<Name-of-Software-Application>

CS 230 Project Software Design Template

Version 1.0

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	<mm/dd/yy>	<Your-Name>	<Brief description of changes in this revision>

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

The Gaming Room is planning on expanding its game Draw It or Lose It from an android only application to a web-based app to be used on multiple platforms. The goal is to design a scalable and secure architecture that supports gameplay for multiple players and teams while ensuring only one active version of the game exists in memory. By using a centralized game service manager to maintain uniqueness across all instances of the game and prevent conflicts. This design will be very efficient, modular, and support future growth and scalability.

Requirements

< Please note: While this section is not being assessed, it will support your outline of the design constraints below. *In your summary, identify each of the client's business and technical requirements in a clear and concise manner.>*

Design Constraints

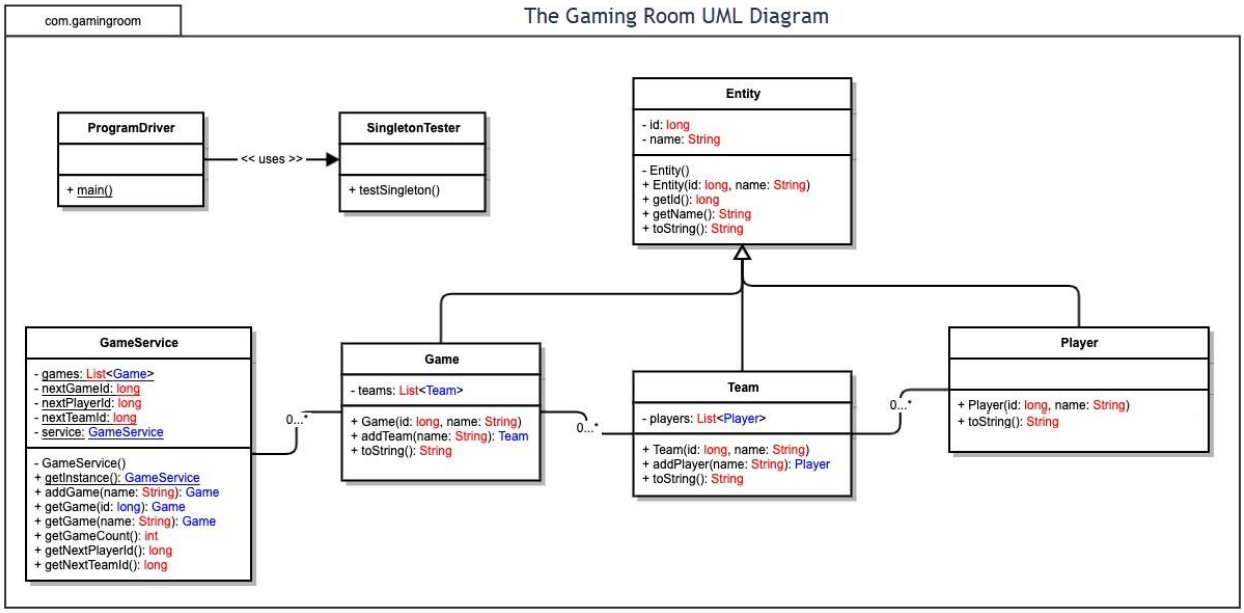
1. Single active game instance
    a. The system will have a singleton pattern for the game instance which makes sure no other game can coexist in the memory.
2. Unique naming/identification
    a. Having unique identifiers for game, team, and player names means each identifier can be validated to avoid conflicts of information.
3. Web based distribution system
    a. The application must work in a web-based environment, things like session management, data synchronization, and client-server communication must all be considered.
4. Cross platform compatibility
    a. The application must be accessible from windows, mac, Linux, and mobile devices.
5. Security and performance
    a. The application must be optimized to keep user data secure and without compromising strong performance.

