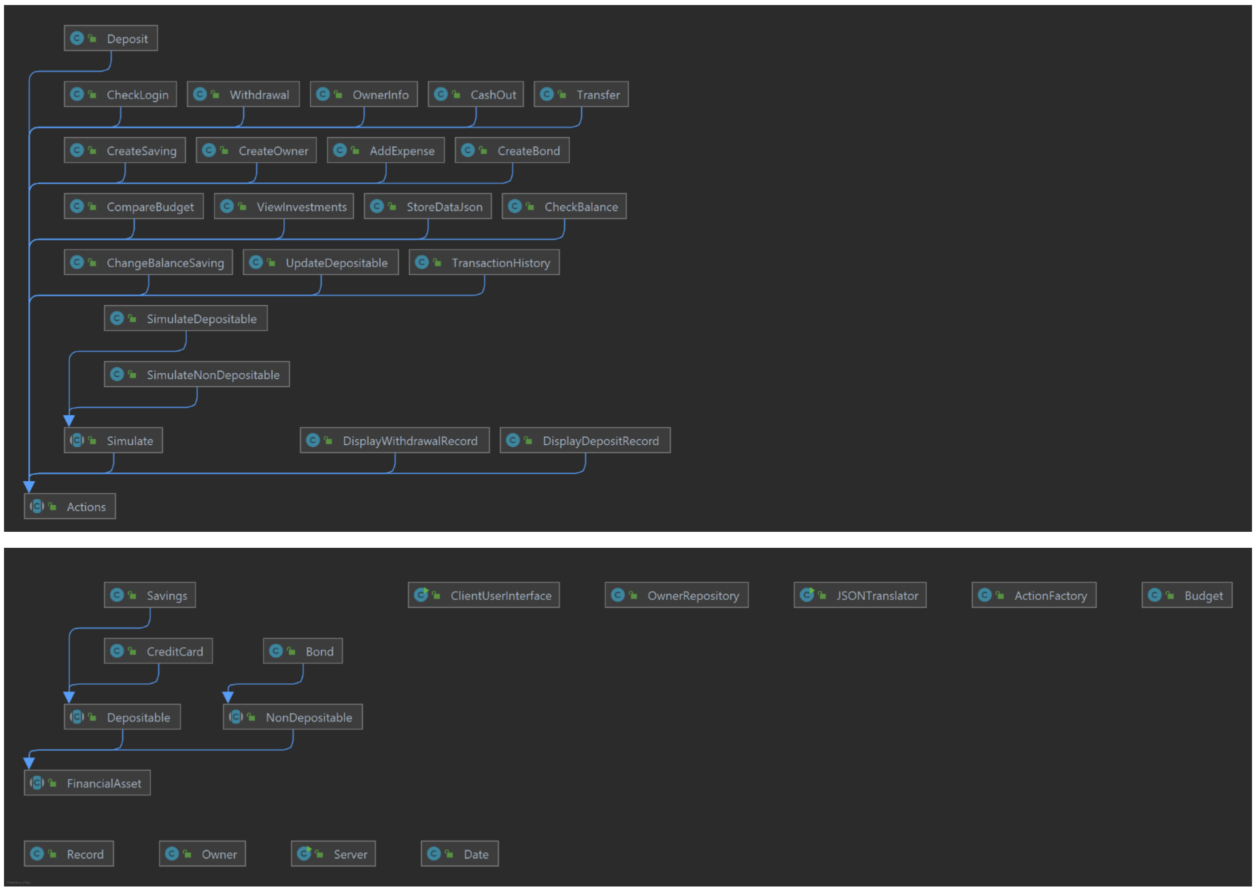
CSC207: Phase 1 Design Document

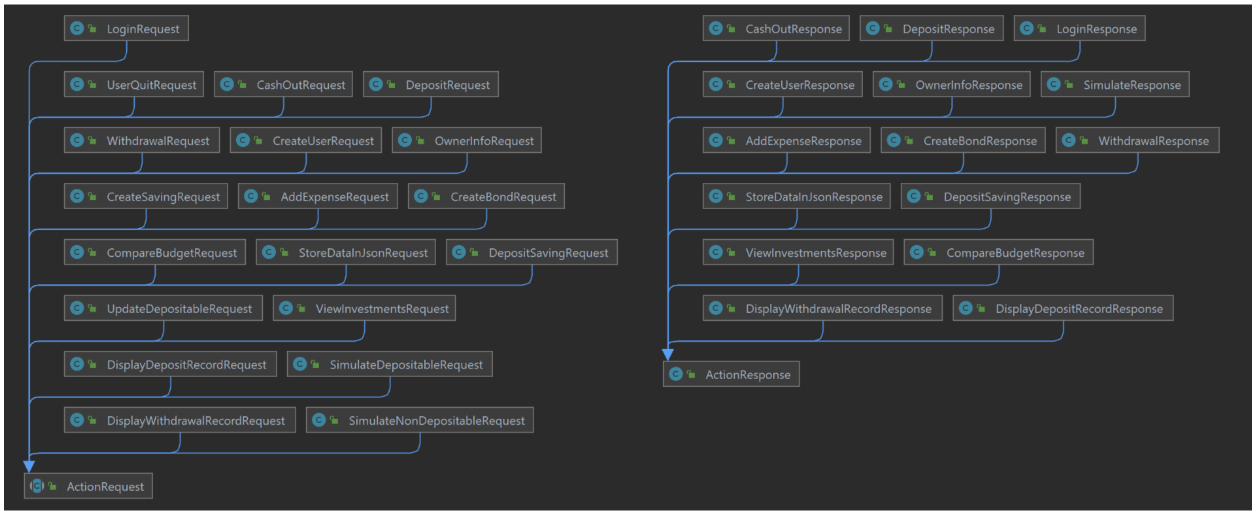
Group Members: Aliyan, Bernard, Ruiting, Tiffany, Miguel, Linh, Anita, Leya

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 into 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 a monthly limit (budget) where they are limited in the amount of money they can withdraw over a certain period of time. Users can 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 investments.

Class Diagram:





Major Design Decisions:

There was some discussion on how a user would be able to call for instances of the various Actions we've made. The main choice was between the Command Design pattern and the Factory Design pattern. We ended up choosing to go with the Factory design, mainly because the way the Command Pattern worked didn't seem to mesh well with the online aspect of our project.

The second major decision we had to make was with the Record class. Initially we had the Record class be abstract with two classes extending it, DepositRecord and WithdrawalRecord. As the Action classes were coded in, we realized how useless it really was to keep deposits and withdrawals as separate records, as they are just meant to be used to show the user their "transaction history". Thus, we decided to completely get right of both DepositRecord and WithdrawalRecord and instead make Record a useable class, where withdrawals being negative values and deposits being positive values

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 and input their data to create an account. This will call the controller ActionFactory to 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.

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. 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.

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.

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.

(*)Note this exception:
Currently ClientUserInterface violates Single Responsibility Principle because if you change the login function or create a new user function or deposit, you need to make changes to ClientUserInterfacein all cases. This violates that a class should have only one reason to change.

ClientUserInterface also violates Open/Closed Principle because when you need to add a new function, for example view non depositable assets, you need to add a new method in ClientUserInterface class to deal with the new function, this violates that the class is closed for modification. These violations will be fixed by refactoring the ClientUserInterface given more time. The plan to refactor this class is for each method inside the ClientUserInterface that is dealing with an user action, have a separate class that is responsible for prompting messages and generating an ActionRequest and a separate class that takes in an ActionResponse and is responsible for presenting the output. And have the ClientUserInterface call a factory that takes in a user input and returns a corresponding ActionRequest. Then ClientUserInterface sends the request to the server and after receiving the response from server, it passes the response to the classes that present the response.

