# The Gaming Room - Complete Software Design Document

Prepared for: The Gaming Room  
Course: CS-230 - Operating Platforms  
Institution: Southern New Hampshire University

## Project One: Software Design Template

## Project Two: Architectural Design and Evaluation

## Project Three: Final Recommendations and Integration

![The Gaming Room logo](data:application/vnd.ms-word.stylesWithEffects+xml;base64,)

**Draw It or Lose It**

# **CS 230 Project Software Design Template**

Version 1.0

Table of Contents

[CS 230 Project Software Design Template](#bookmark) 1

[Table of Contents](#bookmark1)**2**

[Document Revision History](#bookmark2)**2**

[Executive Summary](#bookmark3)**3**

[Requirements](#bookmark4)**3**

[Design Constraints](#bookmark5)**3**

[System Architecture View](#bookmark6)**3**

[Domain Model](#bookmark7)**3**

[Evaluation](#bookmark8)**4**

[Recommendations](#bookmark9)**5**

[Document Revision History](#bookmark10)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | September 20, 2025 | Bryan Paradise |  |

**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 (TGR) wants to expand its Android game Draw It or Lose It into a web-based, multi-platform version. The new application must support multiple games simultaneously, where each game can have one or more teams, and each team can have multiple players. To keep the experience consistent and prevent duplicates, game and team names must be unique, and the application must ensure only one game service instance is used to manage in-memory state.

Our solution uses a Singleton GameService to provide a single authority for creating and retrieving games, teams, and players. We enforce unique names and provide efficient lookups using the Iterator pattern when adding or searching for entities. We also introduce a shared Entity base class (id, name) to reduce duplication and keep the model clean and extensible.

Design Constraints (web/distributed)

* Single authoritative service object (Singleton): Only one GameService lives in memory per process, avoiding conflicting sources of truth and simplifying concurrency in this first prototype.
* Uniqueness rules: Names for games and teams must be unique (and we also enforce unique player names). This constrains how we add entities—every insert must iterate current collections to check for duplicates before creating new objects.
* Stateless web requests, shared model: In a web deployment, requests are stateless. The authoritative in-memory model must be centralized or backed by persistence later; for now, the Singleton encapsulates state to keep logic consistent.
* Extensibility: New features (persistence, scorekeeping, rounds, authentication) must fit the same domain model. The Entity base class and “service manages collections” approach make that straightforward.
* Performance & Safety: Iteration on small collections is fine for the prototype. In production, we’d add indexes or persistence (DB), validation, and security controls (input validation, auth).

Implications:

* Developers must acquire the service via GameService.getInstance().
* All add/get operations go through the service so uniqueness is enforced in one place.
* Using iterators satisfies the course requirement and keeps code clear and portable.

Domain Model (UML, OOP principles)

* Classes & relationships

Entity (base): id: long, name: String

Game extends Entity: has 0..\* Team

Team extends Entity: has 0..\* Player, belongs to one Game

Player extends Entity: belongs to one Team

GameService (Singleton): manages the collections of Game, Team, Player, creates unique IDs, enforces name uniqueness, provides lookups.

* OOP principles in use

Inheritance: Game, Team, Player inherit common id/name from Entity.

Encapsulation: Fields are private/final; access via getters; state changes only through GameService.

Composition/Aggregation: A Game contains teams; a Team contains players.

Single Responsibility: GameService manages lifecycle/uniqueness; entities hold data/associations.

Polymorphism: Shared toString() style and common base type allow uniform handling if needed.

**!["The Gaming Room UML diagram. The top of the diagram is labeled as com dot gamingroom. Test boxes are placed in two layers. The first layer has three text boxes and the second layer has four of them. In the first layer, the 'ProgramDriver' textbox points to 'SingletonTester' textbox. The 'ProgramDriver' textbox contains the text 'asterisk main round brackets.' The 'SingletonTester' textbox contains the text 'asterisk testSingleton round brackets.' The arrow between these two text boxes are labeled 'open two angle brackets uses close two angle brackets'. In the second layer, there are 'GameService', 'Game', 'Team', and 'Player' text boxes. The 'GameService' textbox has texts arranged in two layers. The first layer contains games colon List open angle bracket Game close angle bracket, nextGamesId colon long, nextPlayer Id colon long, nextTeamId colon long, and service colon GameService. The second layer contains GameService round brackets, getinstance round brackets colon GameService, addGame open parenthesis name colon String close parenthesis colon Game, getGame open parenthesis id colon long close open parenthesis colon Game, getGame open open parenthesis name colon String close open parenthesis colon Game, getGameCount round brackets colon int, getNextPlayerID round brackets colon long, and getNextTeamId round brackets colon long. The 'GameService' box is connected with the 'Game' textbox with a line labeled 'zero dot dt dot asterisk'.  The 'Game' textbox also contains text in two layers. The first layers contains the text teams colon List open angle bracket Team close angle bracket. The second layer has Game open round bracket id colon long comma name colon String close parenthesis, addTeam open parenthesis name colon String close parenthesis Team, toString round brackets colon String. The 'Game' textbox is connected with the 'Team' textbox with a line labeled 'zero dot dt dot asterisk'. The 'Team' textbox also contains text in two layers. The first layers contains the text players colon List open angle bracket Player close angle bracket. The second layer has Team open parenthesis id colon long comma name colon String close parenthesis, addPlayer open parenthesis name colon String close parenthesis colon Player, and toString round brackets colon String. The 'Team' textbox is connected with the 'Player' textbox with a line labeled 'zero dot dt dot asterisk'. It contains the text Player open parenthesis id colon long comma name colon String close parenthesis and toString round brackets colon String. The 'Game', the 'Team, and the 'Player' boxes point to the 'Entity' textbox in first layer. The 'Entity' textbox contains text in two layers. The first layer has the text id colon long and name colon String. The second layer has Entity round brackets, Entity open parenthesis id colon long comma name colon String close parenthesis, getId round brackets colon long, getName round brackets colon String, toString round brackets colon String.](data:application/vnd.openxmlformats-officedocument.wordprocessingml.settings+xml;base64,)**

