# Project Title:

# Castle War

## Team Member Names:

Yajie Hao, Qiuhan Li, Yehao Wang

## Team Member Quest IDs:

y35hao, q225li, y752wang

## Team Member Contributions:

- Qiuhan Li
  - Shop and inventory activities
  - Game art searching
  - Record game session
  - Share on social media platforms
  - Manipulate users' saved data
- Yajie Hao
  - GameMnager, GameConstant, GameScreens
  - Small portions of GameObject like Unit.animate()
  - Atomic classes and locks.
  - Combat mechanisms
- Yehao Wang
  - GameObject class as well as all levels of its subclasses.
  - System utility functions like scale Bitmap
  - Algorithms such as where should a unit go to and who should a unit attack
  - Algorithms such as how to divide a whole battle field into small sections

# Architecture Description

## We Now Support All Of Our Functional Properties!

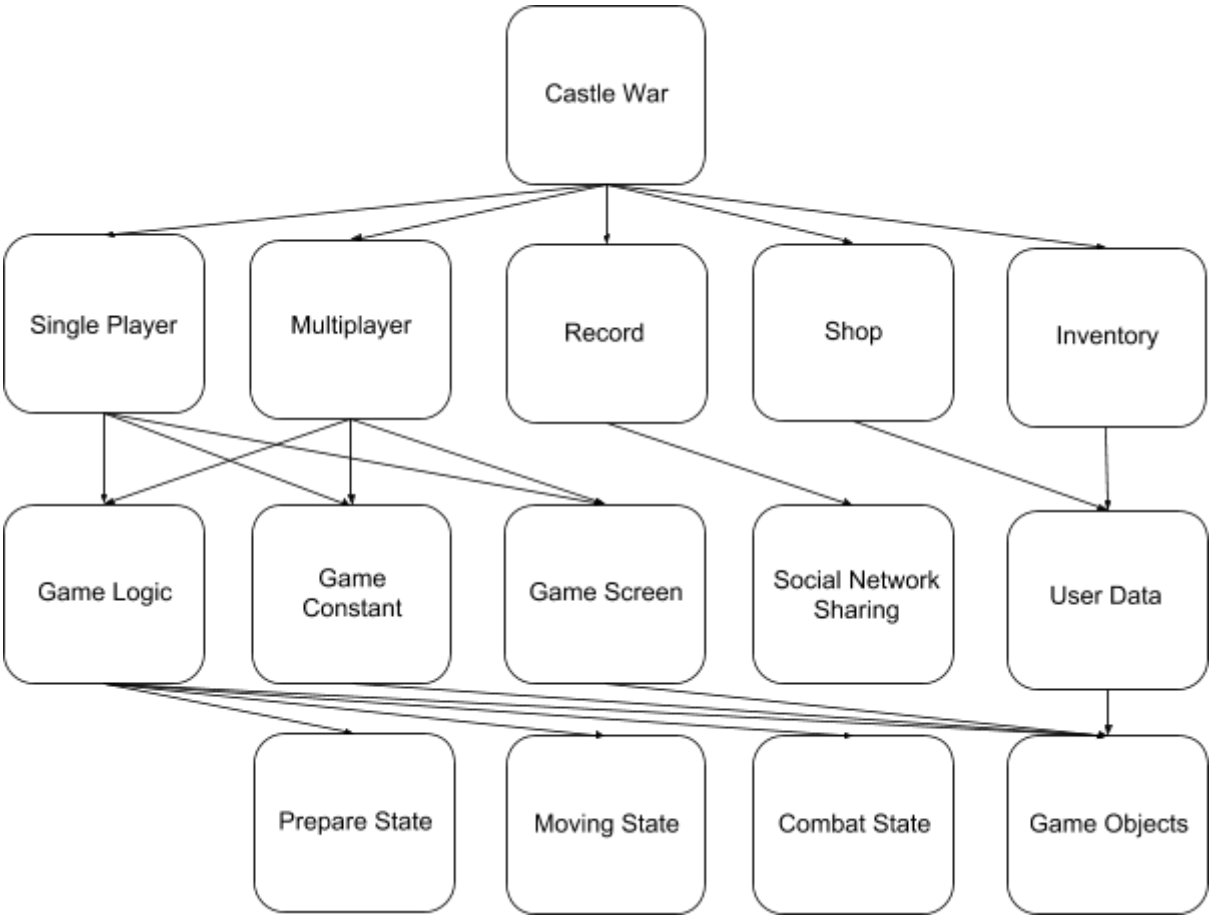
- Single Player Activity
    - When user enters the single player mode, all levels will be presented to users by corresponding buttons as well as pictures; descriptions for different level will be demonstrated within each button so that user is able to select different level by tapping these buttons
    - After user selects a specific level, three components will be initialized to support all activities that are going to happen within a game, these components are game logic, game constant and game screen
        - Game logic will control the entire process of the game, it decides which state that the user is currently located and whether the user's units as well as castles, enemy's units as well as castles are died or not so that it is able to let them disappear and terminate the entire game. In particular, there are three game states in the game which are prepare state, moving state and combat state
            - When the game is in the prepare state, game logic will wait to receive user's command and generates units as well as potions that are going to be employed during the next game
            - When the game is in the moving state, game logic will decide the direction and distance that each unit is going to move
            - When the game is in the combat state, game logic will determine the amount of damages that are going to caused by both sides of units after user's units attack enemy's units and enemy's units attack user's units; then, it will calculate the remaining HP that each user's unit as well as enemy's unit has
        - Game constant will be responsible for switching units' bitmaps, the switching process let different pixel on the background of the map to show pixels which belongs to these units in different point of time; it eventually causes users to see the movements of units on the background. In addition, it also handles the background music functionality so that user is going hear the music while playing the game; it calls different music files that corresponds to distinctive interfaces for which the user is currently at
        - Game screen will let each unit to be animated, users are able to view all units doing actions while they stay on the map. It is also responsible for providing distinctive actions that corresponds to moves and attacks during the game
    - During the combat process, user will be faced upon a custom Artificial Intelligence that is stored within each level. AI will randomly choose its units and potions to fight against user; within certain point of time, AI will select a group of much powerful units to beat users

