Draw It or Lose It

CS 230 Project Software Design Template

Version 1.0

Table of Contents

Document Revision History

Version	Date	Author	Comments
1.0	08/14/25	Michael Marquardt	Initial Draft

Executive Summary

The Gaming Room is looking to develop an application version of "Draw It or Lose It", a drawing guessing game inspired by the TV show "Win, Lose or Draw." This document outlines the software architecture, technical requirements, and development approach needed to create a scalable, real-time multiplayer gaming platform.

Design Constraints

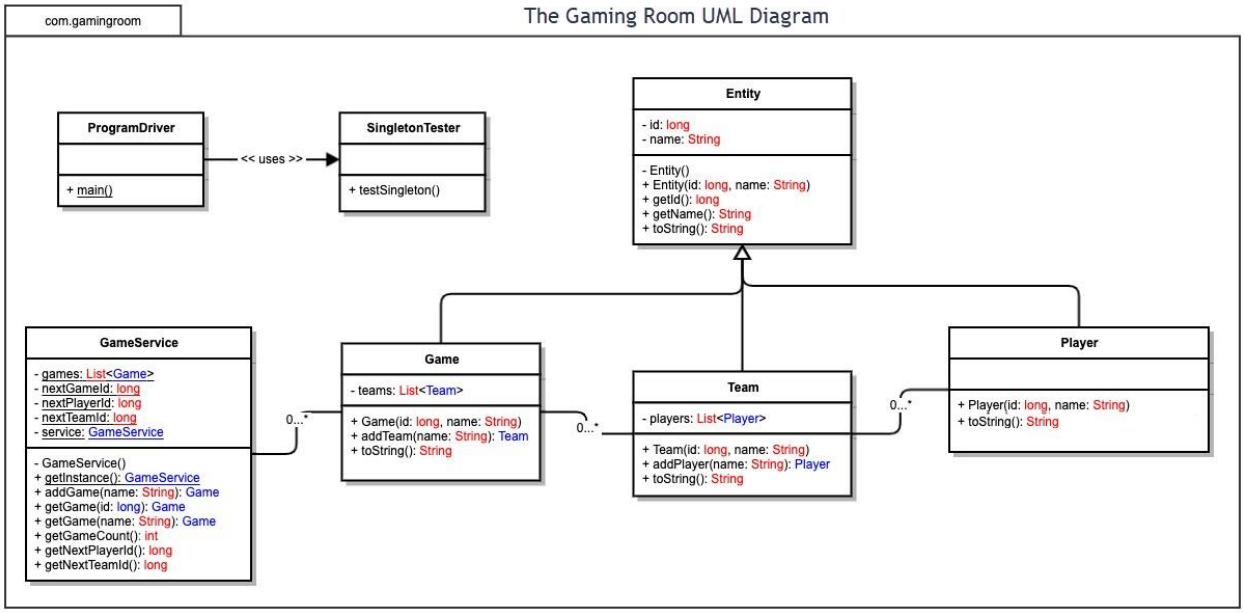
The web-based distributed environment imposes several critical constraints on Draw It or Lose It development. Real-time synchronization requirements necessitate WebSocket connections and server-side authoritative timing to ensure fair 60-second rounds and 30-second progressive image rendering across all clients. The game requires web standard compliance for cross-platform compatibility because it cannot use platform-specific features thus needing extensive multi-browser testing. The system requires horizontal scaling capabilities and stateless design patterns to handle concurrent game sessions at scale. The progressive image system must use adaptive quality settings and CDN optimization because network latency and bandwidth variations affect its operation. Web applications require server-side validation of all game actions and encrypted communications and anti-cheat measures to protect against security vulnerabilities which increases development complexity but maintains game integrity and user trust.

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

The UML class diagram demonstrates a well-structured object-oriented hierarchy with Entity as the base class providing common attributes (id, name) inherited by Game, Team, and Player classes. GameService implements the Singleton pattern to ensure centralized game management with a single instance containing methods for game creation, retrieval, and ID generation. The design shows clear composition relationships where GameService contains multiple Games (0..), each Game contains multiple Teams (0..), and each Team contains multiple Players (0..*), effectively modeling the hierarchical gaming structure. The design implements essential OOP principles through inheritance (Entity base class), encapsulation (data management within each class), composition (parent-child relationships) and the Singleton pattern (GameService) which results in a scalable and maintainable structure that efficiently supports multiplayer gaming requirements and ensures unique identification and proper resource management across distributed instances.



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	OS X Server is available for Mac but unless the client wants to purchase their own hardware, finding hosts can be difficult and expensive. OS X Server is $499 USD for 10-clients. Or, $999 for unlimited.	Linux is the most popular webhosting OS. Because Linux is opensource, maintenance and license costs tend to be cheaper than cloud OSes like Windows. Moreover, typical cloud providers like Google or Amazon offer Linux preferentially over Windows.	Windows servers are nice because they are GUI based, and many applications used in the office will also run on the server — so familiarity is abundant. License costs, typically per user, tend to be very high — especially compared to Linux. Windows server licenses range from $6,200 (up to 16 core licenses) to $500 (up to 50 clients) per installation per year. Hosting platforms may be more limited as compared to Linux	Mobile devices can be used as a personal webserver or file server, but they are not equipped for multi-user serving. The hardware is typically more limited, e.g., RAM, and they are not scalable like blade servers. Cost is unknown as the hosting tools would probably need to be designed and built in-house.

