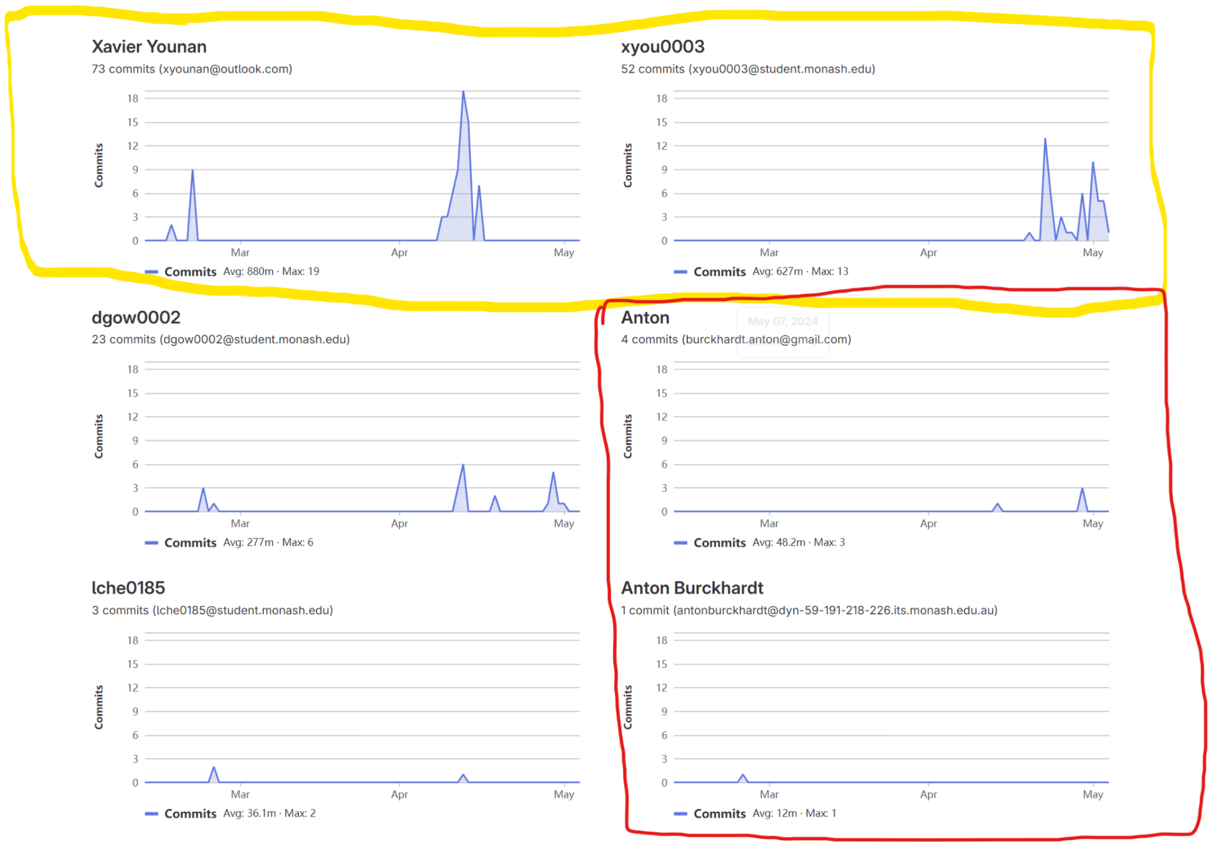
FIT3077
Sprint 4 Documentation

Team XDCA

Table of Contents

Contributor Analytics



Self-Defined Extensions Description

Human Values - Tutorial

One of the extensions we decided upon to address the Human Value's component of this Sprint was the implementation of a short Tutorial on the base game's rules to be made available, before gameplay, on the main menu. This feature aimed to address the value of "Curiosity" under the category of "Self Direction" in Schwartz's Theory of Basic Values.

Notably, the Tutorial has been implemented without describing or demonstrating every single game feature. Missing are the additional features added during Sprint 4 such as the new Chit Cards and Save & Load features. Implementing a couple of extra scenes to explicitly demonstrate these features would have been fairly trivial. However, a conscious decision was made to leave some of the more interesting functionality to discovery during gameplay while still providing curious players with an understanding of the base game mechanics before playing a match.

