PHASE 2 DESIGN DOCUMENT

Group 57: Jun, Akansha, Chris, Koji, Iris, Wei

Scenario Walk-Through:

For the scenario of logging into the game:

1. When the main method is run, a blank user is created and is then passed into LoginOrSignUpPage
2. User input is taken in from LoginOrSignupPage on whether the User wants to login, signup, or play as a guest
3. If the User wants to login to an existing account, the code moves to LoginPage
4. LoginPage uses userManager to check credentials
5. UserManager accesses the SQL Database to check the credentials, and returns back to LoginPage
    a. If the credentials are correct, the code moves into StartPage
    b. If the credentials are incorrect, a message telling the user to try again shows on screen, and a new LoginPage is created

Additional Functionality (Phase 2):

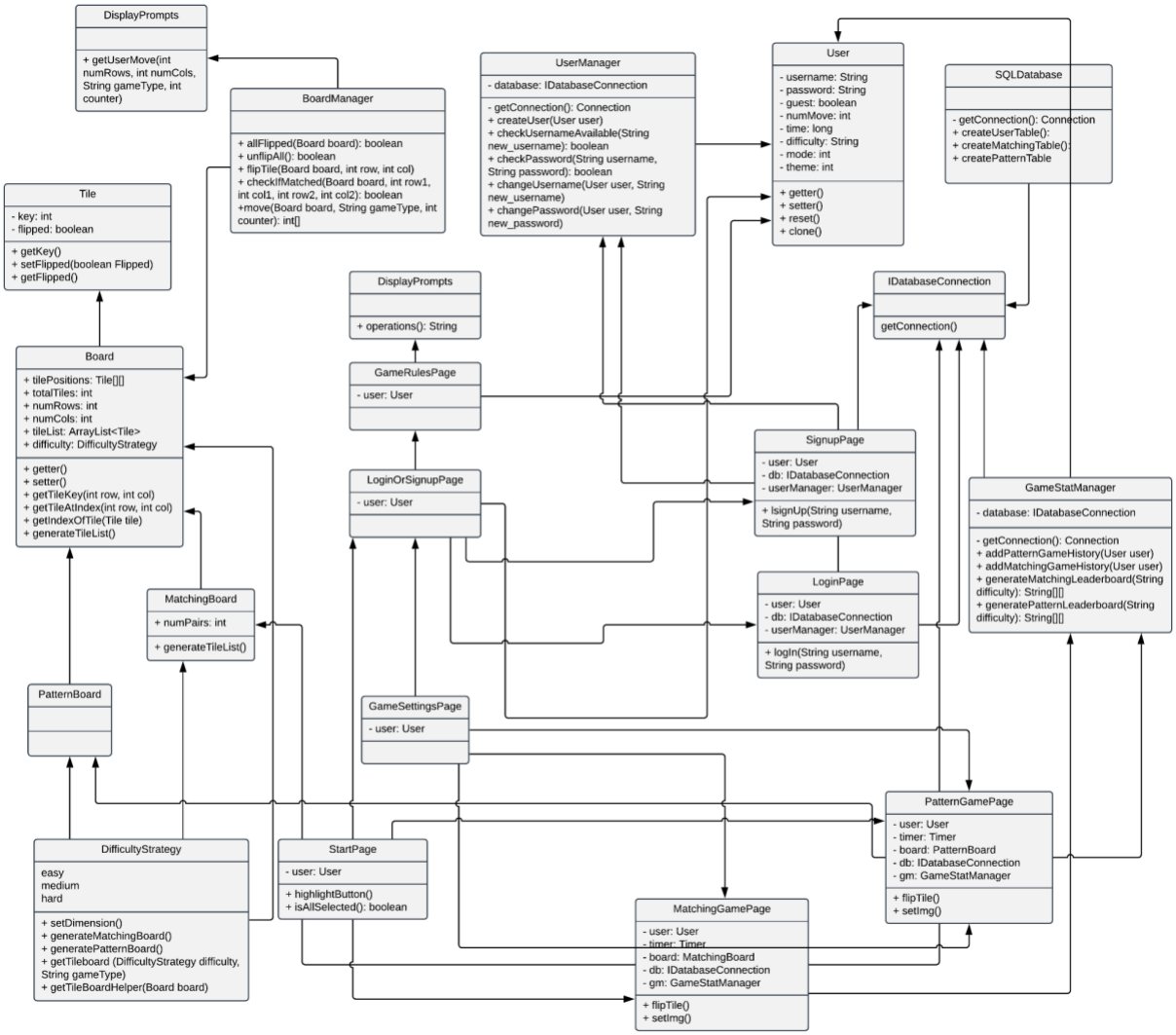
- Board: parent class object, with attributes related to the tile position
- MatchingBoard: inherits from Board, has an attribute where the tiles in the board each have a pair (based on their Key) –> note, was TileBoard
- PatternBoard: inherits from Board
- DifficultyStrategy: a strategy design pattern which generates a board
- DisplayPrompts: all print messages are here, and this class is called whenever something must be displayed to the user
- PatternGame: Logic for Pattern game
- MatchingGame: Logic for Matching Game –> was runGame
- StartPage: where the user inputs difficulty, game type, and theme
- PatternGameHistorySQLDatabase: a table for Pattern game history, which will contain a unique Game ID, and time for each game
- PatternLeaderboardSQLDatabase: a table which shows the top 10 Pattern games of whatever difficulty is passed in (with ties)
- GameSettingsPage: where the user can restart or resume the game
- GameRulesPage: where the user can read the rules of matching and pattern game
- IDatabaseConnection: abstraction layer for the connection of database.

