Design Decisions

December 20, 2018

Architecture

Once we were given the assignment we spent considerable time hashing out a plan for our system's architecture. Although it was not required in the specification, we chose to implement a graphical UI, instead of using the console. We felt that it would be clearer and more visually appealing to do it that way.

Given the mention of extensibility in the assignment brief we opted for a design that would involve a significant degree of decoupling between the User Interface and the implementation of the game's logic. This would allow the development of either aspect of the game without any implementation details of either main part being known by the other, tied together via the core game mechanics and a messaging interface.

Due to this decoupling, we are able to mock the required interactions between the UI and the logic elements entirely in testing, to ensure that correct logical behaviour is observed in key scenarios as found in the specification. This is elaborated upon in the 'Testing' section below.

High Level Concept

On a high level the architecture is as follows. Due to the turn based nature of this game and the relatively linear flow of a turn with a limited set of very basic interactions with the user we decided to implement the logic as a state machine, using the *State* design pattern from GoF. This finite state machine uses the messages sent by the UI as its stimuli and mutates the logic classes based on the current state and received information. If erroneous (i.e. unexpected) messages are received, these are discarded. The logic will then resume state mutation upon receiving an expected message.

The UI itself runs in a pair of threads. The first of these is responsible for the rendering/-drawing of the current UI 'screen' and runs to a set frame rate. The second thread is the action listener that reacts to button presses. This thread is responsible for changing the state of the UI based on the response messages it receives from the game logic.

The messaging interface is the decoupling intermediary between the two. It defines the messages that are allowed to be sent, and handles the passing of messages without requiring any knowledge of implementation on either end of the 'channel'.

GameEngine.java

This class is responsible for the management of the three main components in this design (UI, Logic, MessagingInterface). When the game begins the Interface is instantiated first, followed by the other two components which have a reference to the messaging system passed to it.

Messaging Interface

The messaging interface itself is quite simple. The `GameEngine` object constructs first a `GameLogic` object, containing the 'brain' of the game. It also constructs the UI object, with one of the arguments to the UI constructor being a `MessagingInterface` object. This

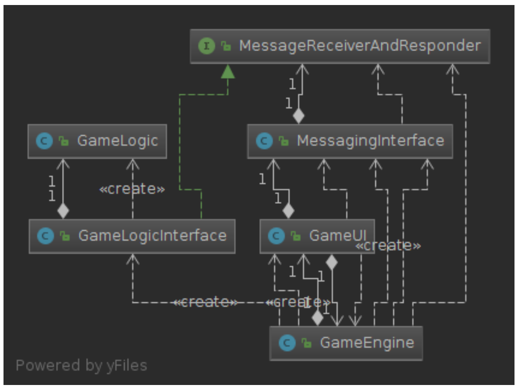

Figure 1: High Level UML.

`MessagingInterface` object owns a reference to an object that implements the `MessageReceiverAndRes`
interface. In this case, that object is a GameLogic object.

As a result, the UI object has a reference to a restricted API of the `GameLogic` object, through which it can send and receive messages. This fulfils our objective of having each component unaware of the other's implementation.

Messages

Twelve messages have been implemented mostly in pairs, which represent a message from the logic and the corresponding response from the UI. Where possible these messages are created using the same class and varying an enumerable message type, however some messages are of a variation on the format and require separate classes to represent them. The types of messages are denoted below along with their relationship structure.

StartupMessage	UIConfigMessage
SpinRequest	SpinResponse
LargeDecisionRequest	LargeDecisionResponse
OptionDecisionRequest	OptionDecisionResponse
AckRequest	AckResponse
EndGameMessage	
SpinResult	

