

<Name-of-Software-Application>

# **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 3**](#_Toc115077324)

[**Evaluation 4**](#_Toc115077325)

[**Recommendations 5**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | Sep 20 2025 | Raynaldo Young | First Draft |
| 1.0 | Oct 5 2025 | Raynaldo Young | Second Draft |
| 1.0 | Oct 18 2025 | Raynaldo Young | Third Draft |

## [Executive Summary](#_sbfa50wo7nsh)

CTS has been asked by The Gaming Room to expand Draw It or Lose It from an Android app to a web version. The new design will support multiple teams and players, keep names unique, and make sure only one game is running at a time. To achieve this cleanly, we’ll use two proven design approaches: Singleton (one shared game manager so duplicates can’t start) and Iterator (a safe way to check lists to prevent name conflicts), giving players a smooth, reliable experience as the game grows. This approach will ensure a scalable, conflict free design that supports smooth gameplay as the app expands to a broader audience.

## Requirements

**Business Requirements**

* **Reach a wider audience:** Moving to the web makes the game available on more devices, bringing in more players beyond just Android users.
* **Keep the game smooth and reliable:** Names for games, teams, and players must stay unique so there are no mix-ups when people join or create groups.
* **Stay efficient:** Only one active game manager should exist at a time, preventing duplicates and wasted resources.

**Technical Requirements**

* **Support multiple teams and players:** The system must allow more than one team per game and several players per team without errors.
* **Prevent conflicts:** The software should check existing names before allowing new ones, keeping the database clean and organized.
* **Control active instances:** By using the singleton pattern, only one shared game service runs at once. Combined with the iterator pattern, the system can scan through lists safely to confirm whether a name is already taken.

## [Design Constraints](#_2et92p0)

**Single Active Game (Singleton Pattern):** Only one game manager can run at a time. This prevents duplicate games and keeps the system consistent, which is especially important in a shared web environment.

**Unique Names (Iterator Pattern):** Games, teams, and players must all have unique names. By using an iterator to check existing names, the app avoids conflicts and ensures smooth gameplay.

**Web Deployment Compatibility:** The system must expand from Android to the web. This means the design must connect to the existing backend while also working in browsers, without breaking current features.

## [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 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.**

The UML diagram shows how the classes in the application work together to meet the client’s needs. The **ProgramDriver** class is the starting point of the program. It calls the **GameService**, which manages the creation of all games. **GameService** is built using the Singleton pattern, which means only one instance of this service can ever exist. This is done by making its constructor private and only allowing access through the **getInstance()** method. If the instance does not exist, it is created; otherwise, the same instance is reused. This ensures that multiple versions of the game manager cannot run at the same time.

Once **GameService** is running, the driver can call the **addGame()** method. This method uses the Iterator pattern to scan the list of existing games and make sure that no game with the same name already exists. If the name is unique, a new Game object is created and added to the list. The same logic applies when a Team is added to a Game, and when a Player is added to a Team iterators are used to prevent duplicate names and keep everything organized.

The Entity class is a base class that holds shared attributes (id and name) for Game, Team, and Player. By placing these in a parent class, the design uses inheritance to reduce duplicate code. This also demonstrates abstraction, since the details of how IDs and names are managed are hidden from the user, who only interacts with higher level methods like **addTeam()** or **addPlayer()**.

Other key object-oriented principles are also clear in the diagram. Encapsulation is shown because attributes like id and name are kept private or protected and can only be accessed through methods. Polymorphism appears through constructor overloading and shared behaviors across subclasses. Together, these principles create a design that is reusable, easy to maintain, and aligned with the software requirements.

Overall, the domain model ensures that only one game manager exists at a time, names are kept unique, and relationships between games, teams, and players are managed cleanly. The Singleton and Iterator patterns support these goals, while core object-oriented principles make the system efficient and scalable.