ShuffleCommand

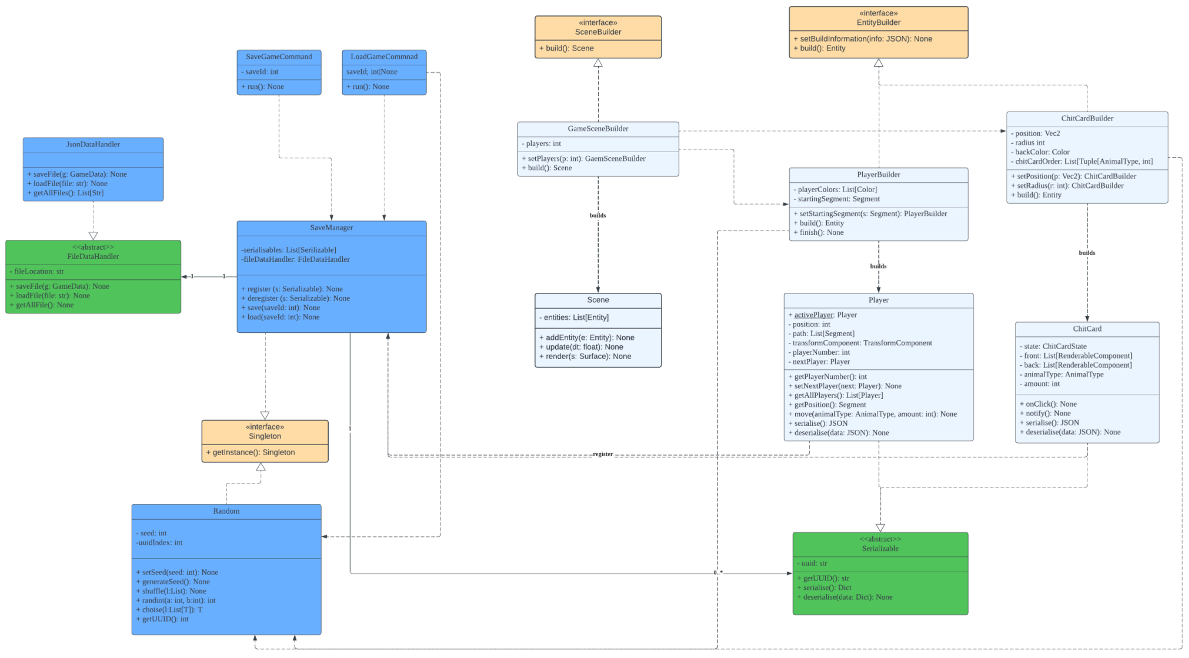
As one of our self-defined extensions we decided to implement a shuffle card to the existing chit cards. This card, once selected, would shuffle the order of all chit cards and once completed would end the player's turn. At first, we believed this would make the (initially easy) game more complex, and after playing a few times, we discovered it added an exciting, challenging and fun new dimension.

Object-Oriented Design

Save and Load

In this representation, the new classes are represented with deeper colours than their existing counterparts.

