DESIGN RATIONALLE

TWO KEY CLASSES

SCENE

The scene is a class that contains information about the setup of gameboards, loading screens and more. The init functions allow me specify where and in what order entities are created. This is a great way to encapsulate the game board creation logic and fully remove it from other logic such as the main menu creation logic. This also allows scene switching which can be extremely powerful. By keeping an instantiated scene object, but not running the update method on it we can save game state extremely easily. Scenes also make resetting a simple matter of destroying the existing scene and creating a new one. Encapsulating scene out from the method Word. SetupGame splits up the game loop domain with the game setup domain.

ENTITY

Entity is an abstract class that maintains a list of Components. All objects in my game share different capabilities and using just inheritance alone does not allow me to pick and choose which parts to re-use. The system also allows me to encapsulate logic into small components that can be easily modified and reused at a later date. One key design decision was to allow outside code to add components via the add_component method. This results in a highly flexible idea of an entity whose behaviour can be completely changed depending on the included components. For example, button and player are both entities, yet they are completely different as a result of the included components. The other benefit is the ability to interface away concrete components. Whilst this implementation has been skipped (for now) the system allows complete flexibility to create fully encapsulated architecture. One of the large downsides from the add_component method is that an entity may not have the components that it needs. For example if an entity instance tries to call a Command but it doesn't have a command component then it will not do anything at all. All this means is that I must put safeguards is the form of null checks when using my get_component calls. Creating the Entity Class means that the logic of individual game objects has been encapsulated away from scenes. It is not viable to bundle all game objects into the scene.

TWO KEY RELATIONSHIPS

ENTITY AND COMPONENT

The first key relationship is that between entity and component. This relationship serves to decouple logic for game objects and allow the creation of single domain components. This solution is similar to the popular entity component system software architecture pattern but differs in the idea that my components are both components of data and systems that operate of the data. This means that often times components depend on other components, which is accepted as the added flexibility greatly improves code reusability. To create an object such as a button simply create components for each of the standalone domains in a button: Transform Component, Rectangle Component, Text Component, Hitbox Component, Clickable Component. Clearly the only new code that needs to be created is then a Button Component that fires off a command when it has been clicked. This relationship is composition. Without components an entity would simply be an empty class. All components must be attached to entities and cannot exist as standalones (this includes the case of the complex sprite and hitbox components who can technically have their own components, but these components would not exist if their parent is destroyed).

TRANSFORM COMPONENT AND POSITION

This relationship is an aggregation. Position is a key part of a transform component and without position the transform component would not exist. However, positions can also be standalone. For example, a segment (one section of the volcano card) stores a position that the player moves too. This position is separate from a transform component and shows how position can stand by itself.

TWO SETS OF CARDINALITIES

1 ENTITY TO 0..* COMPONENTS

My entity component architecture consists of entities that have multiple components that give them logic. When entities are first instantiated they have no components (that's where the 0 comes from) and throughout their life cycle unlimited components can be added. Each component is linked to a single entity through the owner member variable. Without a parent a component will not exist.

1 VOLCANO CARD TO 1..* SEGMENTS

In a system that follows the game rules this relationship would be 1 to 3. However, in order to improve the flexibility of the system I have changed this cardinality to 1 to 1..*. This allows creation of volcano cards with any number of segments. Without any segments a player couldn't move to the next volcano card (or escape the cave if it had one so the minimum number of segments was set to 1.

DECISIONS AROUND INHERITANCE

RENDERABLECOMPONENT

It was decided that all components that purely render such as circle, rect, sprite, animated sprite would inherit from renderable component rather than component. This allows greater flexibility in some areas of the code base and also creates the common interface show, hide and setColour to avoid the repetition of code. The render code is mostly the same for all of these so each sub class can just create the pygame image surface, and the renderable component can take care of rendering. In terms of the flexibility this can be seen with the logic that shows the front and back of the chit cards. By setting this to renderable component it will be significantly easier to switch from circles to complete sprites when the art is ready.

WHY NO RENDERABLE AND UPDATEABLE INTERFACES

The decision to not include renderable and updateable interfaces was taken to prevent classes appearing in the wrong places. For example, a scene consists of entities and entities consist of components. All three of these objects have render and update methods so it might make sense to references interfaces. However, we cannot have a scene being stored by an entity, so keep to classes and not using interfaces prevents that.

WHY NO COMPONENT INTERFACES

When readying about entity component systems it was suggested to extract all the public facing information of each component into interfaces. This would stop objects depending on other objects which is desired for correct object orientated design. In practice the entity component system handles decoupling so well that these interfaces are not needed and would only add extra complexity to a system that is already somewhat complex.

EXPLAIN ALL DESIGN PATTERNS.

ENTITY COMPONENT

Entity Component Systems are traditionally referred to as architecture but it is possibly the most important design decision in this implementation so we will cover it here. By favouring composition over inheritance, we deeply encapsulate game domains and allow game objects to be build up domain by domain until their full functionality is achieved. Reusability is promoted through the creation of domain specific components. Generally, there is no harm in breaking up a component into two or more, as the separation of logic greatly reduces tech debt.

UPDATE METHOD

Update method is a sequencing pattern that allows easy management of the game loop. How do we tell all objects in a scene to update. Since we are using entites and components simply add update methods to both of these and then on the game loop manager (World) fire off the update method for each entity that is currently running when you want the game to process one frame. The method allows the implementation of fixed update timestep variable render timestep which is a very popular architecture in modern games. It also allows easy sprite layering and update ordering by running the update and render methods on objects in an order based on their associated layer.

COMMAND

The command design pattern was introduced to solve the problem of buttons. I did not want to have to create numerous button components for each of the effects that pressing a button causes. By turning the associated action of a button to a standalone class this means that to create a different button effect simply create a new command and wire it into the buttons constructor. This had many great side effects and allowed some key code reusability:

Game end -> trigger change scene command

Animation End -> fire provided command.

And more!

SINGLETON

The pygame provided interface to process keyboard inputs has one significant downfall. When you view the events, you consume them. This means that if two components wish to check for left mouse pressed then the first component will eat the event and the second class will never receive it. As a result, a singleton class named input was created to process all of the pygame input events and make them available to other components. The singleton design pattern was created so that there are never two input classes fighting over event consumption. The early instantiate pattern (ie the singleton is created by the World class rather than on call) was utilised to capture inputs from the start of the game.

STATE

Another staple in the game dev community is state. This is a huge design pattern and has numerous implementations from pointers to switch statement you can get as complicated or as simple as you would like. In this implementation we use the most basic implementation to handle simple button presses. The problem is as follows: a button should only fire off events on mouse click if the mouse is already hovering over it. To implement state to solve this solution we create 3 states with different update methods and transfer between states depending on the inputs of that update frame. For example, if the button is in the default state it wont even check if the mouse is being clicked. It will simply check to see if

the mouse is hovering over and if it is transfer to hover state. This greatly simplifies logic and in more complex situations can completely fix the horrible 10 chained if statements.

OBSERVER

When a chit card is incorrectly picked and the players turn ends, we must notify all chit cards to flip back over. To achieve this the observer pattern is used. On creation all chit cards register themselves to the turn manager. If a chit card is flipped and its animal type does not match the active players animal type then it will tell the turn manager to change player. The turn manager iterates through all observers who implement the changePlayer interface and runs the changePlayer method that they implement.

Observers allows containment of logic to separate domains. No class is required to iterate though all chit cards and as a result we reduce the god class problem.