Technical Design Document

## Final Playtest

## Basic Game Information

**Names of group members if working in a group**

Dev Patel

Tien-Yi Lee

**Will your game be 2D or 3D**

2D

**Elevator pitch! Basically describe the selling points of your game in one tweet worth of text.**

You are the commander of a medieval king’s army, but one day hoards of pirates attack the castle! Your goal is to strategically defend the castle and king from the pirates’ attacks. You can even bring in a friend to control the pirates if you want a greater battle!

**Target audience**

Young children to young adults (maybe a bit lower on the upper side)

ESRB would probably rate it E (Everyone) maybe E10+

**Long description of your gameplay**

In singleplayer, you strategically summon troops to ward off the pirates’ attacks. You passively generate coins as time goes on and are able to purchase defenders in exchange for a certain amount of coins, where better defenders (stats-wise) cost more money. Pirates will continue to spawn in waves until the Pirate King spawns. Killing the Pirate King wins you the game, whereas your king getting killed results in a loss.

The game is also playable in multiplayer where the only difference is the second player controls when pirates spawns (as opposed to a predetermined set of spawn times). (Multiplayer is currently a work in progress, singleplayer is only available at this time).

## Controls System

**Design Patterns used:**

No specific design patterns are used for the controls system. The controls are based on clicking buttons or pressing buttons on the keyboard that basically equate to clicking a button with a mouse. The buttons’ actions are done through Unity’s EventSystem in the inspector (the on click part of a button in the inspector) and do things like set things active/inactive as well as certain methods from the GameStateManager (like restart scene, quit to title, etc.). The button controls are defined using Unity’s Input Manager and the GamePlayManager and GameStateManager listens for the respective buttons in their update function.

**Class Descriptions:**

In the main menu scene, the Game State Manager handles the clicking of buttons in this scene by providing the methods that are called on button press – one method for playing/loading the game scene, another method for quitting the application. Additionally, Unity’s On Click Event System for buttons in the inspector also handles some functions such as turning on an object on/off.

In the gameplay scene, when using buttons in the UI to spawn characters, there is a class that provides methods for the all the spawning buttons – in other words, there is a method for spawning each type of character which is then linked to the appropriate button in the inspector.

If you are using keyboard buttons (instead of clicking buttons in the UI as described above), then the GameplayManager listens for the keys and calls the appropriate spawn method from the CharacterManager given that the spawning conditions are right (enough coins).

**Class Relationships:**

The GameStateManager class is independent and doesn’t rely on other classes.

The gameplay scene controls class is tied with the GamePlayManager because the GamePlayManagers calls a Setup() function that places the buttons (controls) onto their respective spots on the UI canvas. The class is also dependent on the CharacterManager class (explained below). For handling UI button clicks, the SpawnHandler.cs methods are used, for handling button clicks on keyboard, the GameplayManager listens for specific keys.

**Dependency on other systems**

The main menu scene controls class doesn’t depend on other classes – only on Unity’s SceneManagement directive.

The spawn buttons class in the gameplay scene relies on the CharacterManager system because its methods directly call methods from the CharacterManager in order to spawn characters.

**Client Systems:**

There are no client systems (classes) so to say as the user is the one that directly interacts with them (via button clicks or keyboard button clicks). If the game screen is considered a client then that would be the client system the control classes work with.

## Camera System

**Design Patterns used:**

The CameraManager is a singleton so when it’s Setup() method is called by the GameplayManager class, that class doesn’t need the reference to the CameraManager object (it calls the public static instance of CameraManager).

**Class Descriptions:**

The one camera class (CameraManager) has four [SerializedField] that are used to calculate the position of the camera depending on if the game is in singleplayer or multiplayer.

**Class Relationships:**

It has a relationship with the GameplayManager because that class calls the camera class’s Setup() function that puts the camera in the right place.

**Dependency on other systems**

The single camera class is also dependent on the GameplayManager because the GameplayManager tells the camera class if the game is being played singleplayer or multiplayer, which the camera class uses to then determine where the camera should be positioned.

**Client Systems:**

There isn’t really a client class other than GameplayManager which call’s the camera’s Setup() method. I suppose one could define the game scene display as a client because that’s how the players interact with the camera (by seeing what the camera points at).

## Game Data Systems

**Design Patterns used:**

One composite pattern is used for creating/storing/using the spawnable characters (see ICharacter, Character (leaf), Army (composite)). Additionally, there is a decorator that is used if the character is a king, which changes the death method for that king character.

A singleton is used to manage all the character and army objects called CharacterManager.

And a ScriptableObject (CharacterData) is used to store the different stats for each type of character (unsure if this counts as a Design Pattern but I will keep it here just in case).

