

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 | 03/23/25 | Maurice Witherspoon | Updated Summary, Requirements, Constraints, and other information needed for this project |
| 1.2 | 04/20/25 | Maurice  Witherspoon | Recommendations |

**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 client aims to expand *Draw It or Lose It*, a team-based guessing game, from its current Android-only format to a web-based platform that supports multiple operating systems and devices. The enhanced software must support games involving one or more teams, with each team consisting of multiple players. To ensure clear identification and avoid conflicts, both game and team names must be unique, enabling users to verify the availability of a name before selection. Additionally, the system must enforce that only one instance of a game exists in memory at a time. This can be achieved using unique identifiers assigned to each game, team, and player instance, ensuring accurate tracking and efficient memory management across the platform.

## Requirements

The client requires the game to support one or more teams per session, with each team consisting of multiple players. To ensure clarity and prevent duplication, both game and team names must be unique, allowing users to verify name availability during selection. Additionally, the system must enforce a constraint where only one instance of a game can exists in memory at any given time. This can be effectively managed by assigning unique identifiers to each game, team, and player instance, enabling precise tracking and efficient resource management.

## [Design Constraints](#_2et92p0)

The game must support real-time interactions among multiple players without conflicts or delays. It should ensure smooth rendering of game visuals and maintain low-latency communication across all devices. Synchronization of the game state must be consistent and reliable across various platforms to provide a seamless experience. Additionally, the system must be scalable to accommodate a growing user base without any degradation in performance.

## [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)

This UML diagram uses seven classes. Classes Game, Team, and Player have a generalization relationship that references the same general class Entity, this example of multiple inheritances. These three classes, along with the GameService class, share a direct association and multiplicity, shown in the curly brackets that say our classes might share zero to many objects. Multiple players can be added to a team, each identified with an id and name. Multiple teams can be added to a game, each also identified with a name. The GamerService class has a list of games and also the singleton method called to service. This class also has references from the game class. We also have a class SingletonTester that is testing if a single occurrence of the game is running at a time associated with the ProgramDriver class. Within the ProgramDriver class, the main method is stored for the terms of use.

**"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)

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** | * **Characteristics:** Powerful hardware, stable operating system, good developer support. * **Advantages:** Easy to use, reliable, secure. * **Weaknesses:** Can be expensive. | Linux is the most common server operating system, widely used by major cloud providers such as AWS, Azure, and GCP. It is open-source, highly customizable, and optimized for scalability, making it ideal for hosting web applications. Linux offers best-in-class performance, strong security, and stability, with extensive support for web technologies like Apache, Nginx, MySQL, and Node.js. | Windows is a widely used server operating system, particularly in enterprise environments, offering seamless integration with Microsoft technologies. It provides a user-friendly interface, making server management easier compared to Linux. | Mobile devices offer portability and easy access to web-based applications, with advantages like native features (GPS, camera) and support for Progressive Web Apps (PWAs). However, they have limitations, including lower processing power, limited memory, and battery constraints, which can affect performance. Mobile devices also rely on less stable network connections, and security can be more challenging due to the variety of devices and operating systems. While they are ideal for lightweight applications, hosting resource-heavy software on mobile devices may not be optimal. |
| **Client Side** | Developing Mac can be expensive due to licensing fees, development tools, and the requirement for Mac hardware, such as MacBooks or Mac Minis, for testing. Development time may be longer due to Apple's strict App Store guidelines. Developers need expertise in Xcode, Swift, or Objective-C for native development, while web-based applications require knowledge of Safari optimizations, WebKit, and macOS security policies. | Developing Linux is cost-effective since it is an open-source operating system with no licensing fees, reducing overall development expenses. However, server management and custom configurations may require additional investment in skilled developers. Development time can vary depending on the complexity of the application, but Linux supports a wide range of web technologies like Apache, Nginx, MySQL, and Node.js, making it highly flexible for web-based applications. Expertise in Linux system administration, shell scripting, and security configurations is essential to ensure smooth deployment and maintenance. | Supporting multiple types of clients on Windows involves considering costs, time, and expertise. The cost includes licensing for various Windows versions and testing across different configurations, while time is required to ensure compatibility with older and newer versions and support for both 32-bit and 64-bit systems. Expertise in cross-platform development, security features, system integration, and performance optimization is crucial for a seamless experience. Additionally, addressing updates, patches, and user interface consistency across different devices is necessary to ensure long-term success. Balancing these factors is key to effective Windows client support. | Supporting multiple types of clients on mobile devices involves considering costs, time, and expertise. The cost includes developing for different platforms (iOS, Android), which may require separate codebases or cross-platform frameworks. Time is needed to ensure compatibility across various screen sizes, OS versions, and devices, as well as for extensive testing on different mobile models. Expertise in mobile development languages (Swift, Kotlin, or cross-platform tools like Flutter) is crucial, along with knowledge of performance optimization, security, and responsive design. Additionally, managing app store submission processes and maintaining updates across multiple devices adds to the complexity. |
| **Development Tools** | Xcode, Swift, React.js, and Node.js for deployment. | Python, Java, and JavaScript-based frameworks (Django, Flask, Node.js). | .NET Core, React.js, and Node.js. | Android Studio, Xcode, Visual Studio Code, Java, Kotlin, Swift, Objective-C, Flutter, React Native |

## 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 would be an ideal operating platform. It is cost-effective, highly customizable, and widely used in cloud environments, offering scalability and flexibility. By using Linux, The Gaming Room can host the web-based game on a reliable server, ensuring seamless scalability and performance.
2. **Operating Systems Architectures**: The game should use a Microservices Architecture on Linux. This means splitting the game into smaller, independent parts, like login, game logic, and chat. Each part can run on its own and be managed or updated without affecting the whole system. This makes the game run better and more reliably, especially when lots of people are playing at the same time.
3. **Storage Management**: Using a NoSQL database like MongoDB is a good choice for saving game data, player progress, and other important information. NoSQL databases can handle large amounts of data and are great at growing with the game as more people start playing. They are also flexible and fast, which helps keep the game running smoothly.
4. **Memory Management**: Linux uses virtual memory management to help the game run smoothly, even when many players are online at the same time. This system makes sure the computer uses memory efficiently. To make the game even faster, we can use tools like Memcached or Redis. These tools store data in memory that the game uses often, so the game can load things quickly without waiting, which reduces lag and improves performance.
5. **Distributed Systems and Networks**: To make the game work well on all kinds of devices, we can use APIs and WebSockets. These tools let the game send and receive data instantly, so players can see changes in real time. If there are network problems, the game won’t stop working—it can save some data on the device (called caching) and automatically reconnect when the internet comes back. This keeps the game running smoothly, even with connection issues.
6. **Security**: Keeping player information safe is very important. To do this, we will use encryption to protect data when it’s sent between the game and the server. We will also use AES encryption to safely store private data, like usernames and passwords. Players will log in using secure methods, and we will add extra protection by using multi-factor authentication—this means users will need more than just a password to log in, like a code sent to their phone. These steps will help keep player accounts and information safe from hackers.