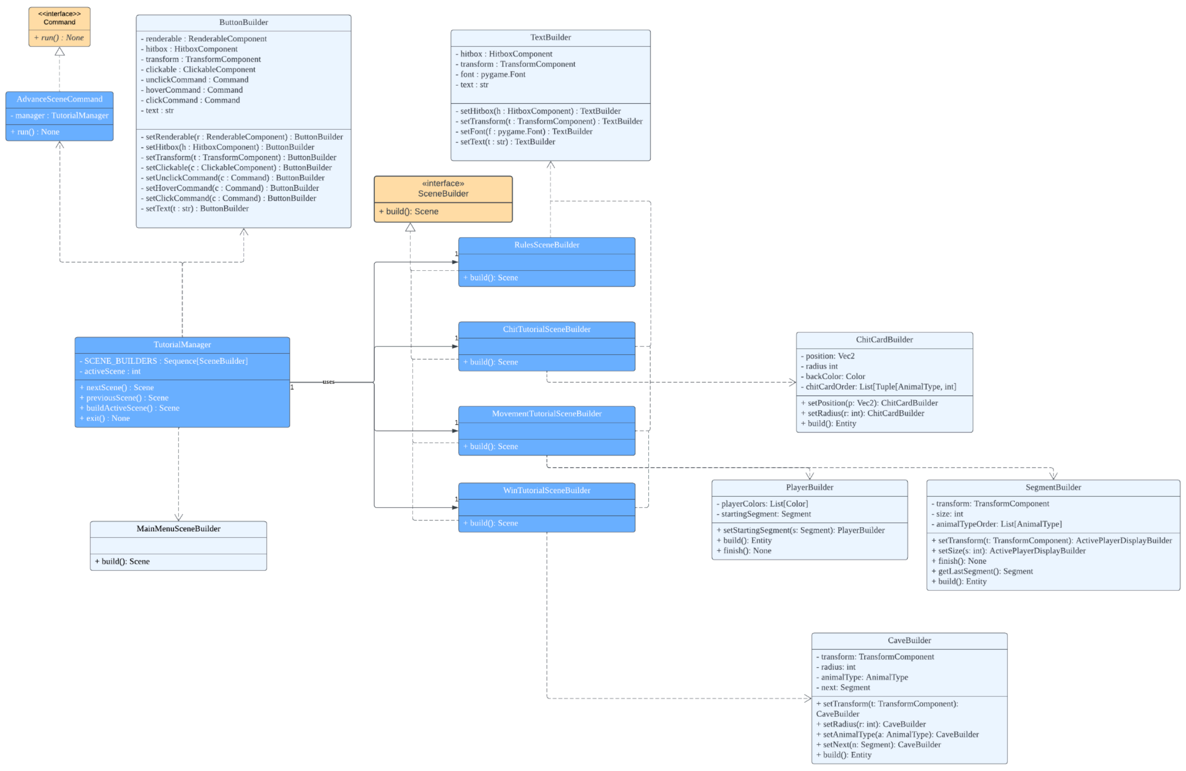
Save and load utilises a Serialisiable abstract class that registers to the save manager. There is also a new abstract class named file data handler. This allows abstraction of different types of files such as Json File Builder in this case. Finally in order to enable the generation of the save ID and to reduce saved data, a seed was implemented in the system. This means that any builders will reference the Random singleton in order to implement seeded random generation.



«interface»
SceneBuilder
+ build(): Scene
«interface»
EntityBuilder
+ setBuildInformation(info: JSON): None
+ build(): Entity
SaveGameCommand
- saveId: int
+ run(): None
LoadGameCommnad
saveId; int|None
+ run(): None
GameSceneBuilder
- players: int
+ setPlayers(p: int): GaemSceneBuilder
+ build(): Scene
ChitCardBuilder
- position: Vec2
- radius int
- backColor: Color
- chitCardOrder: List[Tuple[AnimalType, int]
+ setPosition(p: Vec2): ChitCardBuilder
+ setRadius(r: int): ChitCardBuilder
+ build(): Entity
JsonDataHandler
+ saveFile(g: GameData): None
+ loadFile(file: str): None
+ getAllFiles(): List[Str]
PlayerBuilder
- playerColors: List[Color]
- startingSegment: Segment
+ setStartingSegment(s: Segment): PlayerBuilder
+ build(): Entity
+ finish(): None
builds
SaveManager
-serialisables: List[Serilizable]
-fileDataHandler: FileDataHandler
+ register (s: Serializable): None
+ deregister (s: Serializable): None
+ save(saveId: int): None
+ load(saveId: int): None
<<abstract>>
FileDataHandler
- fileLocation: str
+ saveFile(g: GameData): None
+ loadFile(file: str): None
+ getAllFile(): None
1
1
builds
builds
Scene
- entities: List[Entity]
+ addEntity(e: Entity): None
+ update(dt: float): None
+ render(s: Surface): None
Player
+ activePlayer: Player
- position: int
- path: List[Segment]
- transformComponent: TransformComponent
- playerNumber: int
- nextPlayer: Player
+ getPlayerNumber(): int
+ setNextPlayer(next: Player): None
+ getAllPlayers(): List[Player]
+ getPosition(): Segment
+ move(animalType: AnimalType, amount: int): None
+ serialise(): JSON
+ deserialise(data: JSON): None
ChitCard
- state: ChitCardState
- front: List[RenderableComponent]
- back: List[RenderableComponent]
- animalType: AnimalType
- amount: int
+ onClick(): None
+ notify(): None
+ serialise(): JSON
+ deserialise(data: JSON): None
«interface»
Singleton
+ getInstance(): Singleton
1
register
Random
- seed: int
-uuidIndex: int
+ setSeed(seed: int): None
+ generateSeed(): None
+ shuffle(l:List): None
+ randint(a: int, b:int): int
+ choise(l:List[T]): T
+ getUUID(): int
0..*
<<abstract>>
Serializable
- uuid: str
+ getUUID(): str
+ serialise(): Dict
+ deserialise(data: Dict): None