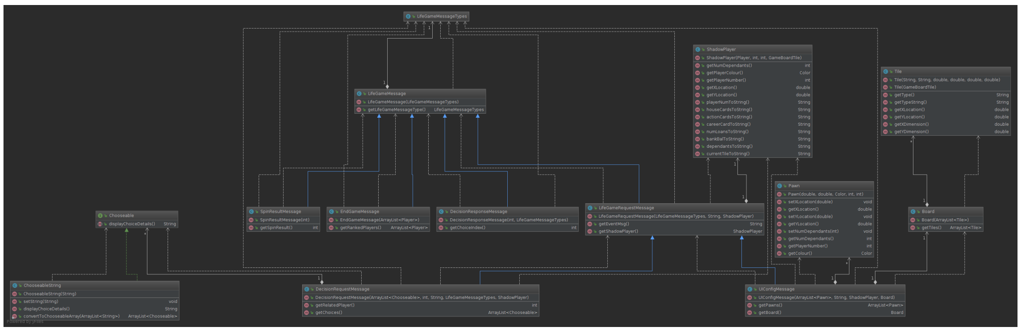

Figure 2: Messaging System Internals.

ShadowPlayer

`ShadowPlayer` is how the messaging system represents the attributes relating to the player object that are of interest to the person controlling that pawn, such as bank balance or the current career card.

In order to abide by the decoupling principle the player object is not directly accessed or sent to the interface, however a representation of this data is sent instead that hides the implementation details.

As far as the UI is concerned this `ShadowPlayer` outputs a series of strings corresponding to each piece of information.

The logic is able to construct a `ShadowPlayer` based on the player objects and send them as an attachment to messages when required.

Tile/Pawn

`Tile` and `Pawn` are the simplest part of this interface, used only when the game is being set up in order to send the coordinates, sizes, types of the tiles and the colours of the respective players to the interface so that they can be drawn.

State Machine

As previously mentioned due to the turn based nature of this game and the relatively linear flow of a turn with a limited set of very basic interactions with the user we decided to implement the logic as a state machine. This state machine uses the messages sent by the UI as its stimuli and mutates the logic classes based on the current state and received information. The original state transition diagram can bee seen in Figure 3.

The idea we had in mind essentially as a pair of state machines with one inside the other. The outside one would handle the overall game state, with actions such as handling the turns, spinning, movement, etc. The inner state machine then would be called into action only when we needed to handle more complex turn logic, such as the spin-to-win tile or the acquisition of houses.

The way in which we managed this was as follows:

When the player had spun the wheel at the start of their turn the logic then enters `HandlePlayerMoveStat`
Once in this state the logic establishes, depending on the tile landed on, what the next state must be. If the state is not complex we handle the turn interaction in `HandlePlayerMoveState` itself. However, if the state is more complicated, the state machine then transitions into a state corresponding to that tile's action, which takes over the handling of user input until the tile action sequence has been completed.

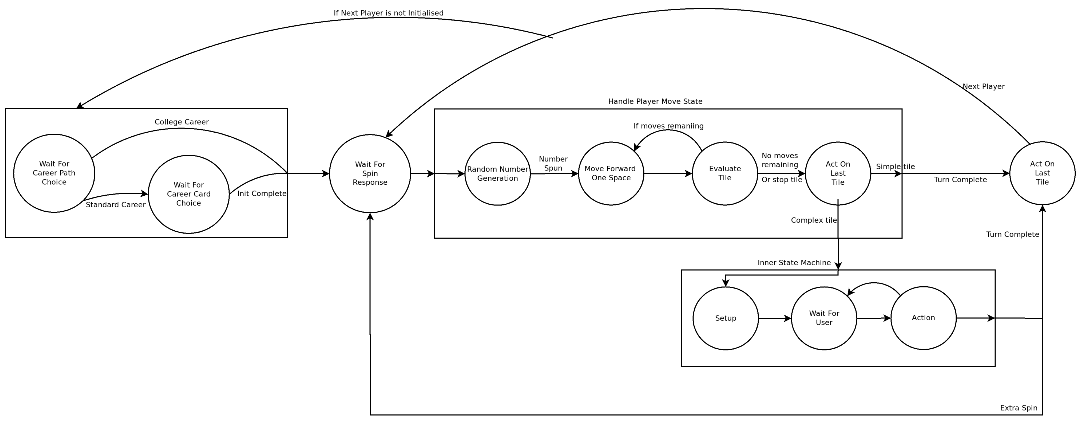

Figure 4: State Machine Design as Implemented.