Another composite pattern is used for special effects such as changing the speed, resistance/armor, etc. of characters (see ISpecialEffect, SpecialEffect\_Group (composite), SpecialEffect\_AttackDamage (leaf), SpecialEffect\_Range (leaf), SpecialEffect\_Resistance (leaf), SpecialEffect\_SpeedMod (leaf), SpecialEffect\_TakeDamage (leaf)).

Also, even though this is not a design pattern, we will mention that PlayerPrefs are used to store simple data such as if the game in singleplayer or multiplayer and singleplayer highscore because PlayerPrefs are globally accessible.

**Class Descriptions:**

For the composite pattern based on ICharacter, ICharacter is the interface that everything implements – void Attack(), void TakeDamage(damage), void Die() are the methods in that interface. One leaf is Character which of course implements ICharacter and also sets its own stats using a CharacterData ScriptableObject – the character class can do things such as take damage, die, and attack. If the character is a king, it can be wrapped/decorated with a KingCharacterDecorator (derived from BaseCharacterDecorator), which gives the king character a special death effect (changes the state to pirate win/castle win). Additionally, the Army class acts as a composite for characters, which is used to create an army for pirates and an army for castle troops. The army class also has an ability to return a unit from its armylist, which characters use to get a target to attack.

For the other composite pattern based on ISpecialEffect, ISpecialEffect is the interface that everything implements – void Effect() is the only method in that interface. There are 5 different leaves - SpecialEffect\_AttackDamage (change a character’s attack damage), SpecialEffect\_Range (change a character’s attack range), SpecialEffect\_Resistance (change a character’s resistances (armor), SpecialEffect\_SpeedMod (change a character’s speed), SpecialEffect\_TakeDamage (make a character take damage or heal if negative damage is applied). Some of these effects were put into a composite (SpecialEffect\_Group), which acts a typical composite (relegates all work to children).

**Class Relationships:**

For both composites, without the need to say, all classes that implement the same interface have some relationship to each other as is the nature of the composite pattern.

The king decorator/wrapper also relates to the Character class because it is intended to wrap/decorate a Character.

**Dependency on other systems**

The ICharacter composite pattern depends on the CharacterManager singleton because that manager spawns characters and puts characters in the armylist (composite). It also is responsible for wrapping/decorating the king characters.

The ISpecialEffect composite pattern depends on the SpecialEffectsManager singleton because that manager handles the execution of special effects, creation of special effects (leaves and composites), and displaying of special effects.

**Client Systems:**

The CharacterManager and SpecialEffectManager both are clients to their respective composite patterns as it is the primary way that the two composite patterns (and decorator for ICharacter) are used.

## Player and Game State Systems

**Design Patterns used:**

GamestateManager Singleton.

GameplayManager Singleton that uses delegates.

CoinManager Singleton.

ScoreManager Singleton.

CharacterManager Singleton that uses prototype.

**Class Descriptions:**

The GameStateManager handles the states of the overall gaming (playing, menu, paused, etc.). It provides functions such as switching scenes, restarting game scene, and other scenes, and is the way for the two scenes (game scene and menu scene) to “interact” with each other because the GSM does not destroy on load.

The GameplayManager handles the general in-game (not menu scene) state in a given game. It is responsible for setting up other managers (by calling their respective setup methods). It also creates a delegate for what happens if the Pirates win and what happens if the Castle wins, which other classes can then subscribe their own methods to.

The CoinManager is a singleton that handles generating coins every fixed amount of time for players to use to buy troops.

The ScoreManager is a singleton that handles incrementing the player’s/players’ score every fixed amount of time.

The CharacterManager is a singleton that handles the creation of characters (and decorators) using prototyping via instantiating prefabs. It also manages the creation of army (ICharacter composite), and in general is the way for any other class to access character related stuff (which is representative of player state because a player’s state is a mixture of their score, coins, and characters in their army).

**Class Relationships:**

The GameStateManager is the ‘ultimate manager’ because it is responsible for switching scenes and telling the GameplayManager whether to play the game in singleplayer or multiplayer (via playerprefs).

The gameplay manager relates to all the other managers (other than the GameStateManager) because it tells them when to setup.

The score and coin managers relate to the canvasmanager because they tell the canvasmanager to update the score/coins ui display when their score/coin value change.

The charactermanager relates to the entire ICharacter composite pattern because again, it is the way the ICharacter composite gets instantiated, stored, and managed.

**Dependency on other systems**

The gameplay manager is only dependent on a PlayerPref for if the game is being played in singeplayer or multiplayer.

