PHASE 1 DESIGN DOCUMENT
Group 57: Jun, Akansha, Chris, Koji, Iris, Wei

Specification:
Our project is a memory matching card game that allows the players to log in with their credentials (case-sensitive) and play.
While in the program, players can select the level of difficulty. Easy mode displays 6 pairs of tiles in a 3x4 dimension. Medium mode displays 10 pairs of tiles in a 4x5 dimension. Hard mode displays 15 pairs of tiles in a 5x6 dimension. Extra hard mode displays 21 pairs of tiles in a 6x7 dimension. Players are also prompted to select a visual theme for the cards. (A random mode disables these options and generates a random set of cards).
The game has a timer and a move counter, to track the time and the number of moves that the player has spent in a game mode. It will also have a score counter to display the number of matches the player has currently made. There are also options to pause and reset the game mode.
The game should also have a leaderboard that records the top 10 highest scores among all players. Each difficulty has its own leaderboard. The leaderboard is based on the lowest total number of moves by a player. If there is a tie, time is used as the second criterion.

Additional Functionality:
- Entities:
    - User: object, with username and password attributes
    - TileBoard: object, with attributes related to the tile position
    - Tile: object, contains key and boolean flipped indicating if the tile is flipped.
- Use Cases:
    - BoardGenerator: generates a tileboard according to user input.
    - BoardManager: performs tasks like flipping tile and checking for matched pairs.
    - UserManager: performs tasks like creating users, changing user's username and password.
    - Difficulty: A set of difficulties for the tileboard set up.
    - Theme: A set of themes containing the tile deck pictures for the tileboard set up.
- Controllers:
    - MainMenu: Options of starting a game, going to the leaderboard, or exiting the program after the user logs in.
    - GameSettings: Options to pause, reset or exit the game mode.
    - UserGameInput: Obtains the user's input during a game mode.
- UI:
    - Game: Runs the actual game.
    - LoginPage: a user enters their username and password. The user database is checked to see if the username and password match, and a boolean is returned
    - SignUpPage: a new user enters a username and password. The user database is checked to see if the username exists, and a boolean is returned. The new user's information is added to the User Database

- Databases:
    - UserSQLDatabase: a table for users, which will contain a unique username and password for each user
    - GameHistorySQLDatabase: a table for game history, which will contain a unique Game ID, score/total moves, and time for each game
    - LeaderboardSQLDatabase: a table which shows the top 10 games of whatever difficulty is passed in (with ties)

Major Design Decisions:

- Using two PostgreSQL JDBC tables to keep track of users and game histories respectively
- Using a PostgreSQL JDBC database for the leaderboard instead of a leaderboard object
- Organizing with Clean Architecture

SOLID:

- Single Responsibility Principle: each class has one overall responsibility. However, we could possibly separate them into further individual responsibilities, such as our UserGameInput class into smaller classes that take input.
- Open/Closed Principle: When implementing our code, we kept in mind the open/closed principle to make our code as extendable as possible. We did this by limiting dependencies between classes while following clean architecture and having one way dependency. Our hopes are that this will make refactoring much more simple for phase 2, and that we will not have to spend so much time deleting and manually changing code.
- Liskov Substitution Principle: Our game does not require any use of inheritance or use of child and parent class so it does follow this principle.
- Interface Segregation Principle: We implemented some of our code in private and some code in public. That way, clients would not be forced to implement irrelevant methods. For example, we made our final Strings in GameHistorySQLDatabase private since we do not want the client to be able to just look up the password or username. Same goes with the LeaderBoardSQLDatabase and UserSQLDatabase.
- Dependency Inversion Principle: When we implemented the code, we made sure that high-level modules do not depend on low-level modules. Also, we added abstraction layers such as databases.

Clean Architecture:

- Entity: Tile, TileBoard, User
- Use Case: BoardGenerator, BoardManager, UserManager
- Controller: MainMenu
- UI: Game, SignUpPage, LoginPage
- Database: GameHistorySQLDatabase, LeaderboardSQLDatabase, UserSQLDatabase
- Our database tables are an example of how the program interacts with an outer layer of clean architecture. The database does not interact with any lower level of the program, so any changes to the databases do not affect the entities, use case, etc. classes. We have two

tables in our database (one for User, one for Game History), each which do not affect their respective sections.

Packaging Strategies:

- Packaged by Layer (Clean Architecture Layer/category)
- Entity, Use Case, Controller, UI, Database
- Also considered by feature, but packaging by layer was more organized and help us adhere to Clean Architecture and separating classes
- Was also easier to check for dependencies

Design Patterns:

- Decorator Pattern (Jun), this is for the incorporation of difficulty and theme: https://drive.google.com/drive/folders/1WehQL0W7A6-JMJFllp8vaaGnh4ZaDECR
- It is hard to implement the theme decorator in phase 1 since we are only running the game program in the terminal and not on a GUI yet, so we plan on implementing this in Phase 2 as Jun designed it and wrote about in the Design Question 1 document linked above.