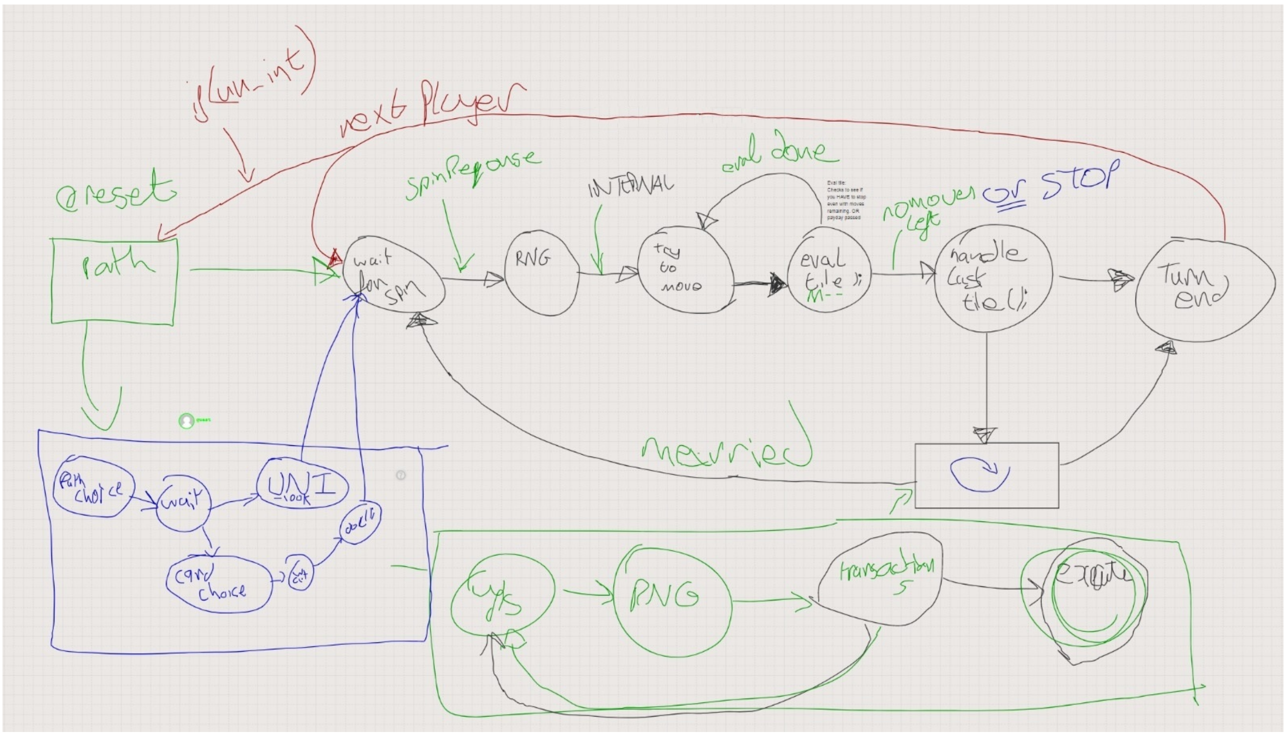

Figure 3: State Machine Design Doc.

UI

As previously mentioned the UI runs in two threads. One thread uses backbuffering to draw/render the various UI screens at a fixed frame rate. All interaction is handled by an action listener attached to the various buttons on the display. Depending on which button is activated the UI sends a message to the UI using the messaging system and awaits a response. Depending on this response the UI's state is mutated to represent this information.

For example if the logic sends a message requesting that the user select one career card from two options the selection screen is displayed to the user alongside the relevant buttons and clicking either of these will send a `DecisionResponse` message as a response to the logic.

Sending a message from the User Interface is a blocking activity in that thread and prevents the user from doing anything until a response has been received from the user.