Major Design Decisions (Phase 2):

- Using a strategy design pattern for Difficulty
- Adding the second Pattern Game mode –> this involved creating a Board parent class and two new classes for PatternBoard and MatchingBoard (originally TileBoard). Also had to change runGame to MatchingGame and create a new class PatternGame for the Pattern Game logic

- Adding new SQL Databases, PatternGameHistorySQLDatabase and PatternLeaderboardSQLDatabase
- Using SWING GUI
- A guest feature so the user does not have to create an account to play the game.
- Added IDatabaseConnection for the implementation of DIP.

UML diagram:

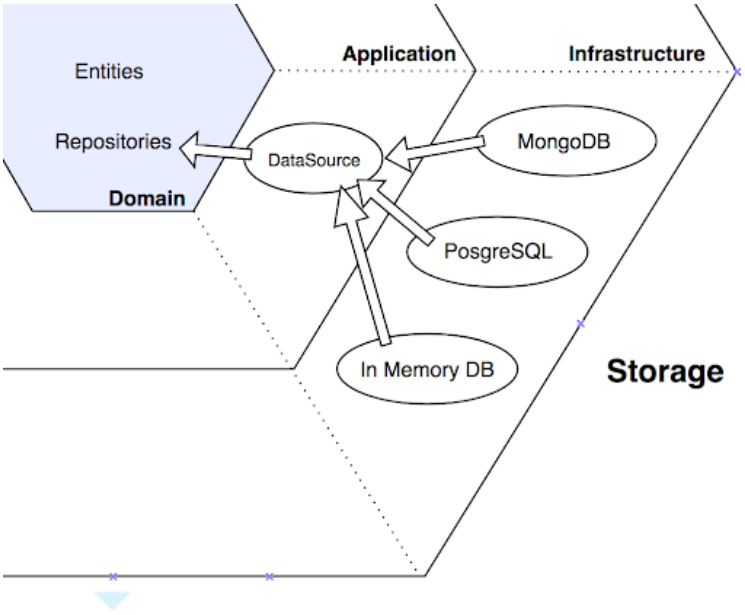


SOLID:

- 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. Furthermore, we used the strategy design pattern in our DifficultyStrategy class which implements this principle.

- Liskov Substitution Principle: Liskov Substitution involves subclasses, inheritance, and superclasses. In our case, we have the superclass Board. Board is a game board which contains tiles, which has various attributes (such as the tileList of tiles, difficulty, the tilePositions, and the dimensions) as well as methods (getting a tile at an index, returning the difficulty of the board). For our project, we have two game modes: Matching and Pattern. Both games require boards that are very similar, except that the Matching Board needs a list of tiles that is different from Pattern Board. Matching requires all tiles to have a pair key, and Pattern requires all the tiles to have different keys. We decided to use inheritance, since there is only one method that is different between the two, and that is GenerateTileList. Both PatternBoard and MatchingBoard are Boards, but not every Board is a PatternBoard (or a MatchingBoard).
- Interface Segregation Principle: ISP states that clients should not be forced to implement interfaces they don't use. Therefore instead of one big interface, many small interfaces should be used based on groups of methods to avoid polluted interfaces. In our project, we did not implement any interface except for the IDatabaseConnection interface explained below in DIP. Therefore this principle is not applicable to us. However, we did make sure that each class served a separate submodule of function.
- Dependency Inversion Principle: For our project, we had a design that looked similar to the diagram below:



(Source:http://rubyblog.pro/2017/07/solid-dependency-inversion-principle)
Our outer layers depended on the inner layers, and not the other way around. So we had a one-way dependency.