[Evaluation](#bookmark11)

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** | <Evaluate Mac for its characteristics, advantages, and weaknesses for hosting a web-based software application.> | <Evaluate Linux for its characteristics, advantages, and weaknesses for hosting a web-based software application.> | <Evaluate Windows for its characteristics, advantages, and weaknesses for hosting a web-based software application.> | <Evaluate Mobile Devices for their characteristics, advantages, and weaknesses for hosting a web-based software application.> |
| **Client Side** | <Determine the software development considerations (cost, time, expertise) that are necessary for supporting multiple types of clients as they pertain to Mac.> | <Determine the software development considerations (cost, time, expertise) that are necessary for supporting multiple types of clients as they pertain to Linux.> | <Determine the software development considerations (cost, time, expertise) that are necessary for supporting multiple types of clients as they pertain to Windows.> | <Determine the software development considerations (cost, time, expertise) that are necessary for supporting multiple types of clients as they pertain to Mobile Devices.> |
| **Development Tools** | <Identify the relevant programming languages and tools (IDEs and other tools) that are used to build this type of software for deploying on Mac.> | <Identify the relevant programming languages and tools (IDEs and other tools) that are used to build this type of software for deploying on Linux.> | <Identify the relevant programming languages and tools (IDEs and other tools) that are used to build this type of software for deploying on Windows.> | <Identify the relevant programming languages and tools (IDEs and other tools) that are used to build this type of software for deploying on Mobile Devices.> |

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**: <Recommend an appropriate operating platform that will allow The Gaming Room to expand Draw It or Lose It to other computing environments.>
2. **Operating Systems Architectures**: <Describe the details of the chosen operating platform architectures.>
3. **Storage Management**: <Identify an appropriate storage management system to be used with the recommended operating platform.>
4. **Memory Management**: <Explain how the recommended operating platform uses memory management techniques for the Draw It or Lose It software.>
5. **Distributed Systems and Networks**: <Knowing that the client would like Draw It or Lose It to communicate between various platforms, explain how this may be accomplished with distributed software and the network that connects the devices. Consider the dependencies between the components within the distributed systems and networks (connectivity, outages, and so on).>
6. **Security**: <Security is a must-have for the client. Explain how to protect user information on and between various platforms. Consider the user protection and security capabilities of the recommended operating platform.>

**Evaluation — Server Side**

**Mac (Server Side)**  
MacOS *can* host a website, but it’s not a common or practical choice for real production servers. Apple only allows macOS on Apple hardware, so scaling in the cloud is hard and pricey. It’s great for developers’ laptops and small tests, but for a site with thousands of players, Linux or Windows Server is a better fit because they have wider hosting options and better support.

**Linux (Server Side)**  
Linux (like Ubuntu or Debian) is the standard for running web apps. It’s fast, stable, and works well with containers like Docker and Kubernetes. Most tools for building and deploying web apps are designed with Linux in mind. Community versions are free, and every major cloud has Linux images ready to go. For our Java API and web stack, Linux gives the best performance for the cost and scales smoothly.

**Windows (Server Side)**  
Windows Server can also host our app (often using IIS or a reverse proxy) and it ties in nicely with Active Directory. This is helpful if the company already uses Windows tools and .NET. The downsides are license costs and a bit more overhead for Java/container setups compared to Linux. If the team already runs Windows servers, it’s fine; if not, Linux is usually simpler and cheaper.

**Mobile Devices (Server Side)**  
Android and iOS devices aren’t used as server. They’re clients that call our web API over HTTPS. “Mobile” affects how we plan capacity (more traffic, smaller payloads) and security (how we handle tokens and CORS), but it doesn’t change the server operating system.

**Evaluation — Client Side**

**Mac (Client Side)**  
On macOS, we should deliver a modern, responsive web app that runs well in Safari, Chrome, and Firefox. Follow web standards (HTML5, CSS, modern JS/TypeScript), make it accessible (good contrast, keyboard focus, ARIA labels), and make sure it looks sharp on high-DPI screens. Test on Safari because it shares a lot with iOS’s browser engine. Most of the cost here is front-end development and cross-browser testing.