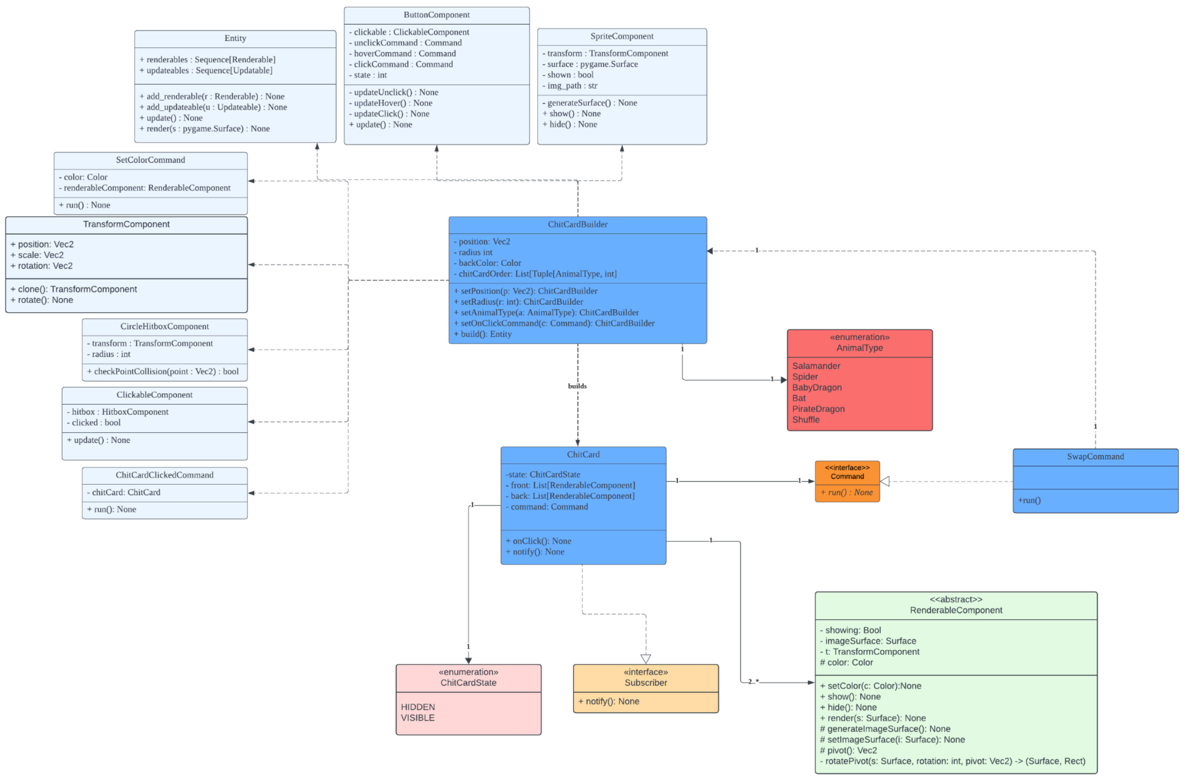
Tutorial

The following UML for the tutorial is represented similarly with new elements highlighted using a dark colour. The design primarily takes advantage of the existing Builders and Scene management solutions created in Sprint 3. Notably, there is an overarching Class for management of the scene that is used to keep track of the current active Tutorial scene and build the scene, previous scene, or subsequent scene accordingly.


