<Name-of-Software-Application>

CS 230 Project Software Design Template

Version 1.2

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	11/12/24	Justin Perez	Added Executive Summary, Design Constraints, And Domain Model.
1.1	12/2/24	Justin Perez	Added Server Side, Client Side, & Development tools.
1.2	12/10/24	Justin Perez	Added Recommendations.

Instructions
Fill in all bracketed information on page one (the cover page), in the Document Revision History table, and below each header. Under each header, remove the bracketed prompt and write your own paragraph response covering the indicated information.

Executive Summary

Draw It or Lose It is loosely like the 1980s television game *Win, Lose or Draw*, where teams compete to guess what is being drawn. Rather than a player drawing images on an easel to help team members guess the puzzle (a phrase, title, or thing), the application will render images from a large library of stock drawings as clues. A game consists of four rounds of play lasting one minute each. Drawings are rendered at a steady rate and are fully complete at the 30-second mark. If the team does not guess the puzzle before time expires, the remaining teams have an opportunity to offer one guess each to solve the puzzle with a 15-second time limit.

Design Constraints

- A game will have the ability to have one or more teams involved.
- Each team will have multiple players assigned to it.
- Game and team names must be unique to allow users to check whether a name is in use when choosing a team name.
- Only one instance of the game can exist in memory at any given time. This can be accomplished by creating unique identifiers for each instance of a game, team, or player

System Architecture View

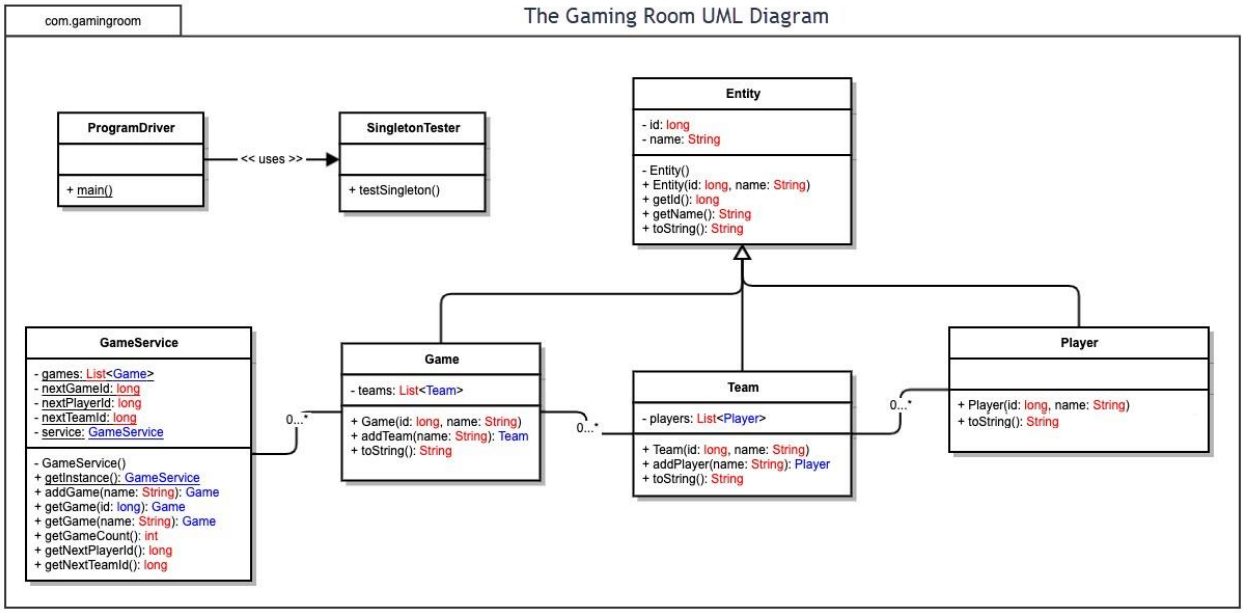
Please note: There is nothing required here for these projects, but this section serves as a reminder that describing the system and subsystem architecture present in the application, including physical components or tiers, may be required for other projects. A logical topology of the communication and storage aspects is also necessary to understand the overall architecture and should be provided.

Domain Model

- **Program Driver**
  - **Handles Main**
  - **Uses Singleton**
- **Singleton Tester**
  - **Test values for debug**
- **Game Service**
  - **Game service is mainly responsible for keeping track of game save data. It will handle the games along with the team names**
  - **Goes into Game**
- **Game**
  - **Game class holds all the team data mainly allowing it to be stored into game service and is responsible for the Active data that is currently in use while playing.**
  - **Gets information from Game Service and Team**
- **Team**
  - **Team mainly holds the data for the teams. It will keep track of the players on the team along with team ID and player names**
  - **Uses data from Player**
- **Player**
  - **Player holds each individual player then will use the team to put it inside a team**

- **Gives data to Team**

- **Entity**
  - **The Entity is mainly responsible for all the data currently in use like game.**
  - **Entities will do everything then distribute it onto the proper channels. For instance, Getting the players name and giving it to the player class.**
  - **Gives data to every other class except for program Driver singleton Tester and game service to an extent.**



Evaluation

Using your experience to evaluate the characteristics, advantages, and weaknesses of each operating platform (Linux, Mac, and Windows) as well as mobile devices, consider the requirements outlined below and articulate your findings for each. As you complete the table, keep in mind your client's requirements and look at the situation holistically, as it all has to work together.

