Design Document - JAVAMAMAS

Updated Specification

Our goal is to create a program that will allow users to play a game of monopoly locally with other players. When the program runs, each user will be able to assign themselves a character image, a character name and a fixed spot in the turn order. Once the game is started, a monopoly board will be created and players will take turns in the order assigned to them. On each players turn they will do the following:

- Roll/Move if they are still in the game.
- Buy Property - which we will call buyable tiles, and when they buy a whole lot (all properties of the same color), they can also start buying houses or hotels which increase the property value.
- Pay rent when they land on a property tile that is owned by another player.
- Collect 200$ from the bank if they passed the starting tile.
- Abide by tasks or consequences brought up when landing on a community chest tile.
- Go to jail if they land on the “Go to jail” tile

The program will control which players own each buyable tile. Additionally, the program will keep track of the number of tiles sold and remaining at all points in the game, as well as the funds of each player. There are 2 possible ways for the game to end: if all players except one go bankrupt (this remaining player wins), or if all the properties are sold (in which case the player with the highest net worth - balance plus property values - wins). A new feature - as per recommendation, was having a board class to remove some dependencies between controllers and entities. It also made the code for traversing the tiles more organized.

Design Choices and Why?

One major design decision we made was choosing to split up the code by class among one another, which we chose to do in order to make sure we have at least one person with an extremely thorough understanding of each class, and also so that we all implement a relatively equal amount of code. Also, we had a hard choice to make regarding the file structure of the project. Once a structure is chosen is should not change at all. So we decided to have every class in its own folder. This would allow the addition of extra helper files to a class if needed. This can be seen with the communityChestTile, multiple Card helper class were needed, are are able to be added the folder that communityChestTile is held in.

Additionally, a design choice we made to simplify development was to throw as many errors as possible. This allows for some using a class to know as quickly as possible when an issues happened. Stopping people from writing incorrect code assuming it is correct.

Clean Architecture

- Entities: Our entities include the bank class and the player class. The bank stores information about the number of houses/hotels and buyable tiles available. The player class stores information about each player (name, properties owned, balance). So they do not enforce rules on any classes and are not dependent on other classes. Storage of the class data is done through classes in the outer layers.

- Use Cases: The board class creates a board which stores movement of players (an entity) in the game. The player status and game status check whether a player or the game can continue. This is done through checking properties in the entity classes such as player balance.

- Controllers: The Move controller controls movement through manipulation of the board class (use case class).

SOLID

Single Responsibility Principle

- All classes follow this principle. An example is, gameStatus file follows this principle because its only job is to determine different states of the game.

Open/ Closed Principle

- tile class with all of its subclasses.
- the tile class is closed for modification, however can add/ removed as many subclasses as needed to choose which tiles we want/ dont want

Liskov Substitution Principle

- Tile class uses this principle - it is a shared abstract interface
- Has subclasses of different tiles - buyable, community chest, jail, pass, prison, start, tile
- All the tiles implement from the interface and behave similarly as the objects from the interface

Interface Segregation Principle

- Initializable is a very small interface, which forces a single behaviour onto a child

Dependency Inversion Principle

- The Tile subclasses are all dependent on an abstraction in order to avoid the high-level, low-level class hierarchy and becoming dependent on higher levels

Design Patterns

Observer Design Pattern

- have a GameObserver which is a lock on all game events, if one class requests a lock another cannot also have a lock. This allows for multiple objects to be related through a single class.

Strategy Design Pattern

- the Tile class uses this. It has an onAction event which needs to do different things depending on the use case. The implementation of onAction is hidden within the subclasses.

Dependency Injection Design Pattern

- inside of PropertyChecker#canBuy take a Tile. This is because we want to be able to use any kind of tile, not just a buyable tile. Say if a player requests to buy a jail tile, they obviously cannot do this, but we must somehow check that a jail tile is not buyable, the common between jail and buyable tiles are is that they are both tiles. So we use a Tile type.

Template Method Design Pattern

- Inside of entities - we used an abstract class for Tile and had subclasses that were shared by the specific tiles (OnAction, canAddPlayer, addPlayer, removePlayer, getPlayers)

Use of GitHub Features

Our group used many features of GitHub including pull, commit, push, merge, pull requests and looking to Githubs conflict manager when trying to solve conflicts. More specifically, we pulled code as we went through everyone's classes when we met up, so that our local repository would be up to date, and we push any changes we made to our personal assigned classes. We also made sure to commit changes before merging or pulling so as not to lose any work.

Code Style and Documentation

- When coding on IntelliJ, we struggled a lot with fixing warnings due to local issues regarding our programs
- We have yet to overcome all of them, however, it is a part of things we are looking to progress on
    - The issue at hand was that for some of us (Rubaina and Dravin), only some classes were being read by the IntelliJ program, and others were not
    - Hence, we were constantly getting the error *"Cannot resolve symbol [insert class name]"*
- In terms of documentation, we have JavaDoc comments present to indicate what each class does

- Although not all the classes have the appropriate documentation yet, we are in the progress of it

Code Organization & Packages

- For code organization, we decided to organize our classes based on the clean architecture model
- We have our individual classes within subfolders/packages of the different portions of the architecture model (interface, controller, entities, interface, observers, use_cases)
- Specifically within entities, we had split into between bank, player and tiles and we have our individual subclasses divided within them

Progress Report

Struggles

A common struggle in our group has been some git functions being glitchy. We worked together in order to tackle these issues and eventually fixed it but it was a common issue among some group members. Also, IntelliJ had many local issues, depending on our computer.

What has worked so far?

The most efficient way of working for us so far has been to split up the classes among ourselves, work on them from our understanding, and then meet up in person and see if everyone understands each other's code and can connect it to create the running program based on CRC cards. This is the process we used in Phase 1 that led to the most efficient outcome, since conflicting schedules were a struggle we faced earlier on in the semester that we have now found a working solution for.

Work Breakup and Next Steps

After Phase 0 and the provided feedback, we started to split up all of our classes among each other. We decided upon the following distribution:

- Shivanshi:
    - BuyableTile
    - StartTile
- Dravin:
    - Property checker
    - Game status
- Yusra:
    - Bank
    - Player status
-
- Dennis:
    - Player

    - Movecontroller
    - Main
-
- Giancarlo:
    - Board
    - Tile
-
- Rubaina:
    - Prison tile
    - Go to jail tile

Each of us worked on our corresponding classes based on the updated CRC Cards (created from Phase 0 feedback). Then, we met up in-person and looked at each other's work to verify our understanding of all classes, as well as ensure that communication between classes implemented by a fellow group member was sufficient.