Client Side	Macs are generally costlier than Windows. Ease of use is about the same as Windows, requiring a short to moderate amount of time to learn with an intuitive interface. To develop for Macs, you need a Mac computer running the latest version of Xcode. Moreover, the macOS SDK is in Objective-C or SWIFT which are lesser-known languages. Lastly, Windows usage is 75% of the market vs. macOS's 16%. This presents a smaller market opportunity. Update: Consider hiring an independent React-native	Linux is ideal for software and web developers because of its cost-effectiveness, and open-source programs that work in unison with the system. However, a maximum amount of time is required to learn compared to Mac and Windows defaults. Development in Linux should be straightforward as Java or C/C++, or Python could be the language of choice — which are all commonly used. Moreover, multi-users support is available on the GNU/Linux platform. GNU/Linux development might have little value as there is no widespread use. Update: Since React-	Compared to Linux and Mac, Windows has many unique tools that can only be virtualized on the other systems and has extensive support for web-app and website development. Can also virtualize other OS. Windows is typically developed using C# or .NET which are both common. There would be no barrier to entry to development of a Windows client application. Windows has been a native multi-user platform since Windows XP. Windows is the preferred OS for 75% of computer users which makes for a better business case. Update: React-native for web codebase can also be worked	Provides clients flexibility of having the app anywhere, anytime. Requires adjustments in developing the app for screen real estate differences. All screen size possibilities for tablets, smartphones, and browsers should be accounted for. Mobile apps should have an intuitive interface designed for their small form factor. Mobile devices are not designed to be multi-user. However, designing a client application for Android or iOS is straightforward. Android SDK is Java based so code developed for Windows and Linux might be able to act as a jumping off point. iOS is SWIFT based so the same requirements for Mac apply, including the hardware needs. Update: Utilizing

	web developer for under $5k. A hybrid app can be used rather than a native app — allowing for portability (one code base, multiple platforms), faster speed to market and cheaper origination costs.	native web can be implemented to support iOS, Android, and Web platforms with a single codebase, this solution is optimal for the front-end client interaction, and front-end JWT-based Authentication can connect to our Flask or Firebase database. The Canvas drawing and chat portion can be streamed over a secured Websocket living on our server or a Heroku instance.	on in a Windows environment using Visual Studio Code. Summary of development needs: Flask dev, React native web dev, Security dev for proper authentication, upscale with cloud-based solutions	react will allow responsive media adjustments to conform to iOS and Android phone form factors and scale up the interface for web

Development Tools	Mac uses Objective-C and SWIFT for development languages. Xcode is the common IDE used for Mac development. Xcode is listed as $99 USD per year per developer.	Linux development may take the form of C/C++, Java, or Python. Python IDEs are often free, e.g., Notepad++. PyCharm is another popular Python IDE. C/C++ IDEs are numerous — but not all are available for Linux. Eclipse can do all of these and is free. VSCode for syntax highlighting and code previews, Homebrew package manager to install Unix and Mac utilities, Xcode IDE, iTerm2 emulator, Tower git client, Dash API browser. In addition, platform agnostic language set: JavaScript, HTML, CSS, React, react-native, react-native-web. Chrome	Windows is primarily developed using C# and primarily .NET. Microsoft's Visual Studio is an immensely popular IDE and offers many plugins and integration options, e.g., Jenkins, TestComplete, etc. Visual Studio ranges from $45 – $250 USD per user, depending on features, per year. Visual Studio Code, Atom, Vim, bash command line, git, node, flask. In addition: JavaScript, HTML, CSS, React, react-native, react-native-web, npm, yarn. Chrome development tools, MySQL Update: Firebase, Amazon AWS, Heroku	Android SDK is Java based, and the most widely used Android IDE is Android Studio which is developed by Google as the official development tool. Android Studio is free to download. iOS's Objective-C and SWIFT languages are almost exclusively developed in Xcode. Xcode is listed as $99 USD per year per developer. Browsers: Firefox, Opera, Samsung browser, Chrome, Metro browser. The website should work across all mobile browsers. Mobile browsers required for testing both the website and the app itself. App: JavaScript should be enabled on iOS and Android to allow app access. Google Play Store and Apple App Store.

		development tools, SQL. Update: Flask, node.js, WebSocket, MySQL		

Recommendations

Operating Platform:
I recommend Linux as the operating platform for the "Draw It or Lose It" game expansion. I believe Linux servers are the better choice because they eliminate licensing costs that Windows requires, and I think this will provide significant cost savings. I recommend Linux because it offers good security and operability while being the most common server platform, which means there will be access to many available tools including security software. I also recommend Linux because it doesn't limit access to data centers the same way that Windows servers can, giving more flexibility in deployment options. While I acknowledge that Windows has a large userbase and offers good integration with existing Android applications, I believe the cost efficiency and operational flexibility of Linux make it the superior choice for your server operating platform needs.

