

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_Toc115077317)

[**Table of Contents 2**](#_Toc115077318)

[**Document Revision History 2**](#_Toc115077319)

[**Executive Summary 3**](#_Toc115077320)

[**Requirements 3**](#_Toc115077321)

[**Design Constraints 3**](#_Toc115077322)

[**System Architecture View 3**](#_Toc115077323)

**Domain Model 4**

[**Evaluation 5**](#_Toc115077325)

**Recommendations 8**

[Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 09/21/2025 | Anthony Magdaleno | Initial document creation |

**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](#_sbfa50wo7nsh)

The Gaming Room seeks to expand its popular Android game, *Draw It or Lose It*, into a web-based version that supports multiple platforms. The goal is to broaden the game’s reach, improve accessibility, and provide a scalable framework for continued growth. To accomplish this, the software design includes support for multi-team gameplay, unique naming for games and teams, and controlled single-instance execution for integrity and performance.

## Requirements

* Support for one or more teams per game, with the ability to manage multiple teams.
* Support for multiple players per team through player identifiers.
* Enforcement of unique game and team names to prevent duplication.
* A single active instance of the game at any time to maintain performance and integrity.

## [Design Constraints](#_2et92p0)

* Admin/Game Management app required:  
  A lightweight administrative interface must exist to create games, define teams, add players, and associate them.
* Player identification/login:  
  The client must let each player identify themselves (simple login or unique join code) so the system can distinguish players.
* One game per player at a time:  
  A player cannot be active in more than one game simultaneously during timed rounds.
* Single active game instance:  
  Only one authoritative instance of a specific game may run at a time to avoid conflicting state.
* **Session size/players per game (cap)**:  
  Cap players per team/game to a course-appropriate number (e.g., 4–8 per team, 2–4 teams) to keep scope manageable.

## [System Architecture View](#_ilbxbyevv6b6)

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](#_8h2ehzxfam4o)

The class UML diagram displays the programs seven classes:

* Entity – Base class for Game, Team, and Player with fields id: long and name: String, plus getId(), getName(), and an overridden toString(). Subclasses may override toString() for more detail.
* Game – Extends Entity. Maintains a List<Team> (0..\*). Constructor calls super(id, name) for encapsulation and an added layer of abstraction. addTeam(String name) iterates the team list to enforce unique team names before creating a new Team. Overrides toString() for readable output.
* Team – Extends Entity. Maintains a List<Player> (0..\*). Constructor calls super(id, name) for encapsulation and an added layer of abstraction. addPlayer(String name) iterates the player list to enforce unique player names before creating a new Player. Overrides toString() for readable output.
* Player – Extends Entity. Constructor calls super(id, name) for encapsulation and an added layer of abstraction. Overrides toString() for readable output.
* GameService – Implements the Singleton pattern (private constructor, private static instance, getInstance() accessor). Holds games: List<Game> (0..\*). Tracks nextGameId, nextTeamId, and nextPlayerId. addGame(String name) uses iteration to ensure unique game names, creates a Game with a newly assigned ID, and stores it. getGame(long id) / getGame(String name) provide lookups. getGameCount(), getNextTeamId(), and getNextPlayerId() support management and ID generation.
* SingletonTester – Verifies that GameService.getInstance() returns the same instance and prints games using their overridden toString().
* ProgramDriver – Entry point. Obtains the singleton GameService, creates sample games via addGame, prints them, and invokes SingletonTester.