Use of Github Features:

- Pull requests have been working well for Jun/Akansha/Chris
    - One person codes and the others review the code and merge the request
    - We usually code together (as Jun's computer is the one with the SQL database set up) but it is still good to individually go over all the changes in Github's pull request feature
- Writing a commit message detailing the tasks done in each commit with the names of the participating members.
    - As mentioned above, our members work on the SQL part together with my (Jun's) computer screenshared so it is necessary to give credits to all the members in the comment.
    - Writing these commit messages is also a good practice to keep track of the progress and to be able to go back to a previous version if anything goes wrong.

Code Style and Documentation:

- Used Pull Requests so that there are no incorrect merges, double checked everyone's work
- Used header comment for every class with clean architecture layer and class description.

Testing:

- JUnit Tests for UserSQLDatabase (written in class in Pull Request #13, moved to actual JUnit in Pull Request #14)
- JUnit Tests for GameHistorySQLDatabase (Pull Request #16)
- JUnit Tests for BoardManager (Pull Request #22)

Refactoring:

- Changed Hashtable to SQL Database (Pull Request #12)
- Updated user login and signup page to SQL (Pull Request #14)
- Changed leaderboard to SQL Database (Pull Request #15)
- Removed some unnecessary classes that we had in Phase 0/CRC Model (Pull Request #10, #16, #22)

Code Organization:

- We have organized our code mainly by Clean Architecture level/description ex. All databases in the same package
- All classes have a green commented minor description of what they do.
- Line comments on some of the more complicated method code so that everyone can understand.

Progress Report:

Open Questions:

- How can we incorporate sound effects?
- How to save changes to the tables in IntelliJ to the local database? Autocommit for JDBC is on but it seems to have no effect on the local database (when accessed in command prompt) outside of the IntelliJ client.
- Any suggestions/advice on learning html coding if we were to implement Spring boot for the next phase?
- What clean architecture layer should the decorators classes be in?

What Has Worked Well:

- Wrote a SQL query that automatically updates the leaderboard for top 10 of each difficulty so there is no need to write a function to update.
- Packaging according to clean architecture, this way we can check if any dependency is violated.

Individual Summaries:

Note: in our pull requests and commit logs, we wrote down in brackets the names of the people who worked on that code over screen sharing or in person

Jun:

- Installed postgresql software on a local pc first, set up the database on the local pc, imported the connector into the class path, set up the local postgresql connection
- Replaced the old hashtable methods in UserDatabase class with SQL database in a new class called 'UserSQLDatabase', completed UserSQLDatabase
- Added GameHistorySQLDatabase class, moved UserSQLDatabase tests into JUnit test class
- Wrote LeaderboardSQLDatabase
- Writing JUnit tests for UserSQLDatabase, GameHistorySQLDatabase, BoardManager

- Updated and Fixed LoginPage and SignUpPage with SQL
- Updated the main method in Game
- Designed packaging strategy
- Progress Report (what I wrote): Additional Features, Clean Architecture, Design Decisions, Packaging Strategies, Open Questions, Github, Testing, Refactoring, Design Patterns, What has worked well

Akansha:

- Installed postgresql software on a local pc first, set up the database on the local pc, imported the connector into the class path, set up the local postgresql connection
- Replaced the old hashtable methods in UserDatabase class with SQL database in a new class called 'UserSQLDatabase', completed UserSQLDatabase
- Wrote LeaderboardSQLDatabase, SignUpPage, LoginPage
- Helped with refactoring and separating the original code from our demo into the separate classes (Tile, TileBoard)
- Writing JUnit tests for GameHistorySQLDatabase and BoardManager
- Wrote UserDatabase and GameHistoryDatabase (later updated to SQL databases)
- Designed packaging strategy and packaged all classes
- Fixed main method in Game, UserLogin, SignupPage
- Progress Report (what I wrote): Additional Features, Clean Architecture, Design Decisions, Packaging Strategies, Open Questions, Organization, Github, Testing, Refactoring

Chris:

- Writing JUnit tests for, GameHistorySQLDatabase
- Wrote LeaderboardSQLDatabase, SignUpPage, LoginPage
- Wrote UserDatabase and GameHistoryDatabase (later updated to SQL)
- Wrote SignUpPage and LoginPage
- Designed packaging strategy
- Helped with refactoring and separating the original code from our demo into the separate classes (Tile, TileBoard)
- Progress Report (what I wrote): Additional Features, Clean Architecture, Design Decisions, Packaging Strategies, Open Questions, Github, Testing, Refactoring

Koji:

- Refactored original code from phase 0 into separate Tile and TileBoard classes
- Finished runGame method in Game main method (using Move, UserGameInput.getUserDifficulty, helper methods)
- Added statistics for each game (time, number of moves, difficulty, username, game id) and added to GameHistory database for user
- Wrote test cases for BoardManager
- Incorporated the implemented login and signup methods into Game
- Wrote SOLID principles for the progress report.

Iris:

- Implemented exceptions for invalid inputs for class UserGameInput (UserGameInput, getUserDifficulty)
- Worked on runGame method in Game main method (using Move, UserGameInput, getUserDifficulty, helper methods)
- Worked on incorporating implementation of login and signup methods into Game
- Worked on statistics for each game (number of moves, difficulty, username, game id)
- Wrote SOLID principles for the progress report.

Wei:

- Worked on main and loginOrSignup method in Game class.
- Refactored code (User class).
- Added change to SignUpPage class.
- Worked on BoardManagerTest.