

# Draw It or Lose It

LaTisha Burns

# **CS 230 Project Software Design Template**

Version 3.0

## 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 3**](#_Toc115077324)

**Evaluation 4-6**

**Recommendations 7-8**

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 05/21/2025 | LaTisha Burns | Summary of essentials needed to execute project such as requirements, constraints, and recommendations. |
| 1.2 | 06/04/2025 | LaTisha Burns | Expanded Evaluation and Recommendations sections for Project Two to include detailed platform comparison and updated deployment considerations. |
| 1.3 | 06/18/2025 | LaTisha Burns | Revised Recommendations section for Project Three based on updated architecture analysis, security strategies, and integration with object-oriented design principles. |

## [Executive Summary](#_sbfa50wo7nsh)

Creative Technology Solutions (CTS) has been commissioned by The Gaming Room, to develop a web-based game that is compatible across multiple platforms, evolving from their existing mobile application, *Draw It or Lose It*, which is currently available exclusively on Android. The game’s primary objective is to support multiple teams and players in a drawing-based guessing challenge. Development will focus on clean, well-structured Java code that follows object-oriented design principles. The final product must operate seamlessly, scale easily, and support multiple users across various platforms.

## Requirements

* A game can have one or more teams participating.
* Each team will consist of multiple players.
* Game and team names must be unique for users to verify their availability.
* Only one game instance can exist in memory at a time, achieved by using unique identifiers for each game, team, or player.

## [Design Constraints](#_2et92p0)

The key constraints for this project include the need to develop a game that functions across multiple platforms and within a web-based, distributed environment. It must be scalable and responsive. To meet these needs, specific design patterns are required: the **singleton pattern** (to ensure only one instance runs in memory) and the **iterator pattern** (to safely manage object lists and avoid duplicates). Unique names and IDs must be enforced, and the solution must satisfy all client requirements for each platform.

## [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 UML diagram below shows how the ‘Draw It or Lose It’ game uses **object-oriented programming principles** to organize its classes. The Entity class is the base for Game, Team, and Player, and includes shared fields like id and name. This demonstrates **inheritance**, reducing duplication and promoting consistent access to common attributes.

Each Game contains a list of Team objects, and each Team contains a list of Player objects. This demonstrates **composition**, where complex structures are built from simpler components.

The GameService class uses the **singleton pattern** ensuring only one instance exists in memory by using a private constructor and the getInstance() method. This fulfills the requirement for a single game controller instance at a time.

Methods such as addGame(), addTeam(), and addPlayer() use the **iterator pattern** to prevent duplicate names by checking existing lists before adding new entries. These lists (games, teams, and players) are privately managed, showing **encapsulation** by restricting direct access and exposing only public methods.

The Entity class also includes an overloaded constructor alongside a default one, demonstrating **polymorphism**, by allowing objects to be created in different ways. **Abstraction** s used to hide internal implementation details, such as how teams and players are created. Users simply call addTeam() or addPlayer() without needing to know how the objects are constructed.

Overall, the UML design supports a scalable and secure application by using **inheritance, composition, encapsulation, abstraction**, and well-known design patterns.

