1.0 Key Classes Rationale

1.1 PygameScreenController

There are many different classes which are unrelated in their responsibilities and need to be able to be drawn on the screen. Having the methods available in PygameScreenController instead be placed in those classes would cause lots of code repetition. For example, to draw any image to the screen in pygame, you call a specific sequence of instructions. Drawing different images changes some of the parameters, but the overall structure is the same. By not having PygameScreenController as a centralised class, you would have to instead paste over and over those sequence of instructions within each of those classes to be able to draw them to the screen, which is bad.

1.2 GameWorld

This class has the responsibility of managing the main game's loop, and the interactions associated with it. By not creating this class, the main game loop and all its interactions would be stepped through sequentially by the individual classes themselves within their methods. The loop order would also be abstracted into the classes themselves. This causes a few issues: firstly, this makes the entire game loop almost untraceable as knowing which class would execute its part next would be hard to determine. Secondly, if you required functionality such as responding to a click, you would have to put it in one of those classes. This violates the Single Responsibility principle as now that class must handle the logic of the click by the player on top of their existing responsibilities. Lastly, the dependencies would be all over the place, likely causing circular dependencies (an anti-pattern), as each class would need to know which class next needs to execute its loop.

2.0 Key Relationships Rationale

2.1 ChitCard implements ModularClickableSprite

ChitCard implements the interface ModularClickableSprite. This is a key relationship as without it, ChitCard would need to specify the method to respond to clicks, and the method to get the drawing instructions by itself. This could cause a mismatch with other parts of the game which depend on the return types and structure of this particular interface, such as PygameScreenController, which takes in a specific return structure specified by one of the interface's methods to be able to draw the clickable sprites. Therefore, not implementing this interface increases cognitive load by needing to ensure compatibility with PygameScreenController, and potentially other classes and may cause a runtime crash upon a missed mismatch.

2.2 GameWorld is associated with PlayableCharacter

By having an association (i.e the list of PlayableCharacters as an attribute), it allows the GameWorld itself to handle the turn logic of the game by way of iterating through the PlayableCharacters themselves. If it was non-existent, the association would have to move to GameBoard or some other class which does not have the responsibility of performing game admin tasks such as turns. This would certainly make those classes more complicated, and harder to understand for other developers, increasing the cognitive load that we want to reduce. Potentially, this could also turn the class into a god class (class with too many responsibilities), which is an anti-pattern.

3.0 Inheritance Rationale

I've decided to use inheritance as extensively as possible. Inheritance is used for AnimalChitCard, PirateChitCard, NormalTile, CaveTile among others. Without inheritance from other classes, firstly, the classes themselves would have to repeatedly specify methods that could otherwise been inherited (e.g set_flipped() for chit cards or get_character_draw_instructions() for tiles). This significantly increases the chance of making a mistake that causes one of the implemented methods to function incorrectly, even though they all perform the exact same function. Secondly, no inheritance prevent classes from using any range of classes which are subclasses of the parent class. This may cause classes to depend directly on many concrete classes which have repeating but slightly different functionality, making any change to the system likely to cause changes within major parts of the system.

4.0 Cardinalities

One example of two sets of cardinalities is between GameWorld and PlayableCharacter.

Each GameWorld has the potential to manage 2 to 4 playable characters currently. 4 is the maximum the game can take with the current specifications. However, the minimum is 2 because you need at least 2 players to be able to be able to compete and win when you play the game.

PlayableCharacter is related to one and only one game world, because the playable character cannot be on multiple game worlds at the same time. It exists 'physically' in the software in one world only at a given time.

5.0 Design Patterns

5.1 Singleton: PygameScreenController

I have applied the singleton pattern to PygameScreenController because it contains a list of utility methods that operate on a singleton instance of the game screen. There is no point to

creating multiple instances of the controller, as they would all perform the exact same functionalities as they would all have the exact same state. Therefore, I have chosen to make it a singleton.

5.2 Observer: WinEventPublisher, WinEventListener

As the GameWorld manages all game admin tasks and the running of the game itself (main loop), it needs to be able to know about a win event at any point in the game to stop the game. Having this done by the cave tile itself would violate the single responsibility principle, as it would be involved with managing part of the main game loop which should not be its responsibility. Furthermore, a win event may not necessarily be triggered by the entering of dragons back to their own cave in the future. By using an observer, it promotes game extensibility, by allowing the GameWorld to respond and deal with any event that causes a win in the future.