Technical Design Document

## First 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:**

Currently, because the game only is able to support singleplayer at the time of the first playtest, the controls are purely based on clicking buttons on the UI, thus not facilitating any design patterns mentioned in the syllabus.

**Class Descriptions:**

In the main menu scene, there is one class that 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.

In the gameplay scene, 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.

**Class Relationships:**

The main menu scene 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)

**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). 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)).

**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:**

GameplayManager Singleton that uses delegates.

CoinManager Singleton.

ScoreManager Singleton.

CharacterManager Singleton that uses prototype.

**Class Descriptions:**

The GameplayManager handles the general game 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 gameplay manager relates to all the other managers 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 gameplay manager doesn’t really have a client.

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. It’s 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:**

Singleton CanvasManager with a delegate that can be called by other classes, which updates the displayed data (current coins and score).

**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 and MainMenu classes have 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).

**Dependency on other systems**

It has close ties with the coinmanager and scoremanager because it reads data from them and displays it.

It also has close ties to the gameplay scene spawn button class (SpawnHandler) because that class provides methods for the singleplayer/multiplayer canvas’s buttons to execute.

**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.

**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.

## 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, we plan to have more notifications, so 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.

TienYi - For the finite machine, we originally planned to have it in the game state manager but since we are only switching scenes for now using it, it is not a finite state machine. But we are looking to make it a finite state machine with special behaviors depending on the current state for the next playtest.