We added an interface for example, called “IDatabaseConnection” that serves as an abstraction layer for the UserManager data for example. Then we can inject IDatabaseConnection into UserManager via constructor. Create a class called “SQLDatabase” that inherits the database interface. This way we would have both the higher level module (UserManager) and Lower Level module (SQLDatabase) depending on the same abstraction.

Clean Architecture:

- Entities: User, Board, MemoryBoard, PatternBoard, Tile, DifficultyStrategy
- Use Cases: BoardGenerator, BoardManager, UserManager
- Controllers: MainMenu, GameSettings
    - 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.
- Views: DisplayPrompts, UserGameInput, PatternGame, MatchingGame, UserLogin, StartPage SignUpPage, UserLogin, LoginOrSignup
- Gateways/Databases: UserSQLDatabase, *PatternGameHistorySQLDatabase*, MatchingGameHistorySQLDatabase, *PatternLeaderboardSQLDatabase*, MatchingLeaderboardSQLDatabase
- 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

Packaging Strategies:

- Packaged by Layer (Clean Architecture Layer/category)
- Entity, Use Case, Controller, Views, Gateways.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:

- Strategy Design Pattern was used for Difficulty.
    - For this design pattern, we coupled it with the implementation of enum. We created a DifficultyStrategy class to store the different strategies/difficulties as enums. For each enum we generate a TileBoard of the assigned dimension according to the difficulty.

Use of Github Features:

- Issues:
    - Raising issues on Github to talk about any potential problems we encounter in the code as well as dividing work and code refactoring
    - Commenting on issues so the group can see the full updates
- Pull Requests:

    - One person uses a branch, codes, makes a pull request, and the others review the code and merge the request
    - We ensure the person coding is not the same one merging the pull request
    - Have been trying to do more individual coding, but we still write 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 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 javadoc header comment for every class with Clean Architecture layer and class description
- Used javadoc comment for the methods as well with tag 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)
- JUnit tests for Board attributes dependent on the DifficultyStrategy difficulty the board was generated with (Pull Request #52)

Refactoring for Phase 2:

- Changed difficulty to strategy design pattern, the representation of difficulty changed from integer to String
- Display the game with swing GUI instead of reading from/writing to console
- Added a second game mode (PatternGame):
    - Board object parent entity which had child classes for MatchingBoard (our original game) and PatternBoard (our new game mode)
    - PatternGame class which contains the logic for the second game mode
    - Separate SQL Databases for the PatternGame game history and leaderboard
    - A more complex Start Page, as user now has to choose game mode as well as difficulty and theme
- Changed DisplayPrompts to return instead of void/print, making it easier for GUI
- Changed the 4 SQL database classes to instead be 1 SQL database class

Code Organization:

- We have organized our code mainly by Clean Architecture level/description. Ex. All databases in the same package
- All classes and their methods have javadoc descriptions of what they do.
- Line comments on some of the more complicated method code so that everyone can understand.

Progress Report:

Open Questions:

- For accessibility, we were wondering if it would be possible to implement some sort of screen-reader so blind and visually impaired people could still play the game.
- Converting this into a mobile game format so people can also play on their phones

What Has Worked Well:

- PatternGame, it is a good implementation of inheritance as well as increasing the scope of our project
- GUI is very effective in making the game more interactive and visually appealing for players
- A guest mode so the player has the option to not sign up and still be able to play the game

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:

- Significant pull request: #28 (implementation of design pattern), #60-62 (DIP implementation, login+signup on GUI, game stats on GUI), some bug fixings and design changes with Akansha stated below
- Finished PatternGamePage and LeaderboardPage.
- Created IDatabaseConnection for the abstraction layer. Restructured the SQL classes and moved methods to Manager classes.
- Log-in and sign up functions on GUI with database.
- Created the second game mode. Created a parent Board class so MatchingBoard and PatternBoard all inherit this parent class. For the game logic: created two separate classes for two game modes. Moved the actual main method to a new class called “main”.
- Separated the SQL classes for the two game modes: instead of the original LeaderboardSQLDatabase and GameHistorySQLDatabase classes, we now have 2 database classes for each game mode.
- Implemented strategy design pattern with enum for the difficulties. Moved the helper methods from BoardGenerator into this enum class and the TileBoard class, so BoardGenerator can be deleted now. The type of difficulty is now String instead of int when running the game so it is easier to tell.
- Fixed javadoc comments for the phase 1 classes.

