

# 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 3**](#_Toc115077324)

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

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

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 3/19/2025 | Brooke Slampak | <Brief description of changes in this revision> |

**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)

CTS’s new client, The Gaming Room, was contracted to develop a web-based adaptation of their Android game, Draw It or Lose It. The game should support multiple teams, with each team consisting of multiple players. Each game instance, team, and player must be unique and should only occur once. A singleton creation pattern will be utilized to prevent multiple game instances, and an iterator pattern will prevent conflict between teams and team members.

## Requirements

* A game will have the ability to have one or more teams involved.
* Each team will have multiple players assigned to it.
* Game and team names must be unique to allow users to check whether a name is in use when choosing a team name.
* Only one instance of the game can exist in memory at any given time.
* Stay within client’s defined budget.
* Design the system to handle multiple concurrent game sessions without performance degradation.

## [Design Constraints](#_2et92p0)

The Gaming Room currently has an Android-based version of Draw It or Lose It, and CTS has been contracted to extend the application to the Web. Since Java is the native language for Android development, the application will be developed in Java to streamline the transition to the Web. Additionally, any existing APIs supporting the Android platform should be reviewed and, if necessary, extended to accommodate both mobile and web usage.

## [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 application is structured around a main driver class responsible for initiating game, teams, and player creation. This process is facilitated by the GameService class, which follows a singleton design pattern, ensuring only one instance of GameService exists at any given time.

## To enforce the singleton design pattern, GameService's constructor is private, preventing direct instantiation. Instead, the getInstance() method serves as the sole access point for obtaining the GameService instance. This method checks if an instance already exists and creates one only if one does not already exist. When GameService is activated, the driver class can invoke the addGame() method to create new games. To avoid duplicate game entries, addGame() utilizes an iterator pattern, ensuring no Game objects with identical names can be created. Created games are added in the games list. After the creation of a game, teams can be added using the addTeam() method. Similar to game creation, addTeam() utilizes an iterator pattern to prevent duplicate Team objects with the same name from being added. Each new Team is added to the teams list. Players are added to teams through the addPlayer() method, which also utilizes the iterator pattern to prevent duplicates. Any newly created Player object is then added to the players list. The Game, Team, and Player classes inherit from a common Entity superclass. Entity contains two protected attributes, id and name, and restricts direct instantiation by making its default constructor protected. Only overloaded constructors can be used, preventing the creation of null objects. This design implements multiple object-oriented programming principles. Inheritance and polymorphism are demonstrated through the extension of the Entity class and the overloading of constructors. Encapsulation and abstraction are used in the team creation process; Direct access to the Team constructor is restricted, but teams can still be added via the addTeam() method without exposing the underlying implementation to the user.