    - Eventually, user will either win or lose the game. If user wins the game, he/she will get coins and potions as rewards. If user loses the game, he/she will get nothing consequently
- Multiplayer Activity
    - The system will support this functional requirements the same way as it supports single player activity, except the AI functionality which is excluded by the multiplayer activity. In this case, AI will be substituted by another actual human player. That other player will have the same set of functionalities from single player activity and the system will support as described above
- Shop
    - User is able to buy and sell different types of potions in the shop. The system will display item descriptions, prices to buy and sell items as well as different pictures that corresponds to distinctive items which are essentially potions. There is a buy button and a sell button in the interface that enables user to purchase and sell potions, data that shows the amount of coins user has will be updated by the system according to the item which user buy each time after he/she makes the purchase
- Inventory
    - User is able to check the information of different game objects in this activity. In the meantime, user is also able to see what kinds of potions that he/she originally has. The system will display game objects' names, descriptions as well as different pictures. Coin will be displayed at the top of the screen so that users can know how much money they have. If users tap on a particular icon, the detailed information about that game object will be displayed. For example, tapping on an ally will show its hit point, max hit point, attack and so on.
- Record
    - Every game session starts to be recorded automatically at the beginning of a new game, and ends recording automatically at the end of a game. In detail, a media recorder and a media projection are used to provide the functionality of recording screens of a device.
- Social Network Sharing
    - At the end of every game session, user is able to share his/her moments while playing each game to a social network platforms, a button will be presented so that user is able to tap and the moments from cache will be extracted to execute the process

# And We Also Guarantee Our Non-Functional Properties!

- Efficiency
    - Using background threads to process complex computations such as the whole game part is an efficient way to minimize responding time so that users can keep interacting with our application without being bothered by low responsiveness. For example, users can scroll screen right after choosing units, using items or ending his/her turn
    - Using separate threads for updating constant animations, background music, game states and game screen increases concurrency and reduces risks of conducting long-time tasks for our application when it runs on a multi-core CPU and etc.
    - Using ThreadPool instead of Thread or Asynchronous Task can reuse any available threads created ahead of time to avoid the expensive operations that create threads every time the game starts, resumes and pauses.
    - Using Atomic Primitives and ReadWriteLock ensures that the costs of normal locks and propagating caches to memories are minimized
- Reliability
    - Using the combination of ThreadPool, Futures and try/catch exception can guarantee that even though an error occurs, only the worker thread terminates and other threads including main thread remains unaffected. The lost data can be recovered by catching exceptions from the futures, storing states of all instances and restoring them later