- Design Document: Major Design Decisions (Phase 2), Design Pattern, SOLID, Code Style and Documentation, Phase 2 Refactoring, Accessibility Report, UML

Akansha:

- *Significant Pull Requests: 35-37 (INDIVIDUAL, creating a start page and editing the workflow, fixing errors), 12 (SQL Databases with Jun), 30-31 (Pattern Game Mode with Jun, as well as designing this new game mode)*
- Renamed packages, fixed naming errors with allFlipped, moved classes according to Clean Architecture packaging
- Edited Game and UserGameInput to have the new class, moved LoginOrSignUp method from Game class into its own class LoginOrSignup.java, adding javadocs
- Created a basic startPage and added it in the middle of the program flow
- Helped create and code the second game mode (a parent Board class so MatchingBoard and PatternBoard all inherit this parent class). For the game logic: created two separate classes for two game modes
- Moved the actual main method to a new class called “main”.
- Separated the SQL classes for the two game modes: instead of the original LeaderboardSQLDatabase and GameHistorySQLDatabase classes, we now had 2 database classes for each game mode
- Changed DisplayPrompts to return, added JLabels in GameSettings for MatchingRules and PatternRules
- Worked on implementing PatternGame with GUI, changing GameStatManager methods to return type instead of void type, worked on rough MatchingLeaderboardPage
- Javadocs, cleaning up code/deleting classes (such as the old SQL classes, since they were combined into one new class),
- Design Document: Specification, Additional Functionality for Phase 2, Major Design Decisions for Phase 2, Clean Architecture, Phase 2 Refactoring, SOLID, Accessibility Report, What Has Worked Well, Open Questions, Scenario Walkthrough

Chris:

- Significant Pull Requests : 50(Created GameSettingsPage and fixed some errors)
- Created GameSettingsPage for GUI
- Renamed Packages
- Helped with implementation of GUI in StartPage, PatternGamePage, MatchingGamePage and GameSettingsPage
- Helped implement MatchingGameLogic and GUI bug fixing
- Design of GUI and different themes in GUI
- Changed picture file types to jpg for cards
- Rephrase input methods: collect User’s input from UGI instead of scanner in GameSettingsPage
- Optimize game structure (Easier to maintain)

- Implementing GUI for PatternGame, different display as it shows you the pattern and then disappears
- Helped with implementation of controller of MatchingGame and PatternGame
- Helped implementation of Game logic for PatternGame

Koji:
- Significant pull request : PR#52 (Created tests for new classes)
- JUnit testing on new PatternBoard and MatchingBoard classes
- Changed picture file types to jpg for cards
- Contributed to implementing PatternGame (controller)
- Collaborated to implement MatchingGamePage, PatternGamePage and Leaderboard GUI
- Tested PatternGamePage LeaderBoard GUI
- Cleaned up code: removing unused methods, java conventions, javadocs

Iris:
- Significant Pull Requests : 53(Created GameRulesPage and fixed some errors), 57(fixed GameSettingsPage and finished up GameRulesPage)
- Created GameRulesPage and added on LoginOrSignupPage
- Edited GameSettingsPage (added buttons Home, Restart, Resume and edited formats)
- Added setting button in MatchingGamePage and PatternGamePage
- Added home button on StartPage, GamesRulesPage
- Changed picture file types to jpg for cards
- Contributed on general GUI codes (fixing minor errors, dimensions, design)
- Contributed on fixing PatternGamePage, GameSettingPage, and MatchingGamePage
- Design Document : Additional functionality for phase 2, Testing

Wei:
- Individual pull request:#39 #40 #59
- Implement Guest mode (For testing, no database needed to run the code)
- Create and Implement GUI of StartPage, PatternGamePage, MatchingGamePage, LoginPage, SignupPage, LoginOrSignupPgae, LeaderBoardPage.
- Add features to User Class
- Implement controllers (MatchingGame, PatternGame).
- Design UI
- Implement game display logic in MatchingGamePage and PatternGamePage
- Rephrase input methods: collect User's input from UGI instead of from Java.Scanner
- Optimize game structure (Easier to maintain)
- Adding tile pictures to GUI