# Design Decisions

December 13, 2018

## Architecture

Once we were given the assignment we spent considerable time hashing out a plan for our system's architecture. 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 we chose 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 portion being known by the other, tied together via the core game mechanics and a messaging system.

## 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. 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 UI then 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 communication system and handles the passing of messages without requiring any knowledge of implementation.

## 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, just consisting of just one function apart from the constructor, `sendMessageAcceptResponse`. This is called by the user interface with an initial message and blocks execution until it returns with a response from the UI. When the interface is initialised it has the logic object passed to it and calls the corresponding receive message function implemented by the logic. 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 user. 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
- SpinRequest
- SpinResponse
- LargeDecisionRequest
- LargeDecisionResponse
- OptionDecisionRequest
- OptionDecisionResponse
- AckRequest
- AckResponse
- EndGameMessage
- UIConfigMessage
- SpinResult

#### ShadowPlayer

`ShadowPlayer` is how the messaging system represents the fields relating to the player object that maybe be of interest to the person controlling that character. 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 as far as the UI is concerned only outputs a series of strings corresponding to each piece of information. The logic is able to construct these 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

### UI