.date used to do what we needed to do but some of the features we needed were made deprecated recently. LocalDate and Period combined almost did what we needed together but one feature we needed was missing so we made a wrapper entity for it that added the feature we needed it to do that was missing and was easier to use in other classes than using LocalDate and Period was on its own.

Description of Clean Architecture:

To demonstrate the Clean Architecture of our program we will describe a scenario walk through of a user signing up to the financial planning app. A user will first run the program and interact with the UI(Command line for now). As an example, they chose to create an account. So they will input their full name, username and password. The ClientUserController will send this information to the server and ActionFactory controller will determine which use case to call. The controller will call the CreateOwner use case to make an Owner object which is an entity that stores the information the user has provided in the sign up process. This object is then stored in OwnerRepository which is on the same level as the Owner object. Thus, as you can see the program follows the dependency rule and only interacts with classes on the same or adjacent levels. Our server and client user interface follows clean architecture as the requests that are sent between them are not entities and rather are data classes that are present in the use case layer and are allowed to move between the controller and use case layers freely. We also removed the earlier violation of using instanceof to detect which type of action should we called and migrated that to using enums

Description of SOLID Principles:

Single responsibilities principle*:

Every class in your program has a unique responsibility. We've done several modifications to the entity layer because of this principle. On November 13th there was a change to the entities layer removing SpendingRecord and DepositRecord because they were too similar; they only differed by one minor change… a minus sign. Furthermore, if you wanted to add a method in the class, we would need to do the same in the other class, which is redundant and bad practice. Therefore we removed it and made it one class, which is Record.

Open-Closed Principle*:

None of the inner layers depend on it's or any outer layers. This is shown by the complexity of making a change to the innermost layers. If you are going to modify the entities, this would shake the foundation of the program. Our team tried this when we recently deleted/merged two similar classes (what was once SpendingRecrod and DepositRecord) together (Phase 1). As well as just recently in Phase 2, when our group made modifications to Budget and Owner. We had to then go into all outer layers and modify the code if it called any of the once made methods of the modified entity class. Contrairily, it's very easy to add new extensions… This is shown in the action layers where we think of any features to add with no consequences to our whole program, especially in phase 2.

Liskov substitution principle:

Financial Asset is an example of Liskov's substitution principle. The Financial Assets class is subclassed by Depositable and NonDepositable where the methods are overwritten for their own unique personalization. Substituting either of the subclasses with the superclass instance of FinancialAsset won't break the application.

Interface Segregation principle:

Our program could have put all the actions into one class. This would shrink the number of classes by a large factor but it would break the interface segregation principle, hence this is why we create a distinguished class for each and every action and every action has it's unique response class that the Client UI calls. For example: ViewBudgetPerformance action has its own class in action and it's own response in ViewBudgetPerformanceResponse.

- Organize the features of our app
- Easy to distinguish errors in when testing
- Each class has a single responsibility
- Code within the class doesn't depend on methods it doesn't use

Dependency inversion principle:

The Server class (which is a controller class) follows the Dependency Inversion Principle because it depends on an abstract layer Actions, and all the concrete actions (use cases) extend from Actions. The Actions abstract class has a method, process, where Server calls on and all the concrete actions implement this process method. You can also notice that the entity layer does not call any methods or use any variables/objects from their outer layers.

Description of Packaging Strategy:

We decided to organize our code using the package by layer packaging strategy to separate out code based on what it does from a technical perspective. Using this strategy we separated each layer of the clean architecture into its own package. We chose this strategy because it was easier to ensure that we did not violate the Clean Architecture. Additionally, because each member of the group was assigned to work on a specific layer rather than a feature, we decided that separating them into different packages and organizing it by layer would be easier to work with.

Summary of Design Pattern:

- Strategy Design Pattern (Pull request #100): The strategy design pattern was used in the Simulate class to implement the simulations for the depositable and non-depositable future values. It is implemented using the strategy interface, the context class which takes an instance of a strategy and the SimulateNonDepositable and SimulateDepositable which implements the simulation algorithms for each type of asset. This design pattern was used because we wanted to use different algorithms for calculating each financial asset and be able to easily switch between simulating the depositable and non-depositable future values when given a list of the users financial assets.

- Simple) Factory Design Pattern(Pull request #29): The simple factory design pattern was used in the ActionFactory class in the server package. ActionFactory takes in objects of type ActionRequests and uses an if - else if - else statement to create and return corresponding Actions. This design pattern was used to reduce the size and Responsibility of ServerThread and remove the hard dependency between Server and concrete Actions. So ServerThread does not need to construct the concrete action objects itself, nor does it need to know which type of concrete action is given, it can just perform the process on the action.

