Phase 1

Specification

Things highlighted in green are specifications we have changed or added

Entities: Spread, Card, User, Deck, ReadingLog, Reading

- Spread: Spread object that holds spread information, every spread has a number of cards, but only some spreads have meanings associated with card position in the spread, and these spreads will use a 2D array to hold possible card positions and their associated meanings.
- Card: Card objects that hold card information. Every card has a number, name, isReversed, isMajor, and meaning associated with it. Some cards have an additional attribute called suit which is one of the following values: Wands, Swords, Pentacles, Cups. These suits have an additional general meaning associated with them.
- User: User objects store all user information. Every user has a username, birthdate, ReadingLog, and password associated with their account.
- Deck: Deck objects hold an arraylist of card objects.
- ReadingLog: Holds reading log objects associated with a user. The reading log is empty when a user first creates their account, and contains all user logged reading objects.
- Reading: an object that holds a reading generated by ReadingGenerator. This reading contains card meanings based on chosen spread. Implements Serializable interface. It will have a toString method that prints the reading. toString method could be made

UseCases: ReadingLogger, ReadingGenerator, UserGenerator, CardInit, SpreadInit

- ReadingLogger: Mutates reading log objects based on user input. Also implements Serializable.
- ReadingGenerator: Responsible for shuffling deck, and getting picked cards, and spread based on user input. It has a shuffleDeck method that shuffles and returns a shuffled deck object. It has a pickCard method that returns an array list of picked cards. It also has a getSpread method that will return the spread the user has chosen. The ReadingGenerator will also be responsible for getting cardMeanings for each card in the pickedCards and tailoring their meaning to the spread that was chosen and stringing them together to create a Reading object.
- UserGenerator: responsible for creating a user object based on input coming from the UserManager. It will implement the getUser method for UserManager to view user information.
- CardInit (implements DataReader): Creates card objects from data read by DataReader and adds them to arraylist object in Deck called deck
- SpreadInit (implements DataReader): Creates spread objects from data read by DataReader and adds them to arraylist object in Spread called spreads

- Login: Responsible for user login.

Controllers: DataReader, CommandLineInterface, ReadingLogManager, UserManager

- DataReader: Will have a method that takes a file name as an input and is called by SpreadInit and CardInit to read data from csv files and construct the appropriate objects.
- CommandLineInterface: Will take user input and send it to appropriate use cases.
- ReadingLogManager: Takes user input to add or remove readings, as well as access the users current reading log.
- UserManager: Will take inputs from the user. It will take all the information from the user including birthday, gender, username, password, etc. Will be able to view user info from UserGenerator through the method. UserManager will implement Serialization to keep the information of the users.

Major Design Decisions

- Our group decided to get rid of some classes as some of them were unnecessary and some were violating the Clean Architecture Principles. Shuffler, CardPicker, and ShuffleManager classes are removed from our design as of Phase 1. We implemented some of the classes we removed as methods under other classes.
- We made the Reading class serializable.
- We will make the Spread class an abstract class in phase 2.
    - The Spread class will have a constructor that calls the method createGrid() which will initially create a 0x0 matrix. fillGrid() will fill the empty matrix according to the spread that is selected by the user.
- We will implement a class named DataReader to do the file reading for the card and spread data we use to implement the card and spread objects.
- Command design pattern is a design pattern that we've already planned to include as a part of our final project. We didn't have the chance to implement the command class in Phase 0, as our team members didn't have any idea on how it works and how we can implement it in our program. The program that we've been working on gets many requests from the client and deploys these requests on a real-time basis. Thus, the Command design pattern would act as a great approach to decrease the complexity of the program throughout Phase 1. This will also create room for us to expand the code and possible functionalities in the future without changing the base code.
- We renamed the TarotReader class to ReadingGenerator

How our project adheres to clean architecture and violations we have found

Our initial CRC model had a lot of these types of violations since there were instances where controllers were collaborating with entities and vice versa. We used the Dependency Inversion principle to fix these issues.

We created a DataReader class at the controller level, and this class contains a method that will read data from the CSV constants file and will be called in CardInit or SpreadInit to create card and spread objects. We also added a CommandLineInterface controller which is going to read input from users and send it to the appropriate use case, which will then return the information back to the controller for output to the user. We also added a UserGenerator and ReadingLogger use case to make sure that the Dependency rule was followed. The UserGenerator takes input from the CommandLineInterface in order to construct a User object, and the ReadingLogger will mutate the ReadingLog object based on CommandLineInterface input. Lastly, we added a Reading object, this makes it serializable and also allows the ReadingGenerator use case to mutate Reading entities.

How our project adheres to SOLID Principles of design and violations we have found

Single Responsibility Principle: Our design was built on this principle from the start, until the point where we recognised we had developed some unneeded classes. After going over each class, we double-checked that each class in our packages has just one responsibility and that there were no ghost classes.