# Component Diagram



# System Design

## Overview And Mappings Of Classes

* Please see the class diagram for the APIs of classes

- SinglePlayerActivity and MultiPlayerActivity
    - Basically these two activities are quite similar since what they need to do is to create a GameManager with different parameters, and then start the game
    - Mapping to SinglePlayer and MultiPlayer in the architecture
- InventoryActivity and ShopActivity
    - When started, these classes create all game objects that will be displayed and retrieve users' saved data from System. The creation will be done through static methods. For example, Ally.getAllAllyInstance() will return a list of Ally for the convenience
    - The only difference is that the former focuses on explaining what a gameobject is in detail and the later focuses on trading between Coin and Items
    - Mapping to Inventory and Shop in the architecture
- RecordAndShare
    - This class is actually composed of Android built-in APIs and resides in SinglePlayerAcitivity and MultiPlayerActivity. It starts recording at the beginning of game activity and stops recording when the game session is ended. Then a prompt will be given to users so that they can decide if they want to share on social media platforms such as Facebook.
    - Mapping to Record and Social Network Sharing in the architecture.
- RecyclerViewAdapter
    - This class is owned by every Activity except MultiPlayerActivity since there is no need to use RecyclerView. It is used to display multiple items in a list, where off-screen holders will be reused to increase efficiency
    - Mapping to SinglePlayer, Inventory and Shop in the architecture
- GameManager
    - This class is basically a blackboard where its clients can share information with each other through this class. It initiates and stores any game objects that are necessary for the users to start a new game, and keeps tracking the state of the whole game. In addition, this class creates three thread-based classes, GameLogic, GameScreen and GameConstant, and manage their life cycles
    - By the way, this class can be treated as UI thread which listens to user inputs, and update views of our application
    - Mapping to SinglePlayer and MultiPlayer in the architecture
- GameLogic

    - This class is a thread that updates all game objects in GameManager, manipulates game states, determines whether win, lose or death occurs, and making progress of the whole game
    - Mapping to GameLogic in the architecture
- GameConstant
    - This class is a thread that updates constant animations such as scrolling background, background music and so on. It hardly interacts with other clients of GameManager
    - Mapping to GameConstant in the architecture
- GameScreen
    - This class is a thread that draws every game objects in the scene on a canvas and pass the canvas to the GameManager(UI thread) so that UI thread can display the contents on the cavas
    - Remainder: this class cannot manipulate views directly since it is not executed on UI thread
    - Mapping to GameScreen in the architecture
- GameObject and its subclasses
    - GameObject contains basic information such as names, images, etc.
    - Subclasses usually override their parent's functions so that they exhibit slightly different functionalities. For example, Swordman override the method takeDamage so that it may block the half damage by a chance.
    - All mapping to GameObject in the architecture
- System
    - This class provides utility functions, stores game configurations and manipulates users' saved data
    - Mapping to UserData in the architecture

## Key Patterns And Data Structures

- Game Loops
    - This pattern is a so basic idea in game design, that every game will have at least one loop to wait for users' inputs, updates game object, updates screens and determines game states.
- Game Object
    - This is a basic data structure that a game can adapt. It means every game-related object should be a subclass of a game object so that common methods can be called easily such as update(). Such data structure also makes organizing game-related objects much easier because of common methods like upda().

## Residence And External API

All classes will of course reside in the users'/clients' devices since our application allows only one or two users interacting with exact one device.

We have one external API, which is used to achieve the functionality of sharing game sessions on social media platforms such as Facebook or Twitter.

## Rationale, Cohesion, Coupling And Comparison

Our project is inspired by the design patterns of other game engines such as Unity. Unity has a really nice organization of prefabs(game objects) that makes game development scalable, extensible, easy to understand and so on.

The first feature of this design is that it achieves low coupling by separating every classes so clearly that each of the classes can be implemented independently as long as a common API or a specific API is provided such as update(). For example, Game Manager does not need to know what is happening in Unit, just a call to move() will make that Unit modify itself. This is also the reason why it is called Game Manager instead of Game Modifier, which modifies game objects directly.

Then this design can guarantee high cohesion as well since every type of functionality is mapped to a class. For example, Unit cares only about itself, the battle field and opponent Unit, and even the use of a potion is executed through the method use(Unit) in the class Potion, which means Unit does not drink Potion actively, but Potion makes Unit drink. Therefore we can see each class focuses highly on its own functionality.

Last but not least, since this is a turn-based game we could have used another kind of design (but with the same data structure), which computes a whole turn until next loop instead of making many updates/frames in a second and keeps tracking of game states. Such design is easy to implement because it fits our intuitions, but later we gave up this design because one loop taking too long until next loop makes pausing and resuming impossible, even preventing our application from ending right away when users want to close our application.

## Future Adaption

In the future, we may be required to achieve remote matching among different devices. At that time, the server device needs to implement our whole design plus the ability to receive inputs from other devices through sockets, bluetooth and so on. However, on the client side clients only need to implement the functionality to receive screens-related information from server and still use GameScreen and GameConstant to display the whole game on the device.