

Draw It or Lose It

# **CS 230 Project Software Design**

Version 1.1

## Table of Contents

[**CS 230 Project Software Design** 1](#_Toc182641098)

[Table of Contents 2](#_Toc182641099)

[Document Revision History 2](#_Toc182641100)

[Executive Summary 3](#_Toc182641101)

[Requirements 3](#_Toc182641102)

[Design Constraints 3](#_Toc182641103)

[System Architecture View 3](#_Toc182641104)

[Domain Model 3](#_Toc182641105)

[Evaluation 4](#_Toc182641106)

[Recommendations 5](#_Toc182641107)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 11/15/24 | Divisri | Initial version of the software design document. |
| 1.1 | 11/26/24 | Divisri | Updated the evaluation section to include detailed platform analysis |
| 1.2 | 12/11/24 | Divisri | Expanded recommendations section with detailed platform, storage, memory management, and security strategies. |

## [Executive Summary](#_sbfa50wo7nsh)

*Draw It or Lose It* is an engaging game similar to the 1980s *Win, Lose or Draw*. The Gaming Room, the client, wants to make this available on the web as well as as an Android app. This new version will make it possible to play smoothly on multiple platforms while still meeting tight software requirements, such as having unique names for each game and team, only one copy of the game in memory, and the ability to have multiple teams with different players. To make sure the design works well, we plan to use object-oriented computing, design patterns like Singleton, and scalable distributed systems. Along with cross-platform support and security, a safe and fun user experience will also be a top priority.

## Requirements

***Business Requirements:***

*Make it possible for the game to work on multiple devices, such as desktops and phones.*

*Let there be more than one team, each with its own name and several players.*

*Make sure that only one version of the game is running at all times.*

***Technical Requirements:***

*Games, teams, and players all have their own unique IDs.*

*Syncing to handle calls from multiple users at the same time.*

*A distributed design that helps with reliability and scalability.*

## [Design Constraints](#_2et92p0)

Concurrency Management: The system has to handle multiple users at the same time while making sure each user has a unique name and keeping data from getting corrupted. It is very important to set up server-side validations and locking methods.

Scalability: The app must be able to handle changing numbers of users without slowing down. We will need a backend that can grow as needed and has load sharing and caching built in.

Support for Multiple Platforms: To make sure that your website works on all browsers and devices, you need to follow the rules of flexible web design and test it thoroughly on multiple platforms.

delay: The design should keep delay as low as possible, especially when drawing and making guesses in a short amount of time.

State Management: The Singleton pattern will be used to make sure that there is only one instance of the game in memory and that state transitions are handled quickly.

## [System Architecture View](#_ilbxbyevv6b6)

## [Domain Model](#_8h2ehzxfam4o)

The provided UML class model shows the main connections between the game's main entities:

1. **Entity (Base Class)**:
   * **Attributes**: id, name.
   * **Methods**: ID generation, name validation.
2. **Game**:
   * Represents the overall game.
   * Relationships: Contains multiple teams (one-to-many).
   * Behaviors: Start game, manage rounds, and enforce unique instance rules using the Singleton pattern.
3. **Team**:
   * Represents a team of players.
   * Relationships: Composed of Player instances (one-to-many) and linked to Game.
   * Behaviors: Add players, validate team name.
4. **Player**:
   * Represents an individual player.
   * Relationships: Belongs to a single Team.

**Object-Oriented Principles**:

* **Encapsulation**: Each class encapsulates its attributes and behaviors.
* **Inheritance**: The Entity class promotes code reuse by serving as a base for Game, Team, and Player.
* **Polymorphism**: Certain types can have shared behaviors, like validation, changed to fit their needs.
* **Singleton Pattern**: Used to keep one version of the game in memory.