<<interface>> Command
+ run() : None
AdvanceSceneCommand
- manager : TutorialManager
+ run() : None
ButtonBuilder
- renderable : RenderableComponent
- hitbox : HitboxComponent
- transform : TransformComponent
- clickable : ClickableComponent
- unclickCommand : Command
- hoverCommand : Command
- clickCommand : Command
- text : str
- setRenderable(r : RenderableComponent) : ButtonBuilder
- setHitbox(h : HitboxComponent) : ButtonBuilder
- setTransform(t : TransformComponent) : ButtonBuilder
- setClickable(c : ClickableComponent) : ButtonBuilder
- setUnclickCommand(c : Command) : ButtonBuilder
- setHoverCommand(c : Command) : ButtonBuilder
- setClickCommand(c : Command) : ButtonBuilder
- setText(t : str) : ButtonBuilder
TextBuilder
- hitbox : HitboxComponent
- transform : TransformComponent
- font : pygame.Font
- text : str
- setHitbox(h : HitboxComponent) : TextBuilder
- setTransform(t : TransformComponent) : TextBuilder
- setFont(f : pygame.Font) : TextBuilder
- setText(t : str) : TextBuilder
«interface» SceneBuilder
+ build(): Scene
RulesSceneBuilder
+ build(): Scene
ChitTutorialSceneBuilder
+ build(): Scene
MovementTutorialSceneBuilder
+ build(): Scene
WinTutorialSceneBuilder
+ build(): Scene
1
1
1
1
TutorialManager
- SCENE_BUILDERS : Sequence[SceneBuilder]
- activeScene : int
+ nextScene() : Scene
+ previousScene() : Scene
+ buildActiveScene() : Scene
+ exit() : None
1
uses
MainMenuSceneBuilder
+ build(): Scene
ChitCardBuilder
- position: Vec2
- radius int
- backColor: Color
- chitCardOrder: List[Tuple[AnimalType, int]
+ setPosition(p: Vec2): ChitCardBuilder
+ setRadius(r: int): ChitCardBuilder
+ build(): Entity
PlayerBuilder
- playerColors: List[Color]
- startingSegment: Segment
+ setStartingSegment(s: Segment): PlayerBuilder
+ build(): Entity
+ finish(): None
SegmentBuilder
- transform: TransformComponent
- size: int
- animalTypeOrder: List[AnimalType]
+ setTransform(t: TransformComponent): ActivePlayerDisplayBuilder
+ setSize(s: int): ActivePlayerDisplayBuilder
+ finish(): None
+ getLastSegment(): Segment
+ build(): Entity
CaveBuilder
- transform: TransformComponent
- radius: int
- animalType: AnimalType
- next: Segment
+ setTransform(t: TransformComponent): CaveBuilder
+ setRadius(r: int): CaveBuilder
+ setAnimalType(a: AnimalType): CaveBuilder
+ setNext(n: Segment): CaveBuilder
+ build(): Entity