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | The Mac operating platform is known for being stable, secure, and portable. MacOS is Unix-based, making it a great choice for web development, especially if you're building apps that need to work with iOS or macOS. It comes with a reliable UNIX command-line terminal, strong performance, and support for essential development tools like Docker, Node.js, and Xcode. However, the downsides are that Macs tend to be expensive, aren’t ideal for adjusting resources to handle an influx of users or data, and don’t offer as many cloud-native deployment options. Macs are better suited for development and testing rather than for hosting a live application. | Linux is a good choice for hosting web-based software because it's stable, secure, and open-source. It's also very affordable, with tons of community support, and it works well with nearly all web servers, databases, and programming languages. The downside is that it has a steep learning curve for beginners, and sometimes you might run into issues with hardware drivers. Still, for hosting large, scalable applications in a production setting, Linux is usually a great choice. | Windows is a good choice for web apps using Microsoft technologies like .NET and SQL Server, with easy integration into tools like Active Directory. However, it can be expensive, has more security issues, and is less flexible than Linux for open-source web stacks, which is why Linux is often preferred for large-scale hosting. | Mobile devices aren’t really built for hosting web apps because they just don’t have the power, storage, or network strength for it. They’re handy for testing and on-the-go use, but when it comes to performance and handling server-side tasks, they're not a good choice. They’re better off as clients, not servers. |
| **Client Side** | Mac hardware and application-building tools are expensive, making supporting different client types such as web, iOS, Android etc. not ideal. Development time can also be lengthy since you’ll need to build and test for each platform. Having the ability to use cross-platform frameworks on other operating systems can save time and reduce the need for specialized skills that are required to develop apps on Mac. | Linux is a budget-friendly option for supporting multiple client types because it's free and offers plenty of free development tools. Development time depends on the project, but Linux’s flexibility, automation, and scripting capabilities can help speed things up. You'll need expertise in web technologies like JavaScript, Python, and PHP, plus experience with Linux server management and cross-platform frameworks. However, for iOS/macOS development, you’ll still need a Mac for testing and building those apps. | Supporting different client types on Windows can get expensive, with costs for licenses and tools like Visual Studio. Building with Microsoft stacks is pretty efficient, but things slow down when you need cross-platform support. It also takes experience with .NET, C#, Windows Server, and PowerShell. Luckily, tools like WSL and Docker make it easier to work with open-source tech when needed. | Supporting different mobile platforms can be costly considering the costs for developer programs and extra devices for testing. It also takes time since apps need to run smoothly on both iOS and Android, often using tools like React Native or Flutter. Developers need to know mobile languages like Swift, Kotlin, or Dart, and be familiar with app store rules and platform guidelines to make sure everything works across devices. |
| **Development Tools** | When deploying web-based software on a Mac, you’ll mostly work with languages like JavaScript, Python, Ruby, PHP, and Swift if you're building for macOS or iOS. Popular tools for development include Xcode for macOS/iOS apps, Visual Studio Code for web development, and PyCharm or IntelliJ IDEA for other languages. To manage packages, you'll often use Homebrew, and Docker is helpful for containerization. Git keeps track of your code changes, and you’ll spend a lot of time using the Terminal or Zsh to handle various tasks during development and deployment. | When deploying web-based software on Linux, you’ll likely use languages like JavaScript, Python, Java, PHP, Ruby, Go, or C/C++. Popular tools for development include Visual Studio Code, PyCharm, IntelliJ IDEA, and Sublime Text. For deployment, you'll use Docker, Git, Nginx/Apache, MySQL/PostgreSQL, and CI/CD tools like Jenkins or GitLab CI. The command-line interface, package managers, and scripting abilities on Linux are very useful for streamlining the process. | When it comes to deploying web apps on Windows, common languages include C#, .NET, ASP.NET, JavaScript, Python, and Java. Developers usually work with tools like Visual Studio, VS Code, PyCharm, or Eclipse. IIS is often used for hosting, with SQL Server handling the database side. PowerShell makes automation easier, Docker for Windows helps with packaging software so that it runs consistently across different environments, and version control and CI/CD are covered by Azure DevOps or Git. | For deploying mobile apps, Swift is used for iOS, Kotlin and Java for Android, and Dart for Flutter. Developers usually work with Xcode, Android Studio, or VS Code. To build apps for both platforms, Flutter and React Native are popular choices, with Firebase handling the backend. Tools like App Center and Fastlane make continuous integration and deployment easier. |

## 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**: Linux operating platform is the best choice. Utilizing Linux helps cut down on licensing costs. Linux is also a solid choice when it comes to security and performance. It’s the most common server platform out there, so there are a lot of tools and security software available.
2. **Operating Systems Architectures**: The backend would handle the game logic, while the frontend takes care of rendering. Since fast reactions aren’t entirely necessary here, everything can run asynchronously. Letting the client do most of the rendering helps keep server costs down and makes sure gameplay stays smooth, even if the connection isn’t perfect.
3. **Storage Management**: Cloud-native storage options are scalable and easily managed. Hardware such as hard disk drives or solid-state drives are also budget-friendly options that provide quick access to game data.
4. **Memory Management**: Linux saves memory by only keeping active data in RAM, using a system that swaps out less-used data. Both server and client memory needs are pretty low, mainly depending on the number of users and what the app or browser needs. Overall, it's a pretty lightweight setup for memory usage.
5. **Distributed Systems and Networks**: Cloud-native setups are popular because they help keep things running seamlessly by how their architecture prevents outages. The frontend and backend will communicate through RESTful APIs, making communication simple and smooth. This way, different platforms like Android, Windows, and iOS can easily connect with the backend.
6. **Security**: Role-based access should be utilized to manage who can do what, with an easy way to control user roles and permissions. Users will only have access to what they need, like managing teams, and team captains can edit or add players as needed. APIs should be encrypted and a firewall should be added as additional security.