**"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)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Large-scale web server hosting rarely uses macOS. Its web application scalability is lower than Linux or Windows. Small, high-performance applications may work well on a macOS server, but thousands of concurrent gamers may not. | Linux is best for scalable web apps. Its open-source nature and distributed system support make it ideal for thousands of participants. Hosting on Linux is secure, scalable, and reliable. | Windows Server supports enterprise apps but is resource-intensive. For larger deployments, Windows Server licensing is more expensive. Medium-to-large organizations can use it, although scaling may demand more resources. | Not for server-side hosting. Mobile devices are client-side and cannot host a web server. We need a more powerful server-based backend architecture. |
| **Licensing Costs (Server)** | macOS server licenses cost more and are less flexible than open-source ones. License fees may rise with growth, thus the customer must consider these. | Linux is a cost-effective server OS for large-scale hosting because it is free. Ubuntu, CentOS, and others are web hosting-compatible. | Server versions, user counts, and scaling needs can make Windows Server licenses expensive. In production, licensing prices should be considered, especially for scalability. | Mobile devices are client-side platforms, hence licensing costs are irrelevant. Mobile app development may incur distribution (App Store, Google Play) or framework expenditures. |
| **Client Side** | macOS has great web browser support and a great desktop UX. However, macOS software development (especially for iOS) may require Apple framework expertise, increasing development expenses. | Linux supports current browsers but requires platform testing and optimization knowledge. While compatible with most major web browsers (Chrome, Firefox, etc.), it requires further tuning for online apps. | Windows supports most recent web browsers and is affordable for web application development. HTML, CSS, JavaScript, and current web frameworks operate well with the platform. Windows keeps web apps running smoothly across browsers. | This project needs Android and iOS mobile platforms. Safari, Chrome, and other mobile browsers must support the app. App development for mobile customers must be web-based (responsive HTML5), native, or hybrid. |
| **Compatibility Considerations (Client)** | Web browser and mobile device cross-platform compatibility is crucial for macOS development. MacOS works great for native programs, but creating a browser-responsive web app takes more work. | Linux developers need technical competence to make web apps responsive across desktop and mobile browsers. Linux supports core web technologies, although testing on multiple devices is necessary for complete compatibility. | Windows is popular for enterprise web application development due to its browser interoperability. Its IDEs and frameworks simplify cross-platform web app development. Browser compatibility will be easier than on other platforms. | The mobile app must be responsive and tested in Android and iOS browsers. Mobile app developers must support native and hybrid technologies to provide a consistent user experience across devices. Mobile browsers must support HTML5, CSS3, and current JavaScript. |
| **Development Tools** | **Tools**: Xcode (iOS/macOS), Swift, HTML5, JavaScript (React/Angular). macOS development tools are robust but might raise development cost and time, especially for iOS native apps. | **Tools**: Eclipse, JavaScript, Python, PHP, Node.js. Linux has many free and open-source web and mobile development tools. However, setup and deployment may require more technical expertise. | **Tools**: Visual Studio, .NET, JavaScript. The Visual Studio IDE offers strong tools for developing Windows-based applications. Though resource-intensive, it supports web apps and cross-platform development tools. | **Tools**: Android Studio (for Android), Xcode (for iOS), Flutter, React Native (cross-platform). Developing a native or hybrid mobile app requires different tools. These features help Android and iOS apps work smoothly. |
| **Technical Requirements Impact** | MacOS software development requires expertise and higher costs due to proprietary technology (Xcode, Swift). Cross-platform macOS, web browser, and mobile device compatibility may require more development. | Linux is cheap and open-source, but it needs an experienced development team to improve speed and platform compatibility. Linux app installations require more technical knowledge. | Windows is popular and supports desktop and mobile app development. Development tools, support, and platform-savvy developers are easier to find. Development costs may rise due to licensing. | Mobile platforms need cross-platform frameworks like React Native or Flutter or native apps. This takes more knowledge and increases development time and cost. |

## Recommendations

1. **Operating Platform**: Linux is the recommended server-side OS for "Draw It or Lose It". The scalability, stability, and distributed performance of Linux are well known. It is cheaper than Windows or macOS for server-side operating systems. To handle huge numbers of concurrent users, such as in online gaming, Linux's open-source nature allows customization and optimization. Cloud providers like Amazon Web providers (AWS) and Google Cloud Platform (GCP), which offer sophisticated game hosting infrastructure, favor it since it supports many recent web technologies. Linux is appropriate for this project since it supports distributed architectures and scales as the user base grows. With more players, distributing the game's workload across numerous machines will prevent single points of failure and assure continuous availability.
2. **Operating Systems Architectures**: The Linux architecture recommended for "Draw It or Lose It" is microservices. This method separates game functionality into different services including user authentication, game state management, and real-time communication. Docker will isolate each service in its own container. This isolation makes component deployment, scaling, and maintenance straightforward without affecting others. Based on active users, the game logic service can scale independently of the user management service. Microservices offer quick development and deployment since separate teams can work on individual services without disturbing the system. Microservices also allow the system to run if one service fails.
3. **Storage Management**: To store user profiles, game logs, and player progress, the game needs a powerful and scalable cloud storage solution. Images, game assets, and logs are best stored in Amazon S3 or Google Cloud Storage. Automatic scaling and redundancy ensure data availability even if servers fail on these platforms. NoSQL databases like MongoDB or Cassandra can scale to accommodate the game's massive data volume and store structured data like player scores and game statistics. MySQL can manage player accounts and game progress for transactional consistency across numerous game sessions.
4. **Memory Management**: Game state management will use the Singleton design to save memory. Only one instance of the game exists at a time, eliminating disputes and guaranteeing all players are playing the same version. By eliminating superfluous game instances, the Singleton paradigm improves memory management. In Java and Python, the garbage collector automatically frees memory used by objects that are no longer in use, decreasing memory leaks.
5. **Distributed Systems and Networks**: WebSocket connections will allow cross-platform real-time synchronization. For fast-paced games like "Draw It or Lose It," WebSockets enable full-duplex data transfer. This real-time communication will notify all participants of game events including new rounds and player activities. A material Delivery Network (CDN) like AWS CloudFront will send static material (images, music) to gamers worldwide faster and with less delay. We'll use RabbitMQ for background operations like game invitations, alerts, and player matching. This allows asynchronous task execution, keeping the game server responsive during peak demand.
6. **Security**: When handling user data and communicating across platforms, security is crucial. HTTPS should protect all game client-server interactions to prevent data eavesdropping. OAuth2 token-based authentication will secure user logins and access privileges across platforms. This prevents illegal access and secures user experience. Encrypting passwords and personal data in transit and at rest with SSL/TLS and AES protects user data. Role-based access control (RBAC) will restrict access to sensitive game data and backend services to authorized users.