The coin and score managers are dependent on the gameplay manager telling them to setup.

The charactermanager is also dependent on the gameplay manager telling it to setup.

**Client Systems:**

The gamestatemanager doesn’t really have a client other than the gameplay manager which indirectly asks (through playerprefs) if the game is in singleplayer or multiplayer.

The gameplay manager doesn’t really have a client other than the other managers (not the gamestatemanager) when they ask if the game in singleplayer/multiplayer at the very start, after that the gameplay manager only acts as a client for keyboard input since its update functions listens for key presses.

The score manager acts on its own other than the gameplay manager telling it to start, so the gameplay manager can be a client in a way.

The coin manager is used by the character manager to see if the player has enough coins to spawn a character. Its setup method is also called by the gameplay manager.

The character manager has a few clients: the first being the gameplay manager telling it to set up. The second being the gameplay spawn class that has methods which use the character manager to spawn a character.

## UI Systems

**Design Patterns used:**

For the game scene - Singleton CanvasManager with a delegate that can be called by other classes, which updates the displayed data (current coins and score). The CanvasManager here has many children Canvas objects that are turned on/off to change the UI.

In the menu scene, there is no specific design pattern for the UI as Unity’s OnClick event system for buttons turns certain Canvas objects on/off to create the UI.

**Class Descriptions:**

The CanvasManager manages different canvases and turns certain ones on/off depending on what is needed. There is a singleplayer canvas and multiplayer canvas (only one should be active) and there is also a pirate win and castle win canvas (only one should be active when the game ends).

TienYi – The GameStateManager class has methods that are hooked onto buttons in the main menu/end of game screen to switch between scenes/restart a scene.

**Class Relationships:**

Relates to the coin/score managers because it reads data from there to display.

Also relates to gameplay manager because it gets told to setup from the gameplay manager (setup a singleplayer canvas or multiplayer canvas).

Relationship with the SpecialEffectManager because that manager tells a specific Canvas object in the game scene what text to display (to describe what special effect is happening).

The GameStateManager also turns on/off a pause menu UI in the CanvasManager on pressing ‘Escape’.

**Dependency on other systems**

A lot of the UI depends on the gameplaymanager and gamestatemanager because they tell the CanvasManager to display the singleplayer or multiplayer UI. Afterwards, the UI is updated based on data from the coinmanager and scoremanager. There is also a dependance on the SpawnHandler class because it provides the button OnClick methods to spawn characters.

**Client Systems:**

The coin manager and score manager are clients because they call a delegate that updates the coin and score displays on the UI.

The King Character Decorator is also a client because it calls a delegate that turns on either the Pirate Win Canvas or Castle Win Canvas.

The gameplaymanager and gamestatemanager also are clients because they tell the CanvasManager which one of its Canvas child objects to display (and which ones to turn off).

**Other:**

TienYi – For buttons that are clickable, their function comes from a reference to a script in the OnClick() section of the button in the Inspector. A script doesn’t assign their functions, which is why we are mentioning this here.

Nav Agent and AI Agent Systems

This will be extending your Game Data System

**Design Patterns used:**

The NavMesh part of the AI doesn’t really have a specific design pattern assosicated with it. Every AI (character) has a reference to their Agent component and uses it appropriately (such as using Agent.SetDestination to move rather than manually translating using Vector2.MoveTowards).

The AI uses a very simple finite state machine using an AIState enum along with a switch statement to execute a certain action depending on the current AIState. See following pictures for a visualization of the two different AI behaviors in the game:

Diagram

Description automatically generated with medium confidence

Diagram

Description automatically generated

**Class Descriptions:**

Both the Nav Agent and AI Agent are handled through the Character class and KingCharacterDecorator class. The Character class has the AIState enum with different states for both non-king and king AIs. It also contains a method with a switch statement that calls certain methods depending on the current state (chosen from AIState). The Character class holds a reference to the Nav Agent and uses it to move (Agent.SetDestination).

The KingCharacterDecorator implements the methods that are used by the King since the King (specifically the castle king) will only use these methods and never use the non-king AI character methods hence the reason these methods are private. There is a similar switch statement that executes on of these behaviors depending on the character’s current state as set in the Character class. Because this class is just a decorator for wrapping a normal character, the Nav Agent for the King gotten via the wrapped Character class and is used accordingly (such as Agent.SetDestination to move when Flee method is called).

**Class Relationships:**

For a non-king character, there really isn’t a class relationship to any other class as this class also handles decision making for the AI (using the finite state machine), so all the AI thinking and Agent usage is handled within the Character class.