Open/Closed Principle: Our design is open for extension in most of our classes as they are able to extend to subclasses and interfaces. However, these classes are close to modification as they should be.

Liskov Substitution Principle: We modeled excellent inheritance hierarchies using the Liskov Substitution Principle. Objects of our superclasses can be replaced by objects of their subclasses without disrupting the program.

Interface Segregation Principle: The Integration Segregation principle makes our code more readable and maintainable. We've pared down our class implementation to only the methods that are required, with no extra or extraneous code.

Dependency Inversion Principle: Our design in Phase 0 had instances where controllers were collaborating with entities. After realizing that this fails the Dependency Inversion Principle, we made changes to the way classes interact with each other, as detailed above in the clean architecture section.

Packaging Strategies Considered

The packaging strategies considered were
- By component
- By layer

After consideration we decided that packaging by layer would be the better choice. The reason was that packaging by layer provides better organization for the classes, since we can think about what layer each class is in, and organize accordingly. This would also help with keeping the code within the clean architecture principles.
To use packaging by component, we would have to group classes that are related to a component, but we felt this was more prone to error.

Design Patterns we plan to implement
- Decorator design pattern for spread class, we will implement this in phase 2 as we will need this for the 2d array/grid we discussed in the major design decisions above.
- Builder design pattern in phase 2 to build the different picked cards into spread that the user chose for the GUI
- We are considering implementing an iterator design pattern, as there are multiple times we iterate over different objects that shouldn't be seen by the user or client.

Progress Report
1. Open Question to the TA
    - For DataReader we have two options, a bufferedReader and Scanner implementation, which do you think is better?
2. What has worked well with the design

Because there were situations where controllers collaborated with entities and vice versa, our first CRC model contained a number of these sorts of breaches. To solve these problems, we applied the Dependency Inversion concept.
At the controller level, we established a DataReader class, which has a method that reads data from the CSV constants file and is invoked in CardInit or SpreadInit to construct card and spread objects. A CommandLineInterface controller has also been introduced, which will receive user input and pass it to the appropriate use case, which will then return the information to the controller for output to the user. To ensure that the Dependency rule was observed, we included a UserGenerator and ReadingLogger use case. The ReadingLogger will change the ReadingLog object depending on CommandLineInterface input, and the UserGenerator will accept input from the CommandLineInterface to generate

a User object. Finally, we introduced a Reading object, which allows the ReadingGenerator use case to change Reading instances.
In addition, we have an excellent organization for our classes as we chose packaging by layer, and our code adheres to the SOLID Design and clean architecture principles.

3. Summary of what group members have been working on and plan to work on
    - Shauna: For this phase, I collaborated with Jai to implement the DataReader class, which is a controller that reads data from a text or csv file. We then made the appropriate changes to SpreadInit and CardInit, which are not use cases to use the data reader to get the appropriate information for its respective entities and initialize them. Added isReversed functionality to ReadingGenerator's shuffle method. Worked on cleaning up Spread, Card, Deck, and added all test cases for the methods I worked on. For the next phase, I would like to add more variety to the readings, to make them more specific to the spread chosen. I would also like to work on implementing the GUI.
    - Elif: I worked on ReadingLog, ReadingLogManager, User, and a small portion of the Command Line interface, with Bora. While working on these classes, my main goal was adding more features to the program. We all collaborated on the design document, the presentation, and the progress report. For Phase 2, I would like to work on the GUI.
    - Yigit: For this, I worked on the Reading and readingGenerator classes, which are mainly responsible for generating the reading through the Reading object by using the inputs gathered from the user. In the last phase, the readingGenerator was generating readings on random. With the new methods and the developments, readings are generated according to the input given by the user.
    - Murat:In this phase I worked on the LogIn, GenerateUser and ReadingLogger classes as well as helping with the redesigning of the project to make it work with the clean architecture principles. The classes I worked on handle the cases of account creation, logging in and logging readings into the logged in user's reading log.
    - Jai: This phase, I worked with Shauna to create the DataReader class, which is a controller that reads data from a text or csv file. We then made the necessary adjustments to SpreadInit and CardInit, which are not use cases for using the data reader to obtain and initialise the proper information for their respective entities. I also worked with Murat to understand how user data can be serializable and wrote the respective tests for Serializing and Deserializing data, which will be useful for future implementations of the next phase.

- Bora: I worked on ReadingLog, ReadingLogManager, User, and helped plan out the Command Line Interface with Elif. My main goal while working on these programs were improving on them and making them more applicable to the Tarot Reader concept. We wanted to improve on the logging functionality of our code and made the logs more reachable by the user.
- Bahati: For this phase I have been working on the implementation of the CommandLineInterface(the program engine) which takes inputs from all the UseCases and the appropriate methods in order to run the program. I plan to continue working on adding more functionality to the CommandLineInterface, or try working on other classes so that I can have a clear image of how the program is built from all other angles.