DESIGN Document for OOGASalad_team01

Ken, Sophie, Andy, Del, Shriya, Emily, Tim, Ted, Alec

Ned IDs:

al510, smw92, xw214, kdk33, tcp90, sc746, eys9, edc39, tdg24

Role(s)

- Ken
  - Co-designer of File Manager package
  - Bridge between File Manager and Runner subteam
- Sophie
  - Principal Runner View designer
- Andy
  - Principal Runner Controller designer
  - Co-designer for main and login functionality
- Del
  - Principal Runner Model designer/maintainer
  - Co-designer of unifying game structure
- Shriya
  - Co-designer for Runner Model
  - Co-designer for social hub and login screen
  - Floater to help with different parts of the team
- Emily
  - Principal designer for Builder MVC-loop and flow
  - Co-designer for login and social hub functionality
- Tim
  - Principal designer for Builder Phase Editor and flow (significant amount of work)
  - Designed complex custom information entry dialogs
  - Integrated Parser dynamic validation/entry options API

- Ted
    - Principal implementer for Runner ChatGPT assistant
- Alec
    - Co-designer/Maintainer of File Manager package
    - Principal designer of app-wide translation package (i18n) (Is mine)

Design Goals

Make a flexible game maker that can support multiple types card games. The program needs to allow users to design their own, unique card games in a no-code enviornment. The project is structured to maximize efficiency and design velocity.

The goal of this application is to design a program that can build a card game which can have rules that are configurable by a user and can be played by multiple players.

The rules of the game are configurable by the user through a `Builder` companion program. The user is able to create conditions to specify requirements for different `Stages` of each round, turn, and game, such as checking the number of cards in a players hand at the end of a turn to see if the player has won.

Flexibile Design

This design is very flexible because any number of games can be created and saved as a JSON file. `Builder` and `Runner` are also completely separate from each other and can be run independently of each other. Allowing games to be previously generated and run at a later time (or saved to the cloud using Firebase and run on a different computer).

The user can create a large variety of different games by changing the rules and conditions of the game and which stages are enabled or disabled.

High-Level Design

The project is broken into three parts.

- Builder

    The Builder provides an interface for a user to build and save game files which can be later run. The interface accomodates multiple language options, a custom casino theme, and dynamic rules, parameters, and validation schemas.

- Parser/File Manager/Translator

The Parser/File Manager loads and saves games and game templates. It also loads valid paramters, rules, and conditions for the Builder to offer to users to include in their custom games.

The i18n Translator manages Error message translation and mappings between user representationa and class resprestation of rules in games for the entire project. It acts as a layer of abstraction to restrict direct access to the resource files.

- Runner

  The runner provides an interface for a user to load and play game files. Creating a board and players that are governed by rules specified in a "game template". The Runner diplays controls that enable the user to complete allowed rules that have been defined by the builder.

Assumptions or Simplifications

Stages

Each of the parts of the game are represented in the Rules Builder as a `Stage`.

- `Game` - The game stage is the top level stage. It contains the game rules, the players, and the game controller.
- `Round` - The round stage contains the round rules and a turn
- `Turn` - The turn stage contains the turn rules, and a list of phases
- `Phase` - A phase is a part of a Players turn. The phase stage contains
  - *The player can only move on to the next phase if the current phase is complete. For example, in Gin Rummy the player cannot discard a card until they have played a valid set or hit onto an existing set.*
  - Phases contain contain `PlayerActions` which are the actions that the player can take during the phase.
  - Phases contain `Conditions` which are the rules that must be met for the phase to be complete.

Computer Actions

Each stage contains Stages:

- `BeforeActions` which is a list of **Computer Actions** that contains the actions that the computer will take before a stage.
- `AfterActions` which is a list of **Computer Actions** that contains the actions that the computer will take after the stage.

Computer Actions: *(For Before or after Stages)*

Create Group: Creates a new group in the game with the specified tag, view type, and direction.

Check Win Amount: Checks if the player has won based on the specified amount.

Check Win Score: Checks if the player has won based on the specified score. **Place Deck:** Places a deck in the specified group tag.

Transfer: Transfers a specified amount from the source tag to the destination tag.

Shuffle: Shuffles the specified group tag.

Player Actions

The `Phase` Stage contains Player Actions which are actions that a player CAN make during a give phase.

A player can advance to a new phase once the required conditions have been met for that phase.

Player Actions:

PlayerDraw: This action allows the player to draw the first card from a specific source and move it to a specific destination.

PlayerTransfer: This action allows the player to transfer an item or card from a specific source to a specific destination.