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

<Describe the UML class diagram provided below. Explain how the classes relate to each other. Identify any object-oriented programming principles that are demonstrated in the diagram and how they are used to fulfill the software requirements efficiently.>



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	Can host servers, but not ideal for large-scale hosting. Better for development and testing.	Great for hosting web apps. Common for scalable, cloud-based servers.	Can host with IIS or on Azure. Used in enterprise but less common for startups.	Not server platforms — these are client platforms. Need a server to connect via API.
Client Side	Supports major browsers. Responsive design still needed.	Works well with all modern browsers. Needs responsive design.	Compatible with all browsers. Requires responsive design.	Must be mobile-friendly, touch-optimized, and tested on iOS and Android browsers.
Development Tools	Needed mainly for iOS builds. At least one Mac required. Tools and languages: Xcode (for iOS), Node.js, React, Swift. IDE: VS Code. Licensing costs: High cost due to hardware and licensing.	One full-stack team can handle both front-end and back-end. Languages and tools: Node.js, React.js, Docker, Git. IDEs: VS Code, Eclipse. Licensing costs: Free and open-source. Low cost if using cloud services.	Might need .NET experience. Can still use standard web tools. Tools and languages: NET (optional), Node.js, React. IDE: Visual Studio. Licensing costs: Licensing and Azure can be expensive.	Use cross-platform tools (e.g., React Native) to save time and cost. Still need testing on both platforms. Tools and languages: Web: React, Angular, Vue. Native: Swift (iOS), Kotlin/Java (Android). Licensing costs: N/A

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**: To help Draw It or Lose It grow and run on more devices, I recommend using a Linux server (like Ubuntu Server) hosted on a cloud service such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud. These platforms are reliable, flexible, and cost-effective. Linux works well with tools like Docker, which make it easier to move the game to different devices. Cloud services also make it easy to add more power when more users play the game and keep the game running smoothly with very little downtime. To help Draw It or Lose It grow and run on more devices, I recommend using a Linux server (like Ubuntu Server) hosted on a cloud service such as Amazon Web Services (AWS), Microsoft Azure, or Google Cloud. These platforms are reliable, flexible, and cost-effective. Linux works well with tools like Docker, which make it easier to move the game to different devices. Cloud services also make it easy to add more power when more users play the game and keep the game running smoothly with very little downtime.

2. **Operating Systems Architectures**: Linux servers use something called a monolithic kernel, which means the core parts of the system run together in one place. This helps the system run fast and allows us to add or change parts without restarting. It can handle many users and tasks at the same time, which is important for online games. On users' devices, the game will run on systems like Windows, Android, and iOS. Each of these has a slightly different way of working, so we can use tools like Unity or Flutter to build one version of the game that works across all platforms.

3. **Storage Management**: For storing game data, we'll use Linux file systems like ext4 or XFS, which are safe and reliable. These systems keep track of changes, so we don't lose anything if the server crashes. We'll also use cloud storage (like AWS S3) to save user drawings and media files. For storing scores, game settings, and user information, we can use databases like PostgreSQL or MongoDB. These storage options allow us to back up data easily and control who can access what.

4. **Memory Management**: The Linux server manages memory in smart ways. It breaks memory into pieces and gives each program what it needs. It only loads parts of the game when needed to save space, and it can use "swap space" on the hard drive if memory runs low. For multiplayer games, it can share memory between users to keep things running fast. These features help the game work smoothly, even when a lot of people are playing.

5. **Distributed Systems and Networks**: Since the game will be played on different types of devices, we'll use a distributed system, which means different parts of the game run on different computers but still work together. Game servers will handle matches and keep track of players. We'll use secure connections (like HTTPS) to send data between users and servers. To make sure things keep running even if a part fails, we'll use backups, automatic restarts, and tools that send game data in the background. This setup also helps the game respond quickly to players all over the world.
6. **Security**: Keeping user data safe is very important. We'll use encryption so that all personal information is protected when stored and sent between devices. Users will log in using secure systems like OAuth2, which is what big websites like Google use. We'll also limit who can access

different parts of the system using role-based access control, which means users and workers can only see what they need to. The cloud service will help block attacks, and we'll watch for strange behavior using monitoring tools. On phones and other devices, we'll store data safely using the security tools built into iOS and Android.