Refactor:

- Refactoring ClientUI (Pull request #123, #125): Renamed ClientUserInterface to ClientUserController to reduce confusion with java interface. Took out each method in ClientUserController that handles one user action to become a helper class in the ClientUI package.
- Refactoring Server (Pull request #91): Took out all the subclasses of the ActionRequest class and made ActionRequest a concrete class. Created Commands enum class where each enum represents an action name. Refactored the instanceof in ActionFactory to use the enums.

Progress Report:

- **What Worked Well In Design:**
    - The use of Factory Design meant that the implementation of new actions were simple and straightforward.
    - The workflow of the program allows for easy extensions of usable tools and implementation of completely new tools. An example would be the data storing, where the easy initialization of main data tool OwnerRepo could read and use stored data and the data could be stored easily using the class that will end the program (Server) and the actions could be handled easily using the design patterns (Factory) that we leaned towards.

- **Summary of Contributions:**

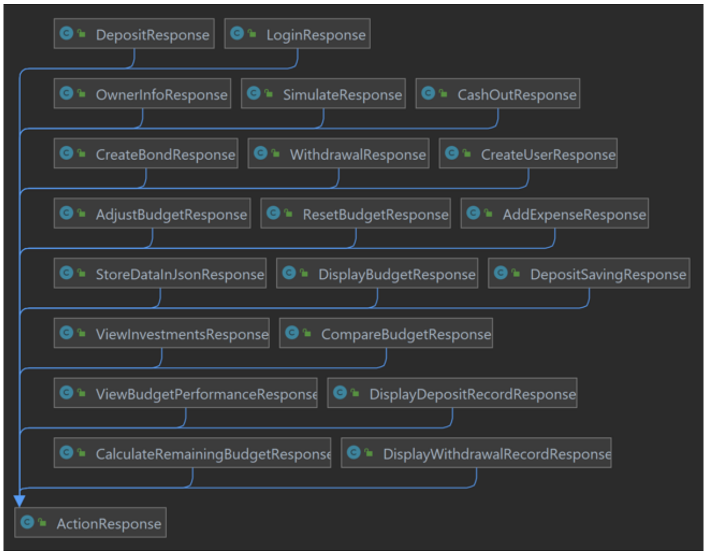
Name	What they've worked on since phase 1	Significant Pull Request and comment on why
Aliyan	Migrated the server and actionfactory to using enums and getting rid of the old request formats Moved Date to use a new	#91 as this is the pull request that we moved to using enums instead of using instanceofs. #96. In this pull request I

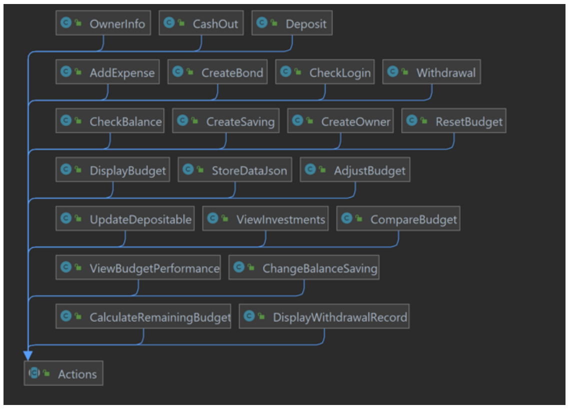
	localdate wrapper as the old one did not take into account leap years and leap days	migrated the tests to the new enum format and deleted the one leftover old request format class that was missed earlier
Bernard	- Adjust and fixed Budget class for implementations - Implemented the first few versions of AdjustBudget, resetBudget and their test cases	#86 This pull request implemented the first few versions of adjustBudget and resetBudget #106 This pull request successfully implemented test cases and passed
Ruiting	- Refactor files in ClientUI package and Server package - Connect the actions related to Budget to the ClientUserController	Pull request #91 - refactored Action factory to use enums instead of instance of Pull request #123 - refactored ClientUI file by moving out methods in ClientUserController to be their own helper classes, connected Budget related actions to ClientUserController Pull request #133 - fixed various bugs related to Budget actions and helper classes in ClientUI
Tiffany	- Implementing ViewBudgetPerformance action and its test - Implementing ActivateBudget and its test - Merging RemainingBudget and GaolBudget into one hashmap and fixing all changes in Budget as a consequence - Test cases for Budget - Worked on Accessibility Report	Pull request #112 - Refactored the Budget entity, all Budget information is now in one hashmap and the Owner entity has a BudgetHistory to keep all records of Budget.