**Linux (Client Side)**  
Linux users mainly use Chrome or Firefox. A standards-based responsive web app will work right away. We should test popular Linux browser versions and check UI scaling and fonts across different desktop environments. Costs stay low because we reuse the same web codebase; just remember accessibility basics like keyboard navigation and reduced-motion settings.

**Windows (Client Side)**  
On Windows, support Edge and Chrome (and optionally Firefox). Pay attention to high-contrast modes, font rendering, and special keyboard input (IME). The same responsive web app runs here without changes. Time goes into testing across browsers and making sure performance is good on lower-end machines, but we don’t need a separate native desktop app.

**Mobile Devices (Client Side)**  
On iOS and Android, the same responsive web app should run inside Safari or Chrome/Android WebView. Use touch-friendly controls, flexible layouts, and keep downloads small (code splitting, compressed images). If we later want app-store presence, we can wrap the web app with Capacitor/Cordova instead of rebuilding it. The main cost is testing on different devices and networks—not maintaining multiple codebases.

**Evaluation — Development Tools**

**Mac (Development Tools)**  
Front-end: Node.js, npm/pnpm, React (or Vue/Angular), TypeScript, Vite/Webpack, Jest/Playwright, ESLint/Prettier.  
Back-end (for local dev): Java 17+, Maven/Gradle, Dropwizard or Spring Boot, Docker Desktop.  
Most tools are open-source and free. macOS is an excellent developer machine, but we’ll still deploy the production server to Linux or Windows Server.

**Linux (Development Tools)**  
Same stack as on macOS. Linux adds strong native support for Docker and Kubernetes, which matches production closely. IDEs like IntelliJ IDEA Community and VS Code are free or low cost. Because dev and prod are similar, there are fewer “works on my machine” surprises.

**Windows (Development Tools)**  
Windows supports the same Node/Java tools and also has great .NET options. Using WSL2 gives a Linux-like shell, and Docker Desktop handles containers. IDEs like VS Code and IntelliJ work fine. Tool licenses are minimal; Windows Server licenses only matter if we deploy servers on Windows.

**Mobile Devices (Development Tools)**  
Mostly browsers, emulators, and real devices for testing. If we choose to package the web app as native: Android Studio and Xcode are free, but publishing to the Apple App Store requires a $99/year developer account. With a web-first approach, most of the effort is in packaging and QA, not rebuilding the app.

**Operating Platform**

For *The Gaming Room’s* game *Draw It or Lose It*, the best operating platform is Linux. Linux is fast, reliable, and secure, which makes it great for hosting a web-based multiplayer game. It works very well with modern tools like Docker and Kubernetes, which help the app grow easily as more players join. Linux servers are also free to use, have lots of online support, and are available through all major cloud providers like AWS and Google Cloud. This makes Linux a cost-effective and flexible choice for expanding the game to many environments.

**Operating System Architectures**

Linux uses what’s called a monolithic kernel. That means the main parts of the system, like file storage, memory, and networking, are all part of one core program. This setup helps Linux run very efficiently and handle many users at once. It’s also modular, so you can load or remove system features when needed. Because Linux uses system resources in a predictable and lightweight way, it’s ideal for running both the main game server and its database. It also performs better than macOS or Windows when using containers or virtual machines.

**Storage Management**

Linux supports several dependable file systems, such as ext4, XFS, and Btrfs. For this project, ext4 is the best choice because it’s stable, fast, and easy to recover if something goes wrong. Linux also uses Logical Volume Management (LVM), which lets administrators resize storage space or add new drives without taking the server offline. For extra reliability and backups, the game’s data can be stored in cloud systems like Amazon S3 or Google Cloud Storage, which are designed for high availability and global access.

**Memory Management**

Linux has a very strong memory management system that balances the use of physical memory and virtual memory. It loads only the data it needs into memory, improving speed and reducing lag during gameplay. When memory gets low, Linux can automatically clear unused space or end non-essential tasks. This helps keep the *Draw It or Lose It* server running smoothly, even when many players are online at once. The Java-based application also works well with Linux because the operating system efficiently manages processes and memory for each game session.

**Distributed Systems and Networks**

To allow players to connect from different devices, *Draw It or Lose It* should use a distributed client-server model. The Linux server would handle game logic and data, while web, mobile, and desktop clients connect to it through the internet using HTTPS. WebSockets can be used for real-time communication so that player actions update instantly. Tools like load balancers and Kubernetes can spread the workload across multiple servers. If one server goes down, another can take over, ensuring players don’t lose progress. Caching tools like Redis can also help reduce lag and server strain.

**Security**

Security is very important for protecting user data. Linux has strong built-in tools like SELinux, user permissions, and firewall controls that make it harder for hackers to gain access. All communication between clients and the server should be protected with TLS encryption (HTTPS). Player accounts can be secured using token-based authentication, such as OAuth 2.0 or JSON Web Tokens (JWT). Sensitive data, like passwords, should be encrypted using strong algorithms such as AES. Role-based access control (RBAC) and monitoring tools like Fail2Ban can help block suspicious activity and keep users safe across all connected platforms.