**"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)

After working on *Draw It or Lose It*, modifications have been made to the Evaluation section to reflect updated requirement needs. This section now includes a detailed comparison of server-side capabilities, client-side considerations, and development tools across Mac, Linux, Windows, and mobile platforms. Each platform is evaluated based on its suitability for deploying the application in a distributed, cross-platform environment.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Characteristics:  Easy to use, consistent, can run multiple OSs.  Advantages:  Great for testing and small-scale hosting.  Weaknesses:  Limited server tools and hosting flexibility. | Characteristics:  Open-source, stable, and secure.  Advantages:  Ideal for web hosting, strong community support.  Weaknesses:  Can be harder to configure for beginners. | Characteristics: Widely used and well-supported.  Advantages:  Easy setup and broad compatibility.  Weaknesses: Higher exposure to security risks. | Characteristics:  Platform-independent and portable.  Advantages:  Accessible from anywhere with an internet connection.  Weaknesses:  Not suitable for hosting full applications; limited to light client use. |
| **Client Side** | Excellent for developing and testing on Apple hardware. Developers must use Safari or WebKit-based engines for full compatibility testing. Deployment is more expensive due to hardware costs. Requires optimization for high-resolution Retina displays and integration with macOS-specific browser behavior. | Requires more technical skill and is often used in enterprise or development-heavy environments. Compatibility with major browsers like Chrome and Firefox is good but display rendering can vary slightly by desktop environment (e.g., GNOME vs. KDE). Must consider font rendering, device scaling, and system-level browser settings. | Most common for end users and easy to support. Highly compatible with browsers like Chrome, Edge, and Firefox. Optimization includes testing for screen resolution variance, accessibility settings, and JavaScript execution in Windows-managed environments. | Must support iOS and Android, covering a wide range of screen sizes, OS versions, and input types. Requires responsive design techniques such as CSS media queries and viewport scaling. Touchscreen interactions and battery efficiency should also be considered. Testing on physical devices or emulators (e.g., Android Studio AVD, Xcode Simulator) is essential. |
| **Development Tools** | Uses Swift and Objective-C for native development. Xcode and Xcode Cloud are the standard IDEs. Cross-platform development may still require macOS for building iOS applications, even if using frameworks like React Native or Flutter. Licensing for Apple Developer Program ($99/year) is required for app deployment. | Supports tools like Docker, IntelliJ, Eclipse, NetBeans, and VS Code. Ideal for server-side development using Java, Python, and Node.js. Most tools are open source with no licensing fees. Often used in DevOps workflows for CI/CD and automated testing pipelines. | Compatible with Visual Studio, IntelliJ, Eclipse, and Git. Most suited for .NET development or Java. Offers access to Azure DevOps tools and GitHub integration. Some tools, like Visual Studio Professional, may require a paid license. | Java, Kotlin (Android) and Swift (iOS) are standard. IDEs include Android Studio and Xcode. Cross-platform tools like Flutter and React Native allow for one codebase, but testing must still be platform-specific. Build and deployment may require use of App Store Connect or Google Play Console accounts. |

## Recommendations

1. **Operating Platform**: While Windows has been considered for integration with the current Android version of "Draw It or Lose It," Linux (specifically Ubuntu Server) is a more suitable operating platform for production deployment. Ubuntu Server is cost-effective, stable, and scalable, with strong community and industry support. Its compatibility with containerization technologies like Docker makes it ideal for deploying applications in distributed environments.
2. **Operating Systems Architectures**: Ubuntu Server operates using a monolithic kernel architecture. This architecture provides efficient communication between system components because most functions operate in kernel space. It supports advanced features such as live patching and custom kernel modules, which can be beneficial for a scalable, always-on application. Its modularity and command-line-based configuration allow fine-grained control over system resources, enhancing performance and reliability. The use of object-oriented principles in the application’s design, including the Entity superclass and Singleton pattern in GameService, integrates well with the system-level structure of the Linux platform.
3. **Storage Management**: For the Linux-based platform, Logical Volume Manager (LVM) is recommended for dynamic and flexible disk storage management. LVM allows resizing of disk volumes while the system is running, providing adaptability as the user base grows. It supports snapshots for data recovery and integrates well with cloud storage services. Combined with a robust database like PostgreSQL, this setup will ensure high reliability and data integrity.
4. **Memory Management**: Ubuntu Server uses demand paging and virtual memory to manage RAM efficiently. The system loads memory pages only when needed, reducing overhead. Swapping and caching mechanisms improve performance, and the Out-of-Memory (OOM) killer ensures system stability under load. This is crucial for maintaining fast response times during gameplay sessions, particularly when handling multiple active users.
5. **Distributed Systems and Networks**: The game should be hosted on a distributed architecture using cloud infrastructure, such as AWS or Azure. A RESTful API over HTTPS should be used for communication between clients and servers. Cloud platforms provide built-in load balancing, autoscaling, and failover services. Container orchestration with Kubernetes can manage application components, improve availability, and simplify scaling. Service discovery tools help maintain reliable communication between microservices.
6. **Security**: Security should include HTTPS for encrypted communication, OAuth 2.0 for secure login, and SSL/TLS for data in transit. At-rest data should be encrypted using disk-level encryption. Role-based access control (RBAC) and regular patching should be enforced. Firewalls and intrusion detection systems should monitor traffic and alert for anomalies. Access logs should be maintained to audit user actions and identify threats.

**Summary:** This overall architecture aligns with The Gaming Room’s goal to expand Draw It or Lose It to multiple platforms while ensuring performance, maintainability, and security in a distributed environment. It reflects the strategic integration of software design principles, technical implementation, and operational efficiency.