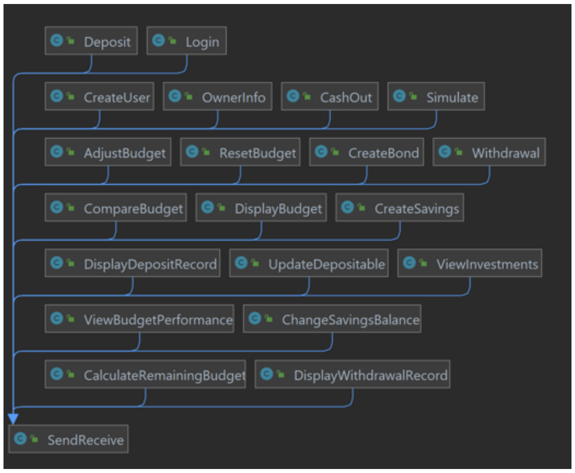
Miguel	- GUI (removed for better design architecture Pull req 138) - Data persistence fixes/Custom Json adapter - Bug Fixing on program flow - ClientUI Fixes - Tests	Pull request #127 → Data Persistence fixed with JSONTypeAdapter which is a custom adapter for data to store with type prior to being stored in list of abstract type
Linh	- Implemented the BudgetDisplay and CalculateRemaining Budget classes along with their corresponding Response classes and test cases - Fixed CompareBudget class to work with changes done to the client UI and Budget entity class - Helped refactored some client UI to fit into changes made for Clean Architecture	Pull request #110 → helps connect the user to their budget in a visual way (all the budget information is presented in one instant to view as a whole)
Anita	- Refactor files in the ClientUI package - Re-implemented the ResetBudget class and adjusted it's respond class - Updated Withdrawal to change the spending section of budget - Worked on the Accessibility Report - Wrote tests for Withdrawal, DisplayDepositRecord, ResetBudget, ChangeBalanceSaving,	Pull request #51 - this pull request implemented the first version of the Record entity that keeps track of all the users spendings and deposits, the Record class replaced both SpendingRecord and DepositRecord to make our code cleaner Pull request #55 - this pull request implemented the first version of the Transfer use case that allows the user to transfer money from

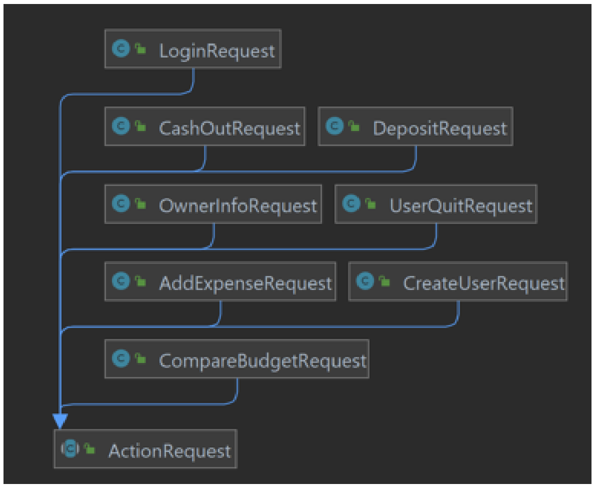
	CreateBond	balance to other accounts (e.g. Saving)
Leya	- Re-implemented the Simulate and corresponding strategy design pattern classes (SimulateContext, SimulateStrategy, NonDepositableSimulatesStrategy, DepositableSimulateStrategy), Updated corresponding SimulateResponse, Updated Simulate tests. - Re-factored NonDepositable class (and corresponding subclass) to take in a dateOfMaturity instance. - Re-implemented Transfer, TestTransfer, and TransferResponse - Updated cashout test to remove current date dependency	Pull request #100 → Updated the Simulate class to work for a list of the users financial assets rather than just a singular asset. Also was able to effectively use the strategy design pattern.