## [Evaluation](#_2o15spng8stw)

This table summarizes how Mac, Linux, Windows, and mobile fit our web-based Draw It or Lose It plan on both the server side and the client side, including scale and basic cost notes. On the client side we will ship one responsive web app and test Safari, Chrome, and Firefox on desktop plus iOS Safari and Android Chrome on phones, with a bit more QA on Windows and a quick Safari pass on macOS. For tools and licensing, we note Xcode on Mac with an Apple developer fee, mostly free stacks on Linux, Visual Studio options on Windows, and Android Studio free with an Apple fee only if we publish on iOS; server and client choices stay independent.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Yes, macOS supports server style deployment for small to moderate traffic. Scaling to thousands is harder because data center options and server grade hardware are limited. There is no separate OS license, the main cost is buying and hosting Mac machines, and the hardware raises upfront cost and limits cloud choices. | Yes, Linux supports standard server style deployment and runs common web servers and runtimes. It is stable, secure, and fast, and it scales to thousands of players on major clouds with many tools. Most distributions are free so there are no OS licensing fees, optional paid support is available from vendors like Red Hat, and real costs are your cloud or hosting bill and any support plan you choose. | Yes, Windows Server supports standard server style deployment and you can host with IIS and run runtimes such as .NET. It scales to thousands with load balancing and extra servers and is friendly for admins with graphical tools clear logging and Active Directory integration. Licensing is paid often per core and many setups also need Client Access Licenses so total cost includes server licenses any CALs and your hosting bill. | Mobile devices are clients not servers. Phones and tablets connect to a backend in a data center or the cloud which keeps data and logic centralized and makes updates fast and scaling is done by adding backend capacity. Deployment can be self hosting on servers or virtual machines or using cloud hosting, cloud is pay as you go with no separate OS license while self hosting includes data center costs and any server OS licenses you choose. |
| **Client Side** | Run the site in Safari, also check Chrome and Firefox on macOS. Cost and time are mainly a bit more QA on a Mac. Use HTML, CSS, and JavaScript, follow web standards, and do a quick Safari pass for layout, media, and accessibility. | Run the site in Chrome and Firefox on a common Linux setup. Cost is low and time is basic QA. Use standard HTML, CSS, and JavaScript, check fonts and rendering, and keep pages responsive and accessible. | Run the site in Edge, Chrome, and Firefox on Windows 10 and 11. Plan extra QA since this is the largest desktop group. Use standard web tech, check Hi DPI scaling, keyboard and mouse use, and make sure it feels smooth on a mid range laptop. | Run the site on iOS Safari and Android Chrome. Use a responsive layout, large tap targets, and plan for weak networks. Cost is mostly phone testing and design time, test on one iPhone and one common Android phone, and offer Add to Home Screen for an app like feel without a native build. |
| **Development Tools** | Xcode handles Apple builds, Homebrew installs tools, and Git manages code. VS Code or JetBrains work well for web work. You’ll need at least one Mac for builds and testing, plus the Apple developer fee if you publish, with optional subscriptions for third-party tools. | Most teams code in VS Code or JetBrains and use Git plus a package manager like apt or dnf. Daily work includes a terminal, debugger, and test frameworks, with Docker as optional. Costs are mostly free and open source, with optional paid IDE tiers or enterprise support. | Visual Studio is common for .NET and C#, while VS Code fits web stacks. Add Git, Docker, a CI tool like GitHub Actions or Jenkins, and WSL2 for Linux-style tools. Licensing can include per-developer Visual Studio, while VS Code is free and cloud services are pay as you go. | Xcode is needed for iOS and Android Studio for Android, and React Native can give you one codebase if you choose. Plan to test on real phones plus emulators. Tools are mostly free, but iOS publishing needs the Apple developer fee and some libraries or cloud services may charge. |

## Recommendations

1. **Operating Platform**: Choose Linux for the server. It is dependable, widely supported on all major clouds, and usually has no OS license cost. It also scales smoothly when we need more capacity.
2. **Operating Systems Architectures**: Use a three-tier layout: load balancer, app servers, and database. Keep sessions stateless so any app server can handle any player. Add a small cache for common lookups, and keep settings in environment variables so the same build works in dev, test, and prod.
3. **Storage Management**: Put the image library in object storage and serve it through a CDN for speed. Store originals plus web-sized and thumbnail copies. Back up the originals and the database on a schedule and use lifecycle rules to move older copies to cheaper storage. Start with about 10 GB and review monthly.
4. **Memory Management**: Keep memory use tight. Hold the current image and next image in RAM, preload one round ahead, and release memory at the end of each round. Add a small in-memory cache for very popular files. Monitor usage and set safe limits so one busy game does not affect others.
5. **Distributed Systems and Networks**: Front everything with an HTTPS load balancer and run multiple app instances with health checks. Use short timeouts and retries between services and show a friendly fallback if something is slow. For live rounds and timing, use WebSocket or Server-Sent Events so players stay in sync.
6. **Security**: Encrypt data in transit with TLS and at rest in storage and the database. Hash passwords with a strong method like bcrypt. Follow least-privilege access, rotate keys and secrets, keep audit logs for sign-in and admin actions, validate inputs, and patch regularly.