**"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.**

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | * Unix-based foundation provides strong stability, security, and networking, which is beneficial for reliable server performance. * Supports common web servers (Apache, NGINX, Node.js), making small-scale deployment straightforward. * Requirement of proprietary Apple hardware increases implementation costs and limits scalability. * Smooth integration with Apple services, but limited compatibility with non-Apple systems. | * Open source and no licensing requirement making it cost effective. * Widely adopted by webhosting services, which can simplify integration and deployment. * Highly modular, allowing custom server environments but requiring more expertise to configure deployment methods. * Command-line–driven administration, reducing hardware overhead but demanding greater technical proficiency. | * Popular user-friendly Internet Information Services included with Windows Server. * Licensing for server instances and CPU core count can lead to high costs for large deployments. * Administration GUI can be more resource demanding, but user-friendly. * Enterprise-level scalability, but this comes with a high cost due to licensing. * Integration of frameworks and services (IIS, .NET, SQL Server) make development, deployment, and management much more approachable. | In a client-server architecture, the server handles all computational logic and data storage. This is most evident in mobile deployments, where limited device resources place greater demands on the server. While mobile devices never act as hosts, a cross-platform deployment of *Draw It or Lose It* supporting thousands of players through multiple mobile clients requires a highly scalable and reliable server infrastructure |
| **Client Side** | * Costs may be slightly higher because testing platforms must be on Mac hardware. * Testing is simplified due to Apple’s control over its hardware-software ecosystem. Mac-Safari will be the dominant OS-browser combination. * While broad expertise is not necessary due to Apple’s strict ecosystem, developers must still be familiar with its unique traits. | * Overall costs remain low due to the absence of licensing fees for Linux or its testing environments. * Testing requires additional time and resources since multiple distribution-browser combinations must be tested for consistent performance. * As a family of distributions (Ubuntu, Fedora, Debian, etc.), Linux introduces more variability than other operating platforms, requiring greater expertise for development and troubleshooting. | * Costs will be similar to Linux. While there are licensing fees associated with Windows, hardware is flexible. One system could test both Linux and Windows. * As the dominant operating platform, testing time and resource requirements will be high. Windows will have the most browser options. * Development and testing the Windows client requires a relatively moderate level of expertise, primarily due to number of browser options. | The differences in development and deployment are most significant with mobile clients. Unlike desktop operating platforms, mobile clients are rarely web-based. Expanding from Android to iOS therefore requires developing a fully functional native iOS application. Because iOS development depends on proprietary Apple hardware, specialized tools, and dedicated expertise, expanding mobile support can be considerably costly. |
| **Development Tools** | * While macOS supports many modern languages, Swift along with the Xcode IDE are Apple-specific tools that provide smooth integration with an Apple ecosystem. * Costs can be relatively high because Apple hardware is required for development and testing. * A single development team would be sufficient for the project, though macOS development requires specialized familiarity with Apple tools and environment. | * Linux supports nearly every modern language. The choice of IDE and deployment framework will depend on the development language of choice (e.g., PyCharm/Django or Java > Eclipse/Spring Boot). * The flexibility of Linux allows development teams to lean towards open-source development and deployment tools, which can lower costs significantly. * A single team can manage both Linux development and deployment, though developers must be comfortable with command-line environments and open-source configuration. | * Like Linux, Windows supports nearly all modern languages. Windows also has its own dedicated languages the integrate well in a windows ecosystem (C#, .NET, SQL Server). * Windows provides a balanced middle ground between cost and capability. Its open-source flexibility enables low- to no-cost development, while investing in the full Windows ecosystem ensures maximum compatibility and user-friendly management. * Because Windows is a common development environment, many teams already have the expertise needed to build and deploy applications on the platform. | The Gaming Room’s existing mobile application already runs as a native Android app. Therefore, the focus here is on expanding to iOS, which introduces additional technical and resource requirements.   * Language: Swift * IDE and Deployment Framework: Xcode * Development requires macOS and therefore Apple hardware. * No licensing cost for development tools, as expenses are associated with Apple hardware. The development cost comes from deployment. Apple has a fee for their developer program, which allows publishing iOS apps to the App Store. * Developing and maintaining the iOS client will likely require a dedicated team. While development and deployment for each client type can be managed by a single team, this may be in addition to the team responsible for the server-side and web client. |

## 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**: The recommended operating platform for The Gaming Room is a Linux-based server environment. Linux offers strong stability, scalability, and cost efficiency through its open-source model. It supports modern web servers such as NGINX and Apache, making it well suited for hosting a multiplayer web application. The platform also integrates easily with cloud services like AWS or Google Cloud, allowing the system to scale as player demand increases. Linux’s modular design and robust security permissions make it ideal for a distributed, multi-platform game like *Draw It or Lose It*.
2. **Operating Systems Architectures**: Linux uses a modular architecture built around a monolithic kernel that manages CPU scheduling, device I/O, and memory allocation. This structure provides low-latency performance and efficient control over system resources, which is important for real-time gameplay. Above the kernel, the user-space layer supports web servers, databases, and application logic, allowing the game to run reliably while isolating processes and maintaining system stability.
3. **Storage Management**: A combination of object storage and a relational database management system is recommended. Game assets such as images and audio should be stored as objects and distributed via a Content Delivery Network for quick, scalable access. Player profiles and game-session data should reside in a relational database such as PostgreSQL or MySQL to support reliable lookups and updates. Implementing a data lifecycle policy to remove outdated session records and compress historical data will further optimize performance and manage storage costs.
4. **Memory Management**: The Linux Kernal uses paging, caching, and virtual memory to maintain performance under load. Frequently accessed data can be cached to reduce retrieval times, while garbage collection and concurrency control within the application prevent memory leaks and thread saturation. These techniques ensure that both volatile and persistent data are managed efficiently, maintaining smooth gameplay even as the number of active players grows.
5. **Distributed Systems and Networks**: To enable communication between clients on different platforms, *Draw It or Lose It* should use a distributed client-server model built on RESTful APIs. Clients exchange data with the main server over secure HTTPS connections. Redundant servers, load balancing, and containerization tools like Docker or Kubernetes can maintain high availability and simplify scaling. Asynchronous data handling and retry logic will minimize the impact of network interruptions or temporary outages.
6. **Security**: Protecting user data is critical. All communications should be encrypted using TLS, and client authentication should rely on secure token-based systems such as OAuth 2.0 or JWT. On the server, Linux’s role-based access control and strict file permissions help safeguard stored information. Database encryption, input validation, and regular patching will further protect user information and maintain trust across all connected platforms.