Description of Packaging Strategies:

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 Patterns:

- Template Method (Pull request #68): The template method design pattern was used in the Simulate abstract class and SimulateDepositable and SimulateNonDepositable subclasses. The template method getSimulatedAssetValue() is used to calculate the simulated asset value for either a Depositable or NonDepositable object. This design pattern was used because calculating the simulated value was similar for both types of assets. So the duplicated code was moved to an abstract class and the remaining calculations were moved into the subclasses.
- (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:

- Class name Expenses was changed to AddExpense for clarity

- Int values in Budget and Owner were changed to be double values
- SpendingRecord and DepositRecord classes were replaced with Record class -> So we refactored Owner by replacing list of SpendingRecord and list of DepositRecord with just one list of all Records
- Refactored controller class (Server) to use ActionFactory and call use cases (subclasses of Actions) through an abstract class (Actions), removed dependency between controller (Server) and entity(OwnerRepository), moved OwnerRepository to entity package

Progress Report:

- **Questions:**
  - What is a good method to test the server?
  - When is better to use different storing types like JSON, CSV, Database etc…?
- **What Worked Well In The 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 (ClientUI) and the actions could be handled easily using the design patterns (Factory) that we leaned towards.
- **Current Contributions and Future Plans for Each Member:**

Name	**Plan**
Aliyan	Contributions:Worked on making the server able to handle multiple users at a time. Implemented FinancialAsset abstract class and its subclasses. Implemented the date class because Java.util.date was not suitable for our purposes Worked with Ruiting to fix CashOut, DisplayDepositRecord, DisplayWithdrawRecord, OwnerInfo, ChangeBalanceSaving, UpdateDepositable, ViewInvestments actions and their response and request; and create CreateBond and CreateSaving actions and their response and request Future Plans: Working on use cases and possibly improving the way the server

	interacts with the client. Add more financial assets
Anita	Contributions: Implemented Transfer, TransactionHistory, UpdateDepositable, ViewInvestments, and Record. Refactored Owner and CRC files. Worked on test cases. Future Plans: Expand on more action classes and make adjustments to current Action classes based on changes in Owner class
Bernard	Contributions: - Implemented Budget, Owner, DepositRecord, SpendingRecord Future Plans: - Help to implement Actions
Leya	Contributions: - Implemented Withdrawal, DisplaySpendingRecord, DisplayDepositRecord, Simulate, SimulateDepositable, SimulateNonDepositable - Made response and request response for corresponding actions. - Made tests for the classes listed above. - Updated CRC cards for new classes - Worked on design document Future Plans: - Help to implement UI, Expand on Action classes
Linh	Contributions: - Implemented CashOut, OwnerInfo, Deposit, AddExpense, and CompareBudget. Implemented the corresponding test cases, response classes and request classes to these Action classes - Contributed to ActionFactory code

	- Added new CRC cards for classes mentioned above and some misc. Classes that had empty/missing crc cards - Worked on Design Document Future Plans: - Expand on more action classes and make adjustments to current Action classes based on changes in Owner class
Miguel	Contributions: - Implemented basic GUI (in separate branch (GUI) for now because they are not effective towards towards Phase 1 functionality) that will be connected to work in Phase 2 will be created using Builder design pattern - Data Storing via JSON using DataStore request /response and action - JSONtranslator - BugFixes Future Plans: - Tests for all classes related to data storage and interaction - Will add ways to store data besides just OwnerRepo Objects (JsonTranlator can store any object but ClientUI requires implementation) - Finish GUI Design - Move data storage to server computer rather than user computer - -Typechecking for JSON
Ruiting	Contributions: - Implemented ActionFactory, ActionResponse, LoginResponse, CreateUserResponse and corresponding tests for them - Implemented factory design pattern in ActionFactory - Refactored controller class (Server) to use ActionFactory and call use cases (subclasses of Actions) through an

	abstract class (Actions), removed dependency between controller (Server) and entity(OwnerRepository), moved OwnerRepository to entity package - Worked with Aliyan to create Bond, FinancialAsset, Date and corresponding tests for them - Worked with Aliyan to fix CashOut, DisplayDepositRecord, DisplayWithdrawRecord, OwnerInfo, ChangeBalanceSaving, UpdateDepositable, ViewInvestments actions and their response and request; and create CreateBond and CreateSaving actions and their response and request Future Plans: - Refactor ClientUserInterface to make it adhere to the SOLID principles
Tiffany	Contributions: - Implemented Budget, Owner, DepositRecord, SpendingRecord - CRC cards on entities Future Plans: - Creating budget type actions