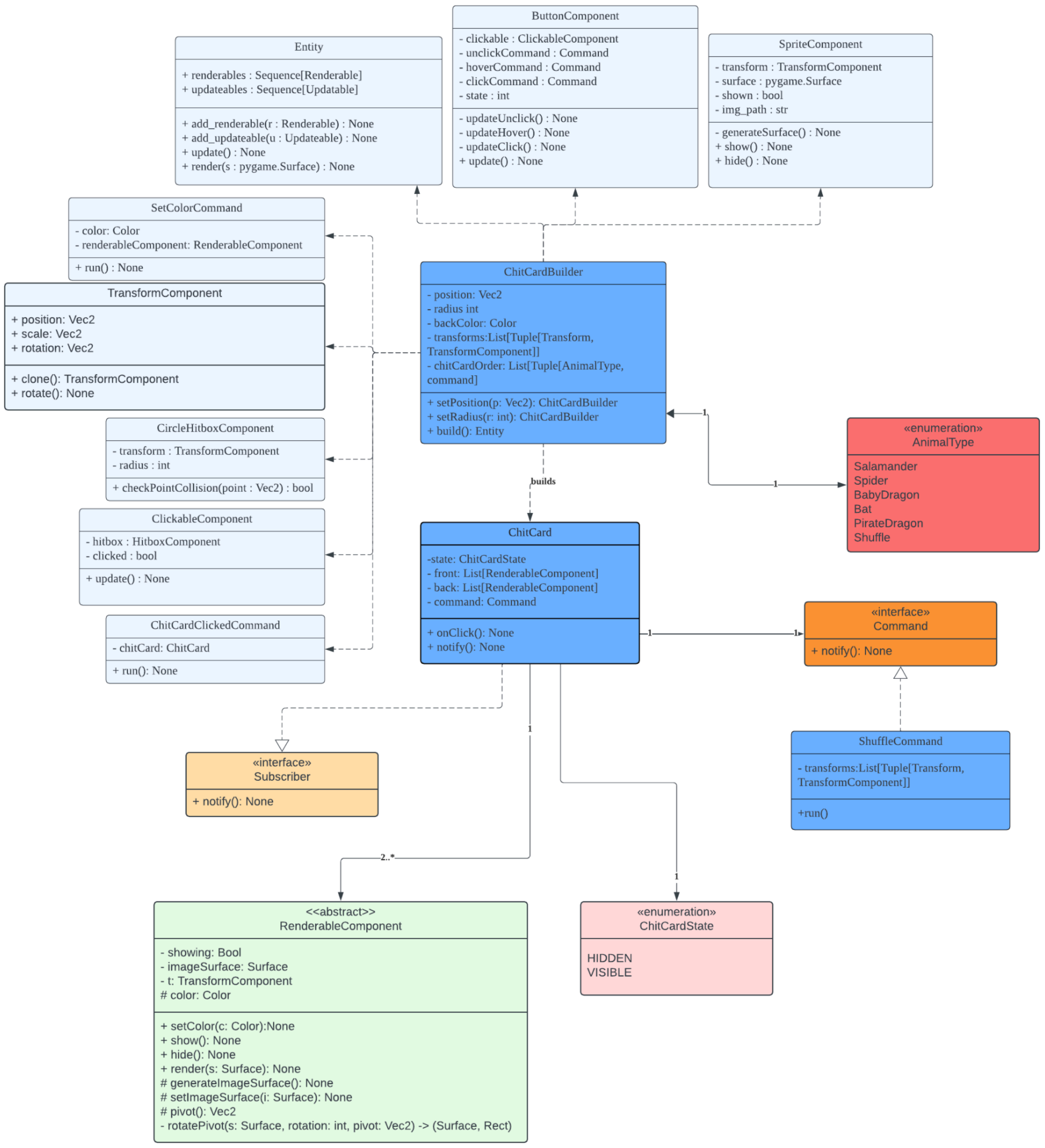
Swap Player

For the swap UML diagram is basically similar to the original design of the Chit card part but with a few change on the animal type, a new element swap was added to represent the Swap player card, in addition, A new command response to the card function SwapPlayerCommand is also added and linked with the Chitcard builder class so that when the Swap card type being selected, that command will be executed.



ShuffleCommand

For our class diagram of the initial prototype an abstract command class was added to the ChitCard entity in order for the new ShuffleCommand class, and also therefore various other later defined commands, to easily be linked with a specified ChitCard accordingly. Slight additions to add the order of the ChitCards as a list was needed, as well as adding the ShuffleType to the AnimalType of Chitcards. Other components were otherwise kept the same as other than the executed command, no functions should differ from other ChitCards.



Sprint 3 Reflection

New Chit Card

During the development, Integrating the new chit card functionality required ensuring compatibility with the existing ChitCard structure and game mechanics, However, In the original design from Sprint 3, the responsibilities were well-distributed, with specific components handling distinct tasks. The Use of the Builder pattern helped in modularizing the creation of chit cards, The game components are relatively independent which is making it easier to incorporate new attributes and behaviours. Which makes future implementations or updates much easier.

With the new feature Swap player command the logic for swapping player positions required a precise understanding of the game's state and how positions are managed to ensure that both players' positions and transform components were correctly swapped without causing inconsistencies in the game state was challenging. Managing the transformComponent during swaps required careful handling to ensure that both the position and the transform properties were correctly transferred between players. Similarity, By leveraged design patterns like the command pattern, adding new commands becomes almost plug-and-play. The MoveActivePlayerCommand can be structured similarly to existing commands, enhancing consistency and reducing development time.

The need to clone and copy transform components and understanding the method being created, what value or outcome it will bring and the lack of naming convention or comment added to the complexity, especially in maintaining the integrity of the game's state, trying to move the player object or Finding the Closest Player as the distances of the game object are relative depend on the cave they are created, Therefore, Improve Distance Calculation Logic like refactor the logic for traversing and finding players to be more efficient and modular might be something to consider when back to sprint 3.

