

# **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 | 07/27/2024 | Elba Correa | Filled in the summary, requirements, constraints, described the UML model, filled in evaluation cells, and the 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 Gaming Room goal is to expand the reach of their popular Android game, "Draw It or Lose It," by developing a web-based version accessible on many platforms. This change poses a software design challenge, particularly in setting up the development environment and ensuring a flawless gaming experience.

## Requirements

*Business:*

* Develop a web-based version of the "Draw It or Lose It" game, ensuring its availability beyond the current Android app.
* The game should support multiple teams competing against each other.

*Technical:*

* Each team should be able to have multiple players.
* Game and team names must be unique to avoid confusion or conflicts.
* Only one instance of the game can exist in memory at any given time, making sure there is consistency and avoiding errors.
* Unique identifiers should be assigned to each game, team, and player to maintain data integrity and facilitate tracking.

## [Design Constraints](#_2et92p0)

1. **Connectivity Issues:** Players may experience temporary or complete loss of network connectivity. The game must be able to gracefully handle disconnects, potentially allowing players to reconnect without losing progress. Implement mechanisms to save game state and provide feedback to users about connectivity issues.
2. **Scalability:** The game needs to handle a potentially large and unpredictable number of simultaneous players and game sessions. Design a game that can handle the data being handled all at the same time.
3. **Platform and Browser Compatibility:** The game should run smoothly on various web browsers (Chrome, Firefox, Safari, Edge) and different devices (desktops, laptops, tablets, phones). Thoroughly test the game on different platforms and browsers to ensure compatibility. Use responsive design principles to adapt the game's interface to different screen sizes.
4. Security: Protecting user data and preventing cheating or hacking is crucial for the game's integrity. Implement security measures, including data encryption, user authentication, input validation, and server-side checks to prevent unauthorized access and malicious activities.
5. **Real-Time Interaction:** The game relies on real-time communication between players and the server for drawing updates, guesses, and scoring. Choose a real-time communication protocol to ensure low latency and reliable message delivery. Implement mechanisms to synchronize game state across all connected players.

## [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 following classes are present:

1. **Entity:** This is the base class for all entities in the game (e.g., Player, Team, Game). It provides basic attributes like id and name and common methods for accessing and manipulating those attributes (getId(), getName(), toString()).
2. **ProgramDriver:** This is the main entry point of the application. It has a main() method that starts the program.
3. **SingletonTester:** This class is used for testing the GameService singleton instance (a design pattern ensuring only one instance of the service exists). It has a testSingleton() method for this purpose.
4. **GameService:** This class is a singleton, meaning only one instance exists throughout the application. It manages the creation and retrieval of Game objects (addGame(), getGame()), and provides unique identifiers for Game, Player, and Team entities (getNextGameId(), getNextPlayerId(), getNextTeamId()).
5. **Game:** This class represents a game instance. It holds a list of Team objects (teams) and provides methods to add teams (addTeam()) and retrieve information about the game (toString()).
6. **Team:** This class represents a team within a game. It holds a list of Player objects (players) and provides methods to add players (addPlayer()) and retrieve information about the team (toString()).
7. **Player:** This class represents a player in a game. It has a name attribute and a toString() method.

**Class Interactions:**

* ProgramDriver uses the GameService to create and manage game instances.
* GameService uses the Entity class to generate unique IDs.
* GameService interacts with Game to create and manage games, and to generate IDs.
* Game interacts with Team to add teams to the game and generate IDs.
* Team interacts with Player to add players to the team and generate IDs.

**Object-Oriented Programming (OOP) Principles:**

* **Encapsulation:** Each class encapsulates its data and behavior, promoting modularity and code reusability.
* **Inheritance:** Game, Team, and Player inherit from Entity, allowing them to share common attributes and methods, reducing code duplication.
* **Singleton Pattern:** GameService is a singleton, ensuring there's only one instance of the game service, which is essential for managing game data consistently.