In each cell, remove the bracketed prompt and write your own paragraph response covering the indicated information.

Development Requirements	Mac	Linux	Windows	Mobile Devices

Server Side	**Characteristics**: Mac OS has built in Support tools like mac os Server **Advantages**: Its relatively light weight compared to most other servers. It's also extremely simple to integrate it with the apple ecosystem **Weakness**: Very limited port ability. If designing for mac it almost always must stay on Mac. The hardware prices only rise by day and require a licensing cost.	**Characteristics**: Very popular webhosting and supports common industry practice like MYSQL. **Advantages**: Free and open source with one of the largest most dedicated community support. Being able to run on multiple devices. **Weakness**: Not manly people know how to operate linux so it's an initial learning curve as well as having an extremely complex set up. Due to it being open source there may be vulnerabilities.	**Characteristics**: Windows naturally supports webhosting with IIS. **Advantages**: One of the most popular OS so making a server on windows allows you to connect with other window devices easily. .NET Support. **Weakness**: High licensing cost. Is known to have way more outages compared to any other system. It's also worse in high traffic environments.	**Characteristics**: Not used for hosting but can be used for small things like P2P connections. **Advantages**: Portable. Has little cost. P2P connection. **Weakness**: P2P is known to only be as good as the main host internet. EXTREAMLY Limited Power and Storage. It also struggles with networks.
Client Side	**Pros**: Very easy to use after the learning curve. Once you understand MAC it's one of the easiest to use. **Cons**: Requires lots and lots of testing for mac OS. This can only be realistically done with MAC machines which can be expensive.	**Pros**: Affordability and no licensing cost as well as having lots of community support. **Cons**: Requires to be tested off multiple distributions which is Time consuming. Booting into Ubuntu, Fedora, and Kali will get annoying for every test. Security issues	**Pros**: Windows is on everything and is the most popular OS that uses some of the most popular browsers. **Cons**: Testing needed for Internet and explorer and edge even though multiple people will not use them. Visual studio can have a high price if working with multiple people.	**Pros**: The biggest userbase is reachable. **Cons**: Required to be tested on MULTIPLE DEVICES and OS. Samsung, Google, Mac, Windows, and even more. It also should be fast and responsive.

Development Tools	**Languages**: Swift, Xcode **Tools**: Reactive Native **Cost**: Mac systems are costly and are needed per person as well as testing. You could easily rack a huge build **Impact**: dedicated system for testing	**Languages**: Python, Java, PHP **Tools**: MySQL, Eclipse, VS code **Cost**: Nothing! It's all open source **Impact**: Requires being knowledgeable about Linux which is a BARELY known platform with most people as well as extremely confusing in the start. While the cost is nothing it will cost time finding competent workers.	**Languages**: C#, .NET, **Tools**: Visual Studio **Cost**: Windows licensing is expensive. **Impact**: Higher cost of programs but gives excellent support and is commonly used by everyone. You can find people who specialize in this relatively easily.	**Languages**: Java, Kotlin, Swift **Tools**: Flutter, React Native, Android studio, X code **Cost**: Everything is free except deployment depending on the app store you'd like to join you will have to pay a high fee. **Impact**: Requires skilled mobile developers as mobile is unlike PC at all.

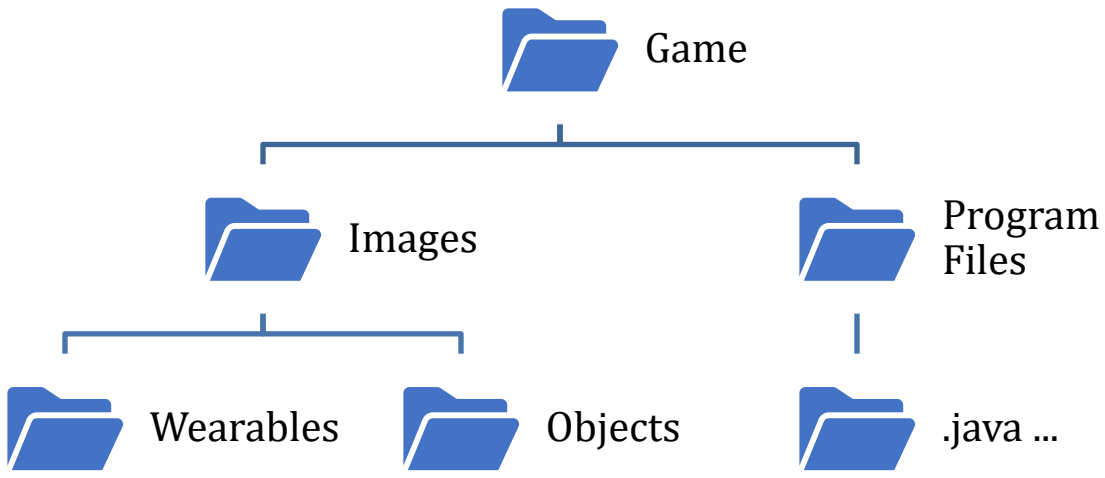
Recommendations

Analyze the characteristics of and techniques specific to various systems architectures and make a recommendation to The Gaming Room. Specifically, address the following:

1. **Operating Platform**: Linux should be the be the clear winner out of the all the options due to the minimal License cost and does not limit access to data centers.
   Front and end back end could connect with each other using APIs, Allowing front end to work independently to the back end.
   A perk of using Linux is since the front end is using an API to talk to the Linux server, we can develop in the best language for each OS we would like to port it too. For instance, we can use .NET if we want to bring it to windows but also use Java for Android.
2. **Operating Systems Architectures**: The Setup I suggest is letting backend manage game scores and photos and allowing front end to only call the folder they selected randomly to use for the round then pick out a new folder. While this could cause issues with latency the game is always on a timer, we can expect a player to take a few seconds to see the image before making a correct guess giving more than enough time to get this done in the background However if we do, we can put a timer saying you have 5 seconds before you can guess.
3. **Storage Management**: For the server-side management. My biggest suggestion is to look for cloud native tools. These offer flexibility and ensure a smooth experience for all players visiting the game. It also is expandable with multiple storage options in case you would like to add more to the game in the future.

   As the game will not be downloaded mainly there is no need for client-side things like SSDs and HDDs.

   Arranging your file structure like this may lead to good results as well in the future.



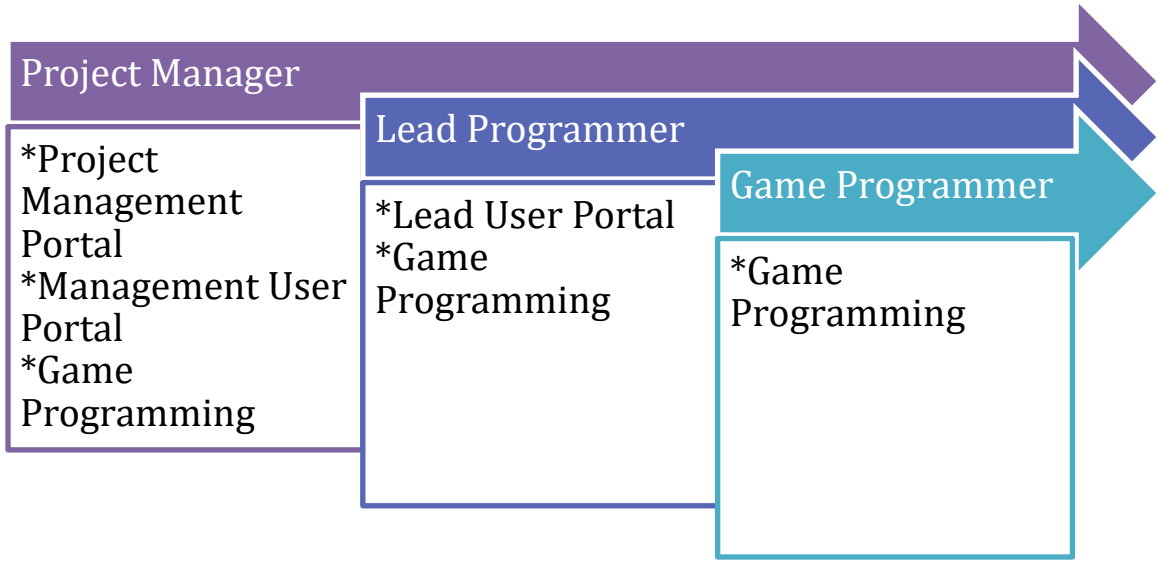
4. **Memory Management**: Clients due to not needing to download anything will mainly be playing off their RAM. The clients RAM should have the application they use to view to such as a browser for example and 3-5 preloaded pictures.

   On the server side it will depend on the system we are developing for.

   For example Android uses ART and Dalvik VM: Which allow for easy memory allows the client to immediately connect to the folders without copying everything into ram.

5. **Distributed Systems and Networks**: Cloud-Native architecture is the way to go if your goal is to have a game that can be enjoyed by multiple different devices. Cloud providers ensure good security as well as being able to move to different locations whenever a client requests it. It also prevents outages as Cloud-Native providers not only handle your game but multiple others if one goes down, they all go down so you can be sure they will work their hardest to ensure it will be up and running again.

6. **Security**: Security will be Role based Authorization. This means that Users will have different options available based on level. They can also be promoted by people above them or demoted by people above them. An example of roles could look like this.



(Some Notes)

No User will EVER be allowed as an admin as it's a high security risk to have too much power.

APIs will be protected using encryption SHA 256.

A firewall should also be added as part of the server.