Class Descriptions

# Board, Square, StartSquare and Element

The *Board* class is used to hold the list of squares which make up the layout of the game. With add and remove methods it is easy to adjust the number of squares which make up the board.

The decision was made to have the *move()* method within the *Board* class. There was discussion over placing it in either the *Player* or *Game* classes as all three are needed to complete a move. Logically to our group it made more sense for *Board* to hold *move()* as when physically playing a board game the players move around a board not the players move a board.

The *onPass()* method is an inherited method of the *Square* class and is primarily used to add resources to the player when they ‘pass go’ which happens on instances of *StartSquare*.

As with *onPass(),onLand()* is an inherited method of the *Square* class and is primarily used to set the player’s current square and to deal with purchasing elements and paying rent. If the square a player lands on is an instance of *Element* the *onLand()* method checks who owns it before calling either *attemptPurchaseElement()* or *requestRent()*, or telling the player they are the owner. Both *onPass()* and *onLand()* can easily produce a different results in any class that extends *Square*.

In the event the player doesn’t want to purchase they are given the option to offer it to another player. The mapping of players and the process of choosing a player were separated out to individual methods for code reusability as they can be used in other scenarios.

Other notable methods within the *Element* class are *adjustRent()* which increases the rent price when the development level is increased, and *increaseDevLevel()*which calls *adjustRent()*so they happen together.

# Dice and Roller

The *Dice* class was designed to allow the user to change the size. *numOfSides()* is used to set the limit of the integer generated by the *roll()* method. The *Roller* class is used to total the result of multiple dice rolls. This allows for easy scaling if many dice are required.

# PlayerManager, Player and ElementSystem

*PlayerManger* was designed to enable multiple players to be created and stored in a list. The *getUserCount()* and *getUsernames()* methods make use of the *UserInput* class to request a number of players within a set range and the player’s usernames. Constants *MIN \_USERS* and *MAX \_USERS* are used to easily set the acceptable range without having to go looking for the method that limits the number of players. *createPlayers()* uses a lambda expression to create new players and map the list of usernames to a list of players. While developing the code, the welcome and startup menu with the associated methods made sense to have within *PlayerManger* as when the game is initialised an instance of P*layerManager* is made. Also, within *PlayerManager* are the methods mentioned above which map players to a number and allow a player to choose a player from the player map.

The player class is used to hold the player’s resources and what elements they own. Methods to add or remove resources or elements make adjustments straightforward. Several booleans are used to make various checks, including if it’s the player’s turn, if the player has moved, if they are bankrupt, or if they own a complete system. The *ownsFullSystem()* method compares the set of elements the player owns with an *ElementSystem* which is used to group elements into different systems. *ElementSystem*s allows for different sizes of systems to be created.

# UserInput

The *UserInput* class is the central point for dealing with all user input. All methods are *static* to allow them to be called without having to create an instance of *UserInput* in every class. The *getInt()* and *getString()* methods take in a *String* which is the message or question to be asked. There are many points in the game where an *int*, *String* or *boolean* is needed, and having reusable methods that can have the question easily changed cut down on a lot of repetition. Method overloading on *getInt()* allows a range to specified for the requested number. The *yesOrNo()* method is the most reused method. It returns a boolean and controls a lot of the gameplay based on the user answering ‘Y’ or ‘N’. Within *UserInput* is an *isSpeak* Boolean which is used to enable the text to speech functionality.

# Speech

The *Speech* class was included to allow text to speech as an additional accessibility feature. If the user enables speech then all text printed to screen is also spoken. As a free API is used, the build path must be set locally for the game to work.

# Message

The *Message* class was created part way through development and made code review and development a lot easier and quicker. It is a central point for all text printed to screen. With numerous static strings, editing what is outputted to screen is simple and there is no repeated text. Also included in the *Message* class is a *delay()* method as this is used multiple times across many classes with only the length of the delay changing.