This could involve using strategies or utility classes dedicated to player distance calculations and comparisons, ensuring that this logic is well-tested and encapsulated, making it easier to reuse and extend in different commands. Also focusing on a more flexible component system would have been beneficial. Like Encapsulate player position logic in a separate class. This way, position-related operations can be managed independently from the player class, making it easier to modify or extend. In this way, Instead of handling the transform component directly within the SwapPlayerCommand, we could introduce a TransformManager responsible for managing transform operations. This manager would handle cloning, copying, and swapping of transform components, providing a clean API for these operations.

Save & Load

Save and load was implemented using a serialisable class. This provided methods Serialise and Deserialise to generate data and convert data into an existing class. The existing system made it easy to register serialisable components. Since serialisable components could not register themselves to the save manager singleton, the existing builder implementation was able to register with no issues.

Creating the save and load interface was aided by the existing buttons and change scene commands. This enabled the complete interface to be created with no modifications to the existing implementation or underlying game engine. A new builder named save popup builder was created that leveraged the existing button builder.

Ultimately components, commands, and builders ensured that generation of serialisable components and saving data to a file required no refactoring.

The decision to include elements via dependency injection is standard practice for both Unity and Unreal Game Engines. It removes hard coded dependencies and decreases the coupling of the system. Unfortunately this results in classes not having the entirety of their creational data, and results in classes being built by builders that can manage the creation and injection of dependencies. For example the ButtonComponent requires four commands, and a clickable component. The builder creates these dependencies and injects them into the ButtonComponent returning the final entity at the end of the build processes. This poses an issue when working with data that would modify dependencies that are injected into an object.

For example, the VolcanoCard has a dependency for a List[Segment]. These segments must be created, positioned and injected into the VolcanoCard upon creation. Therefore we must somehow deserialize the VolcanoCardBuilder (and all other builders) before we deserialize the individual Volcano Card. This presented a significant challenge and ultimately resulted in a 'hack'. As a project grows larger and larger there will always be unaccounted edge cases that break the existing system and require a refactor. In this instance I decided to opt for the hack, reducing the implementation complexity but also adding a layer of tech debt into the project.

Ultimately the seed approach allowed a minimisation of saved information encoded by the builders. This approach is preferred over saving complete game information as it gives the developer greater control of the stored information, at the cost of synchronisation issues in certain instances. Figure 1.0 indicates the method of storage for each storage requirement. It also indicates the method of storage for extensions. The storage method of Serialisable indicates that the data is written and loaded to a file. The storage method of seed indicates that the data can be re-created through our seeded randomization system and there is no need for a serializable implementation for that data.

Data	**Storage Method**
Volcano Card Sequence	Seed
Cave Location and Animals	Seed

Player Token Location	Serializable
Location of Chit Cards	Serializable
Player Turn Order	Serializable
Variable Volcano Card Tiles	Seed
Variable Cave Position	Seed
Variable Volcano Card Number	Seed
Shuffle Extension	Serializable
Swap Extension	Serializable
Tutorial	N/A

Figure 1.0, Data Storage Method.

The decision to include complete generation animations with elements of the game scene flying in at different times further introduced synchronisation issues but a small refactor changing the clone time to on run rather than on command generation ensured that the animations moved the elements to their post-deserialisation positions, rather than their scene generation positions.

One development practice that worked great throughout the entire project was the use of separate feature branches. There was only one instance of a merge conflict and the branches allow simultaneous development on different features. A personal development practice that I need to improve is my use of doc string. At the beginning of a project when creating core classes I use docstring to guide my development and provide an easy api reference. As the project moves into its later stages I often skip the docstring process resulting in difficulty remembering the purpose of certain commands or components. In the future I plan to slow down my development process and properly document functions where appropriate.

Finally, in this project I felt as though I had to take on the brunt of the planning and implementation process. Sprint three was when I first noticed the problem, we had no set due dates and no specific task requirements. As the project progressed two of the students failed to offer any help. To combat this I created introductory tasks for each student, however it seems they thought this was the entirety of their required contribution for this project. In sprint four I proposed a full task breakdown before the assignment had begun. This resulted in increased accountability, however the contribution was still at an unsatisfactory level and I still had to design three features and implement two. Moving forward, more frequent progress updates and more specific task breakdowns can help ensure work is being completed in a timely manner.