Operating Systems Architectures:
I recommend the microservices architecture approach for your "Draw It or Lose It" game system architecture. I believe a modern backend running containerized microservices with Kubernetes or Docker is the best choice because it allows for better scalability as the game expands across platforms. I recommend this backend server architecture that manages the game environment with frontend/client-based rendering because it's perfectly suited for the game, I think the asynchronous communication approach will work excellently. I strongly recommend this architecture because choosing to use the frontend for rendering allows your server to offload resource-intensive parts of the application, which I believe will reduce the monthly data center costs. I think this client-side rendering approach will also insulate gameplay from network issues since framerate is important to gameplay, and I recommend it because clients can cache subsequent images ahead of active gameplay to ensure smooth rendering. While I acknowledge that the hybrid architecture offers good customization and improved security over singular approaches, I believe the containerized microservices architecture provides better cost efficiency and modern cloud-native capabilities that I think will better support the multi-platform expansion needs.

Storage Management:
I recommend the cloud-native storage approach for the "Draw It or Lose It" game storage management. I believe that unless The Game Room wants to purchase their own hardware, they won’t need to make decisions about storage medium like HDD vs SSD, and I think either option should provide the performance needs of your application, especially with caching behavior and client-side rendering. I recommend using cloud-native tools on the server-side because I believe this will add the flexibility needed, particularly where scalability or localization is concerned. I think this approach is superior to purchasing dedicated hardware because I believe cloud-native storage will scale more efficiently with user growth and provide better geographic distribution options for multi-platform game expansion. While I acknowledge that Microsoft Azure offers competitive pricing and good integration with Windows systems, along with various storage options for your 200 8MB images, I recommend the cloud-native approach because I believe it provides

better flexibility and won't lock the company into a specific vendor ecosystem, giving it more options as the game grows across different platforms.

Memory Management:

I recommend the Linux memory management approach for "Draw It or Lose It" memory management. I believe Linux's use of pagecache for data stored in main memory and demand paging is superior because it allows for lower memory usage since pages not actively being used will not be loaded into memory. I recommend this approach because Linux uses the Least Recently Used (LRU) algorithm for page replacement, which I think will be more efficient for the game's needs. I strongly recommend this memory management strategy because with client-side rendering, there will be minimum need for the amount of RAM on the server, and I believe that if a modern architecture with containers and microservices is used, the cost will scale appropriately with the number of users. I think the client-side RAM requirements will also be minimal since only 1-2 images need to be stored in memory at any given moment plus the RAM needed to drive the client application. While I acknowledge that Windows offers virtual and physical address space for memory allocation and newer versions like Windows 10 allow each process to utilize virtual memory address space in its entirety, I recommend the Linux approach because I believe its demand paging and efficient memory management will provide better performance and cost efficiency for your multi-platform game expansion.

Distributed Systems and Networks:

I recommend the RESTful API communication approach for "Draw It or Lose It" distributed systems and networks implementation. I believe that having the frontend and backend communicate through RESTful APIs asynchronously is the superior choice because RESTful API usage allows the client/server communication to be transparent to the deployed frontend, whether it's Android, Windows, or iOS. I recommend this approach because I think it provides the flexibility needed for your multi-platform expansion while maintaining clean separation between systems. I strongly recommend this distributed architecture because I believe uptime considerations and outage prevention are critical, and many cloud providers can replicate and shift services amongst different deployments to prevent large-scale outages. I think this cloud-native approach will provide better reliability for the game. While I acknowledge that Azure Cloud services offer maximum uptime using cloud-based email alerts, App Insight Logging and Monitoring Service, and can help reduce time spent on network loads through automation, I recommend the RESTful API approach because I believe it gives more flexibility across different cloud providers and won't lock into a specific vendor's ecosystem for distributed systems and network management needs.

Security:

I recommend the role-based authorization security approach for "Draw It or Lose It" game security implementation. I believe that security consisting of role-based authorization is the superior choice because it will require creating an entitlements interface for effective administration of roles and accounts. I recommend employing the idea of least-privilege, which I think should limit users in their scope to game controls like game creation, team name creation, and team enrollment, with the possibility of extending user scope into a team-captain/member hierarchy if needed. I strongly

recommend this security approach because I believe no user should be allowed as an ADMIN on the system, and I think APIs should be protected using encryption with SHA 256, 128-bit keys, with TLS below 1.2 disallowed and certificates purchased from Entrust. I recommend adding a firewall as part of the server using industry-standard best practices for default settings. While I acknowledge that Azure's App Service with Active Directory offers good features like IP configuration limitations, VPN storage options, and database security with SSL connectivity requirements, I recommend the role-based authorization approach because I believe it provides better security control and flexibility without vendor lock-in, allowing more options for protecting user information across multi-platform game expansion.