**"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.

| Development Requirements | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Its stability, performance, and ease of use are definite advantages. However, the higher cost of macOS server hardware and the limited hardware options compared to Linux might be drawbacks for larger-scale applications that require extensive scalability or customization. macOS is a viable choice for hosting web-based applications, particularly smaller-scale ones or those where user-friendliness and stability are very important. | Its versatility, stemming from numerous distributions, and robust community support, facilitate extensive customization and optimization for specific needs. Impressive performance and security, making it a reliable platform for handling high traffic and safeguarding data. While it may present a steeper learning curve for newcomers and require careful navigation of its diverse ecosystem, the benefits of cost-efficiency, scalability, and community-driven support often outweigh these challenges. | A user-friendly graphical interface and extensive software compatibility, making it accessible for administrators and developers familiar with the Windows environment.  However, it can be expensive due to licensing costs and may require more powerful hardware to achieve optimal performance. Windows has historically been more susceptible to security threats, necessitating robust security measures. | Their portability and constant connectivity offer convenience, but limited resources, diverse operating systems, and battery constraints pose significant hurdles. performance limitations and scalability challenges make it unsuitable for demanding applications like real-time multiplayer games. Security concerns and network reliability further underscore the need for more robust hosting solutions for large-scale, high-traffic web applications. |
| **Client Side** | While the Xcode IDE and Unix-based stability are advantageous, potential costs for additional tools and hardware, along with the learning curve for new developers, can pose challenges. | While a cost-effective and powerful option for development, requires careful consideration due to its unique landscape. The availability of free tools and the flexibility of open-source software are significant advantages, but the learning curve for newcomers and the potential for setup and configuration challenges can impact project timelines. Thorough testing across various distributions is vital to ensure compatibility, and developers should possess a good understanding of Linux fundamentals, specific distribution intricacies, and relevant open-source technologies. | Cost-wise, while development tools like Visual Studio are often free for individuals, enterprise licenses and specialized tools can be expensive. Time-wise, familiarity with the .NET framework or other Windows-specific technologies is often required, potentially adding to development timelines for teams new to the platform. Expertise-wise, developers need proficiency in languages like C#, C++, or VB.NET, along with an understanding of Windows APIs and deployment processes. | Cost considerations involve investing in various development tools and frameworks for different platforms like iOS and Android. Time is a factor due to the need to test across multiple devices and screen sizes, requiring thorough quality assurance to ensure a seamless user experience. Expertise-wise, developers need proficiency in platform-specific languages like Swift for iOS and Kotlin or Java for Android, along with knowledge of mobile-specific design principles and user interface guidelines. Additionally, optimizing for different hardware capabilities and addressing potential fragmentation issues across various device models demand specific skills and experience, which can affect project timelines and budget. |
| **Development Tools** | To build the "Draw It or Lose It" game for Mac, we will be using a few tools. For the front-end, which is what players interact with, there is JavaScript or TypeScript along with popular frameworks like React, Angular, or Vue.js. Think of these as the building blocks for the game's look and feel. For the behind-the-scenes work, like handling the game's logic and data, there is Node.js with Express.js or Python with Django. We can use Apple's Xcode as your main development tool. It's like a toolbox for building software on Mac, and it comes with everything you need to write, test, and package your game. Additionally, tools like Homebrew (a way to easily install other software) and Git (for keeping track of changes and collaborating with others) will come in handy. | To build the online version of "Draw It or Lose It" on a computer using Linux, we will need a few different tools. The main ones are JavaScript or TypeScript for the front end of the game, which is what players will see and interact with. Python or Java will be used for the backend, which handles the game's logic and data. We will also use tools like Visual Studio Code, Eclipse, or IntelliJ IDEA to write and organize the code. Node.js can help with the behind-the-scenes work, while Git is used for tracking changes and collaborating with other developers on the project. Additionally, there are special tools like Docker, Jenkins or Travis CI, Nginx or Apache, PostgreSQL, MySQL, or MongoDB to help with deploying the game and storing game data. | To build the online version of "Draw It or Lose It" on a Windows computer, We will need a few different tools. For creating the game's look and feel, you'll use C#, C++, or a simpler language like Visual Basic. To help with the game's code, we can use Visual Studio or Visual Studio Code. These are programs that help developers write, test, and organize their code. Additionally, the .NET Framework and Windows SDK will provide the necessary tools and libraries for building the game specifically for Windows. We can use a handy package manager called NuGet to easily add extra pieces of code to the project. | To build the mobile "Draw It or Lose It" app, we will use special coding languages designed for phones and tablets. For Android devices, there is Java or Kotlin, and for Apple devices like iPhones and iPads, there is Swift. Think of these languages like the building blocks of the app. We will use tools like Android Studio for Android and Xcode for Apple to put the code together and create the app. If we want to make the game work on both types of devices, we can use React Native or Flutter, which are tools that let us build one app that works on both. |

**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: I recommend using **Linux**  to host Draw It or Lose It. This offers excellent scalability, flexibility, and cost-effectiveness for web-based applications with potentially high traffic.
2. **Operating Systems Architectures**: The Linux kernel is stable and secure, providing a solid foundation for the operating system. Kubernetes enables efficient management of containerized applications, allowing easy scaling and fault tolerance. The game can be broken down into smaller, independent services, making development, deployment, and maintenance more manageable.
3. **Storage Management**: Using cloud-based object storage (e.g., Amazon S3, Google Cloud Storage) offers scalability, durability, and accessibility from anywhere. A relational database (e.g., PostgreSQL, MySQL) or a NoSQL database (e.g., MongoDB) can store game data, user information, and drawings.
4. **Memory Management**: The Linux kernel efficiently manages memory allocation and deallocation for processes, ensuring optimal performance. Kubernetes provides fine-grained control over resource allocation (CPU, memory) for each containerized service. Implementing caching mechanisms can help reduce the load on the database and improve response times.
5. **Distributed Systems and Networks**: Kubernetes handles networking between pods (containers) and services, abstracting away the underlying network infrastructure. A load balancer distributes incoming traffic across multiple servers, ensuring high availability and responsiveness. An API gateway provides a single entry point for clients to access various microservices, simplifying communication and enhancing security. Use WebSockets for real-time communication between players and the server, enabling immediate updates and a seamless gaming experience. Design the system with loose coupling between components, allowing for independent updates and fault isolation.
6. **Security**: Encrypt all communication between clients and the server using HTTPS to protect user data in transit. Implement robust authentication (e.g., OAuth, JWT) to verify user identities and authorization to control access to specific resources. Validate all user input on both the client and server sides to prevent injection attacks and other vulnerabilities. Employ firewalls to restrict unauthorized access and intrusion detection systems to monitor for suspicious activity. Keep the operating system, software libraries, and dependencies up to date to address security vulnerabilities. Encrypt sensitive data at rest (e.g., passwords, personal information) and implement access controls to limit exposure. Conduct regular security audits to identify and remediate any weaknesses in the system.