Human Values - Tutorial

The primary aim in designing this feature was to use as much of the implementation of existing parts of the system as possible without compromising functionality. This included the use of the various builders in the system, i.e. PlayerBuilder, ChitCardBuilder, and SegmentBuilder, as well as all Scene management related functionality such as the SceneBuilder interface.
Overall, the implementation itself is quite close to the original design for this feature and the code per-scene is not overly lengthy. However, there were some issues related to the Sprint 3 implementation that caused some issues.

Firstly, many of the builders developed in Sprint 3 are very tightly coupled to the game functionality. As we wanted to ensure that the tutorial had elements that were functional it was necessary to add some features to these builders, as well as make use of some “hacky” implementation techniques, in order to get them working in the tutorial.

This included added the ability to disable the initial animation of elements appearing on the board as the timer on these was hard-coded and would cause confusion in a tutorial, and manually setting the path of the player to demonstrate the correct behavior and avoid the “Game Win” screen once the end of the path was reached. The major problem with this is that some future, unrelated, implementation changes may break the way the Tutorial is set up as it, in places, makes use of implementation details for specific classes.

For example, should the implementation of player movement be changed from a pre-computed path approach to a run-time approach, the implementation of the Tutorial will need to be changed. Ideally, the movement of the player would basically work out of the box.

One design approach that could have been considered during Sprint 3, without straying too far from the current design, would be to design and implement more specific components for pieces of functionality and then have these as togglable in the respective builders. For example, a WinConditionComponent on Player entities that could be excluded when a win need not be triggered. Or, alternatively, dependency injection could have been used with specific commands to tell the player entity and its components what to do in specific scenarios. For example, a WinCommand could be injected for real gameplay with the DoNothingCommand injected for the Tutorial. These design choices have the drawback of offloading greater burden onto the Object/Module making use of the Builder but allow more fine-grained control and code reuse as a result.

Secondly, the use of both entities and concrete classes compounded with this previous difficulty by making it somewhat difficult to access the necessary objects and fields that needed to be modified for use in the tutorial. For example, the getLastSegment method had to be used on the SegmentBuilder as the build method for this class only returned entities despite also constructing Segment objects.
A better solution would be to just choose a single approach. For a project as simple as this it may have been best to stick to concrete classes exclusively and simply attach components to them directly as fields rather than making use of the Entity abstraction.

Shuffle Chit Card

Adding the new ShuffleCommand was relatively easy due to the encapsulation of various other commands and the principles of our previously established entity component system. The goal was to leverage existing game features to seamlessly integrate the ShuffleCommand with the ChitCards.

The Observer design pattern, implemented to notify when a player's turn is over, allowed the ShuffleCommand to integrate effortlessly into the player's turn capabilities. Additionally, having encapsulated commands such as DelayExecuteMFCommand facilitated the implementation of execution and animation times.

Our system's distribution, enabling each chit card to know its position, allowed the command to easily access and shuffle the transform positions.

However, slight refactoring was needed to link commands directly to each chit card. Previously, only one command (MovePlayerCommand) was naturally assumed to be linked to each chit card. This change made it easier to implement various other commands for newly created chit cards in future iterations of the game. This change was reflected in the initial UML design. However, further modifications were made, including the addition of the ExecuteShuffleCommand class, to better encapsulate the shuffling and animation logic. This encapsulation will facilitate easier reuse of the logic in future commands which was overseen in the initial UML design. Additionally, minor changes to the logic of the number of cards created were necessary, as the 16-card limit had been hard-coded.

These changes in how commands are linked to ChitCards could have been better accounted for in Sprint 3 to simplify implementation. During Sprint 3, no straightforward options were put in place to simply add a new command or a list of commands. Immediately incorporating a command interface to a desired chitcard could have simplified the entire process.

Executable Information

Antivirus Issues

If the exe is flagged by windows defender or your antivirus then you must add it to its whitelist folder. For windows defender (windows 11):
1. Open windows security app
2. Navigate to virus and thread protection.
3. Click manage settings under Virus and Threat protection settings.
4. Scroll to Exclusions (at the bottom)
5. Click add or remove exclusions
6. Add the dist folder

Build

In the base directory (the location of the file build.bat), run the following command:
./build.bat
The exe file is located in dist/FieryDragons.exe

Run

1. Check whitelist settings as described in antivirus issues
2. Download the .zip and extract or build the project
3. Open the dist folder and double click FieryDragons.exe to run