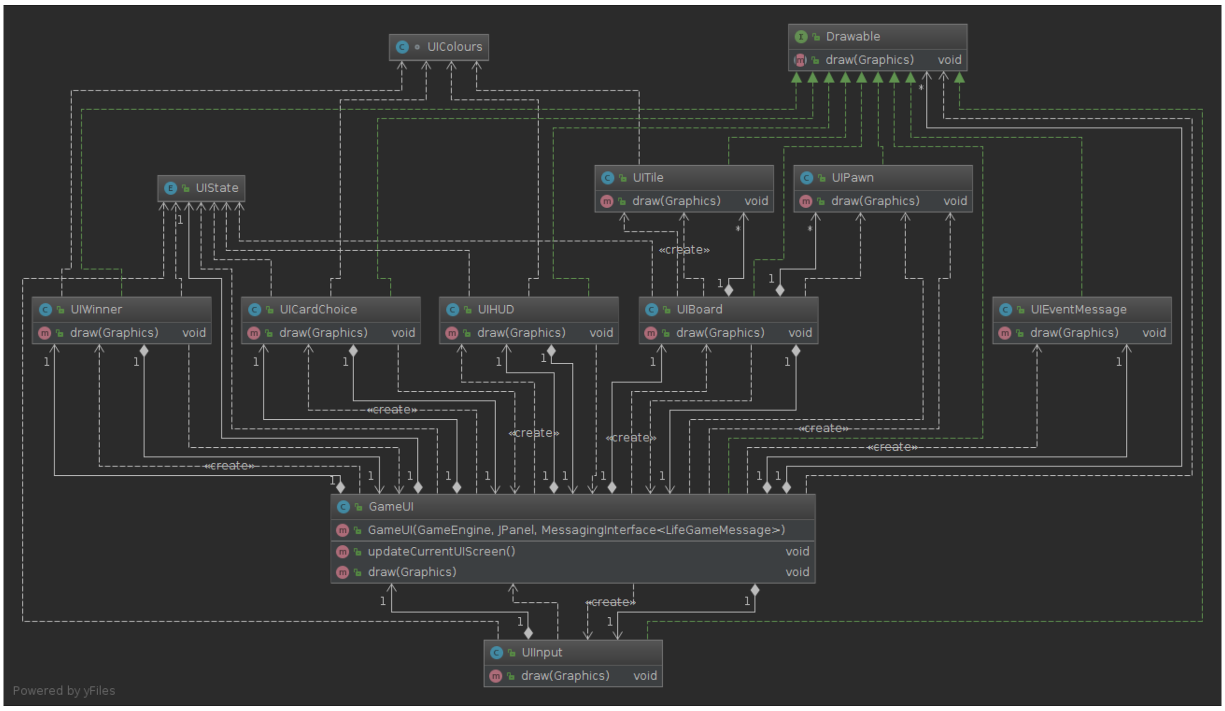

Figure 5: UI Internals.

Board

The `GameBoard` object itself uses a unidirectional acyclic graph to represent the layout of the tiles on the board relative to each other. The layout of this board is at present specified in a `JSON` configuration file. This configuration file consists of two lists:

- List of the tiles, with each element specifying the tile type, position in the UI, and a unique tile identifier (i.e. the vertices of the graph).
- A list of the edges of the graph, each element having a source and destination tile. The tiles are referenced using their unique identifier.

The graph object, and the ID-to-tile hashmap are initialised using an object that handles the `JSON` parsing itself. This is an application of the *Façade* design pattern, from GoF.

Card Decks

The initialisation of each of the `CardDecks` is handled using a similar approach to the `GameBoard` initialisation. The `JSON` parsing itself is handled by a *Façade* object, which outputs a list of the cards of each deck. This list can then be used to construct the internal `LinkedList` used to represent the Deck data structure in our implementation.

Testing

JUnit 5 was the unit and integration testing framework used for our software system. We created unit tests for some of the simpler classes, like the various Deck objects, and the `Bank` functionality. Integration tests (i.e. tests that test behaviour involving communication between multiple classes) for asserting the correct behaviour for each tile type were created. These integration tests are specific to the particular layout of the board that we have created ourselves.