The KingCharacterDecorator depends on the Character class because it is created and used as a wrapper for a Character object. It uses the Character class’s reference to the Agent for its unique methods for the king’s movement. Also uses the Character class’s AIState to make decision for what the king AI should do.

**Dependency on other systems**

The KingCharacterDecorator is dependent on the Chracter class because it’s a wrapper (already explained dependency above).

The AI in Character class is dependent on the two Army objects because it makes decisions such as where to move, who to attack, etc. based on data from the two Army objects (example: the army tells the character the closet enemy based on a given position, and the Character AI will then target that enemy (go to it and attack it).

**Client Systems:**

There isn’t a client system because the Character class (and by extension KingCharacterDecorator) makes decisions itself using information given to it by outside classes (mostly Army class giving an object/enemy to target).

Local Multiplayer System

This will likely be an extension of the Player and Game State Systems.

**Design Patterns used:**

There is no specific design pattern used in the creation of local multiplayer. When local multiplayer is played, the multiplayer canvas in turned on in the CanvasManager singleton. Then, input is detected through button clicks or key clicks (key clicks through gameplaymanager singleton), and these inputs are given functionality via either the gameplaymanager or the SpawnHandler class.

The game reuses the same systems/patterns as described in the Player and Game State Systems part for the core gameplay.

**Class Descriptions:**

There are no unique classes implemented just for the local multiplayer. Instead, classes as described in the Player and Game State Systems section are built to work with both singleplayer and multiplayer.

In multiplayer (local), the gamestate manager will first say the game is multiplayer. Accordingly, the coinmanager and scoremanager both keep track of coins and score for each player. The gameplaymanager and UI buttons will then keep track of requests from either player to spawn a certain Character. And the CharacterManager will spawn these requests if the respective player has enough coins (info gotten from coinmanager).

Additionally, the SP\_PirateSpawner is not used because in multiplayer because Player 2 will control spawning pirates, not a predetermined list of pirate spawns.

**Class Relationships:**

The same class relationships exist as described in the Player and Game State Systems section. Simply, there was just additional functionality added to these already existing relationships to support two players playing at once (such as the coinmanager and scoremanager keeping track of coins/score for two players at once).

**Dependency on other systems**

Again, the dependencies are the same as described in the Player and Game State Systems section, just with some more functionality to account for two players instead of just one.

**Client Systems:**

See Player and Game State Systems section.

Opponent AI System

This will likely be an extension of the Player and Game State Systems.

**Design Patterns used:**

Composite Pattern (Tree Pattern)

**Class Descriptions:**

Designed for AI’s behaviour on single player mode. Letting the AI know how to make decisions (which button to click) depending on the current situation in game with its current gold.

**Class Relationships:**

Used CharacterManager's Army List in order to tell the current number of troops on the map, so the AI can compare both side's troop number. Used force spawn function from CharacterManager to spawn some troops at the start of the game without using money. Used CoinManager's CoinManager.\_instance.Coins[1]; in order to find the current coin that the AI has throughout the game, so the AI can spawn different level of troops depends on how much coin it has.

**Dependency on other systems**

Need army list from CharacterManager, force spawn function from CharacterManager, and coin amount from CoinManager.

**Client Systems:**

GameplayManager uses the AI behaviour script when the game is playing on single player mode, it also calls the Setup function for the AI behaviour.

A picture containing diagram

Description automatically generated

Networked Multiplayer System

While I initially wanted to add this to our game, after playing with it for a few days, I decided that with the limited time I had, it would be unlikely that I could finish the system in a way that would satisfy the vision I had of this system. As such, we didn’t implement this. However, I am interested in coming back to this some other time in the future, so I decided to keep the networked multiplayer/Photon code in our scripts (but nothing significant in this networked multiplayer/Photon code is ever called). For example, for spawning characters, there’s one method that uses Unity’s Instantiate() method and another method which calls PhotonNetwork.Instantiate() (the Photon method is never called). So, please ignore any code that deals with networked multiplayer/Photon because it is not “done” at the time this playtest is due. (Professor Salmon said it was okay to keep the unused code and to just note it here).

The following files will have the majority of the networked multiplayer code that isn’t used:

Character.cs

CharacterManager.cs

GameplayManager.cs

Launcher.cs

PhotonManager.cs

PlayerNameInputField.cs

But there may be a few more files with very minimal networked multiplayer code.

## Other Notes

TienYi - We didn’t include factory method for our UI element of notifications because there is only one type of notification so far (you don’t have enough coins), so we feel like it will be more efficient to turn a text on/off for that instead of creating a new notification every time. For the future, if we end up adding more notifications, we will be looking into implementing a factory method for that so we don’t have to keep track of too many things at a time.