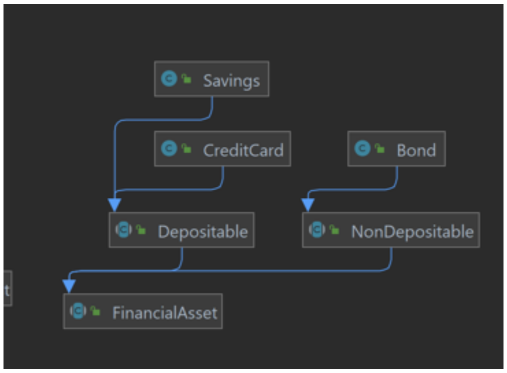
Class Diagram (Optional): - Miguel

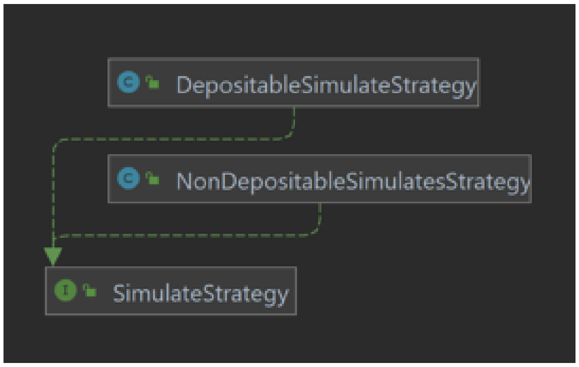


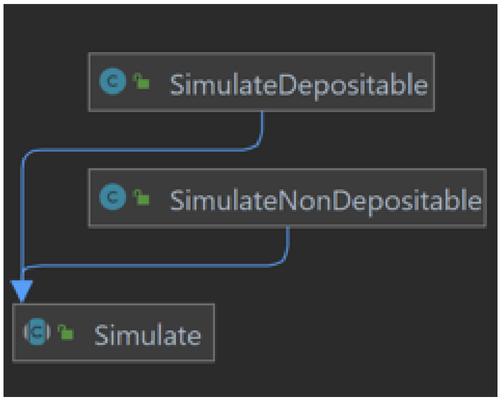


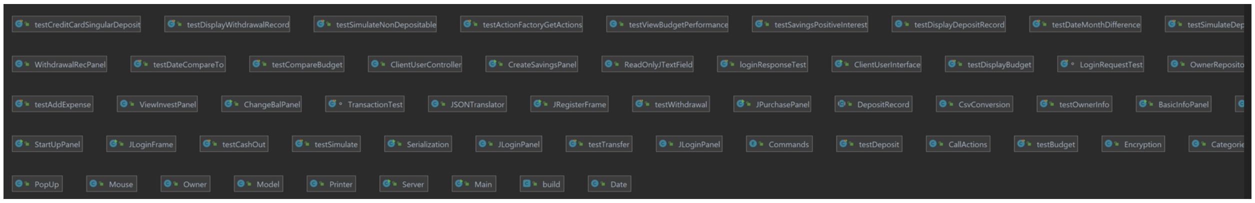












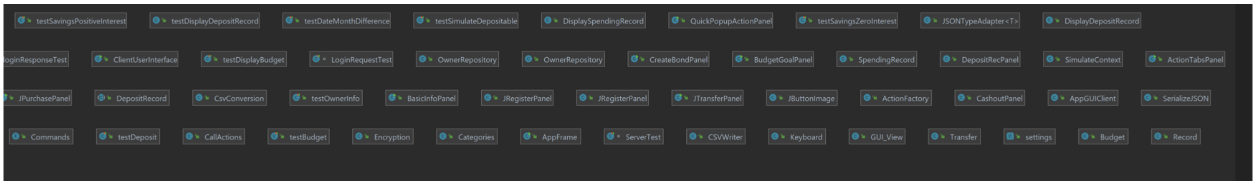
testCreditCardSingularDeposit testDisplayWithdrawalRecord testSimulateNonDepositable testActionFactoryGetActions testViewBudgetPerformance testSavingsPositiveInterest testDisplayDepositRecord testDateMonthDifference testSimulateDep

WithdrawalRecPanel testDateCompareTo testCompareBudget ClientUserController CreateSavingsPanel ReadOnlyJTextField loginResponseTest ClientUserInterface testDisplayBudget LoginRequestTest OwnerReposito

testAddExpense ViewInvestPanel ChangeBalPanel TransactionTest JSONTranslator JRegisterFrame testWithdrawal JPurchasePanel DepositRecord CsvConversion testOwnerInfo BasicInfoPanel

StartUpPanel JLoginFrame testCashOut testSimulate Serialization JLoginPanel testTransfer JLoginPanel Commands testDeposit CallActions testBudget Encryption Categorie

PopUp Mouse Owner Model Printer Server Main build Date



testSavingsPositiveInterest testDisplayDepositRecord testDateMonthDifference testSimulateDepositable DisplaySpendingRecord QuickPopupActionPanel testSavingsZeroInterest JSONTypeAdapter<T> DisplayDepositRecord

oginResponseTest ClientUserInterface testDisplayBudget LoginRequestTest OwnerRepository OwnerRepository CreateBondPanel BudgetGoalPanel SpendingRecord DepositRecPanel SimulateContext ActionTabsPanel

JPurchasePanel DepositRecord CsvConversion testOwnerInfo BasicInfoPanel JRegisterPanel JRegisterPanel JTransferPanel JButtonImage ActionFactory CashoutPanel AppGUIClient SerializeJSON

Commands testDeposit CallActions testBudget Encryption Categories AppFrame ServerTest CSVWriter Keyboard GUI_View Transfer settings Budget Record