CreateGroup: This action allows the player to create a group with a given tag, view type, and direction.

Swap: This action allows the player to swap the contents of two different groups identified by their tags.

Conditions

A condition is a rule that must be met for a phase to be complete. For example, in Gin Rummy, the `Draw` phase is complete when the player has drawn a card from the deck or the discard pile.

Phase Conditions:

Group Empty: Checks if a specific group (specified by the group_tag parameter) is empty.

Meld Condition: Checks if a specific group (specified by the group_tag parameter) contains cards that form a meld (a set of three or more cards of the same rank).

Run Condition: Checks if a specific group (specified by the group_tag parameter) contains cards that form a run (a sequence of three or more cards of the same suit in consecutive rank order).

Run Or Meld Condition: Checks if a specific group (specified by the group_tag parameter) contains cards that form either a run or a meld.

Transfer Amount: Checks if a specific amount of cards (specified by the amount parameter) can be transferred from one group to another.

This is not an exhasustive list of all possible actions or conditions in the game, just some of the most significant features

Changes from the Plan

We had a number of team meetings in the initial planning phase where we worked out how the game data would be encoded and transfered from the backend to the frontend. Once we started coding, the Parser team met with the Builder team to discuss the specific formatting of the objects and the JSON file. However, the parser team didn't meet with the runner team until later in the week. At that point the Runner team had come come up with a slightly different implementation of the condition checking than the Builder team was anticipating. This caused us to have to reconcile the two designs which slowed down the process. **Making sure that everyone is on the same page about implemetnation level details before splitting up into groups is something that we need to work on**

Changing Runner Backend Methods:

This flux in the rules of a game resulted in a disconnect between the teams and required collaborative efforts to unify our design decisions:

- @Del (Exampleks of changed condition names)

Stage Record API:

Since the Runner team was unable to solidify their design needs since not all game variations had been considered. The Parser team had to change the expected definitions required for bulding a complete game and redesign their API. To support new features, paramters, rules, actions, the Builder and Parser teams worked together to design a flexible API based on "records"

The structures of the records were **closed** but the contents were **flexible and open** to change.

`StageRecord` API structure:

```
    public interface StageRecord<SUBSTAGE extends StageRecord> {

      List<ComputerActionRecord> beforeActions();

      List<ComputerActionRecord> afterActions();

      List<SUBSTAGE> stages();
    }
```

This interface was implemented by `GameRecord`, `RoundRecord`, `TurnRecord` and extend by `PhaseRecord`. It was cemeted early on in the design process as a key interaction between the Builder team and teh parser team. However, by leaving the content of the these records open and extentible, any number of before actions, after actions, and stages could be included in a record **including ones that had yet to be inmplemented by the runner**

Valid Options API:

The API is a two way interface between the Builder UI and the Parser. The purpose is to help the Builder present valid options to the UI and check that the correct paramters (and types) are provided to the builder.

```
/**
   * Fetch all available options for a category
   *
   * @param categoryType category type
   * @return list of available options
   */
List<String> fetchOptions(Category categoryType);

  /**
   * Fetches a list of all required parameters for a particular option
   *
   * @param categoryType category type
   * @param optionName   list of required parameters
   * @return list of all required parameter names
   * @throws NonexistentOptionException if the provided option does not exist in the ca
   */
  List<String> fetchParametersForOption(Category categoryType, String optionName)
      throws NonexistentOptionException;

  /**
   * Validates if a given option was provided with the correct input type
   *
   * @param optionName option name
   * @param parameters map of values to parameters
   * @throws RuntimeException if the given information is invalid
   */
  void validateOption(Category categoryType, String optionName, Map<String, String> par
      throws RuntimeException;
```

How to Add New Features

- Add implementation in `src/main/java/oogasalad/runner/model/`
  - `action/computer/` for new before/after actions (or win conditions)
  - `action/player/` for new player actions (valid moves in a phase)
  - `conditions/` for new end of phase conditions
- Add filepath for new condition/action methods to `src/main/resources/oogasalad/file_manager/ParserClasses.properties`
- Add expected Configuration information in valid_options
- Add Translations for Builder and Runner to display for each language in `src/main/resources/oogasalad/file_manager/valid_options`

- Add Runner error messages if necessary in `src/main/resources/oogasalad/runner/languages`

Any modification to existing functionality or new features requries no changes to the builder, it will dynamically load in the data about rules and paramters into the UI and enforce paramter types based on the files in the valid_options directory. Coding for new rules will only need to be in the runner.