

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.1

## 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 4**](#_Toc115077323)

[**Domain Model 4**](#_Toc115077324)

[**Evaluation 6**](#_Toc115077325)

[**Recommendations 7**](#_Toc115077326)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.2 | 06/19/25 | Danielle Alvis | Completed Recommendations |
| 1.1 | 06/07/25 | Danielle Alvis | Completed Evaluation |
| 1.0 | 05/25/25 | Danielle Alvis | Initial draft of software design document. |

**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 has requested a software solution to support the development of a new game called Draw It or Lose It. The game involves teams competing to guess puzzles based on gradually revealed images, with four rounds of play lasting one minute each. The challenge is to design a system that can organize games, teams, and players while ensuring names are unique and only one version of the game system runs at a time.

The problem we're solving is how to structure the software in a way that keeps everything organized, easy to manage, and ready for future development into a web-based version. The proposed solution includes a design that allows multiple teams and players within each game, ensures game and team names are unique, and uses a single instance of the system to keep everything in sync.

To move forward, the client should understand that this design is focused on setting up the foundation. It creates a basic structure for the game logic without including things like saving progress or running in a browser yet. This makes it easier to build, test, and expand the application over time. Once this version is complete, it can be used as the starting point for building out the full web-based version of the game.

## Requirements

The client, The Gaming Room, has outlined several key requirements for the Draw It or Lose It game application. These include both business goals and technical needs to support future development into a web-based version.

**Business Requirements:**

* The game must support multiple teams competing in each game.
* Each team must be able to have multiple players.
* Team and game names must be unique to avoid confusion during gameplay.
* The game must follow a consistent structure: four rounds, each lasting one minute, with drawings revealed over time.

**Technical Requirements:**

* Only one version (or instance) of the game should run in memory at any time. This will help avoid bugs and keep the system organized.
* Each game, team, and player needs a unique identifier to keep track of them.
* The structure should be easy to expand later into a full web-based version of the game.
* The design should make it simple to check whether a game or team name is already being used.

These requirements form the foundation of how the application will be designed and ensure that it’s prepared for both current needs and future updates.

## [Design Constraints](#_2et92p0)

Since *Draw It or Lose It* is being developed as a web-based game, there are a few important limitations and considerations that will shape how the application is built.

* **One Game Instance at a Time:** The system must only allow one active version of the game to run in memory at once. This helps avoid confusion, prevents accidental overlap between games, and ensures the application runs smoothly. This means special care must be taken in the design to manage memory use and how users access the game.
* **Unique Identifiers Required:** Every game, team, and player needs a unique name or ID. This is important to keep everything organized and ensure teams don’t accidentally choose the same name. It also supports future development where users might join games online from different locations.
* **Web-Based Access:** Because this game will eventually run in a browser, the design must work across different devices and internet connections. It also means certain things—like saving progress, sharing data between users, and real-time updates—need to be handled carefully to avoid lag or data loss.
* **Security and Fair Play:** To make sure everyone gets a fair experience, the application needs to prevent cheating or glitches. That includes making sure only one team can guess at a time when rules allow it, and ensuring the drawings display correctly and on time for all users.
* **Scalability:** The design should support more players and teams over time, meaning it should be easy to grow or upgrade without having to rebuild everything from scratch.

These constraints help guide the design so that the final product works well for users, avoids bugs, and can grow in the future.

## [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 diagram shows the structure of the software and how different parts of the game system are connected. It’s a visual map of how the code is organized to support the *Draw It or Lose It* game.

**Key Components:**

1. **Entity Class**
   * This is the base class shared by all major elements in the game: Game, Team, and Player.
   * It provides each object with a unique ID and name.
   * This design reduces repetition in code and keeps things consistent.
2. **GameService Class**
   * This class manages everything happening in the game system.
   * It keeps track of all games and is responsible for creating new ones, assigning unique IDs, and helping manage teams and players.
   * It’s built using a singleton pattern, which means only one instance of it exists during a session. This matches the client’s requirement that only one game should run at a time.
3. **Game Class**
   * A game contains a list of teams and controls how teams are added and managed.
   * Each game is assigned a unique ID and name.
4. **Team Class**
   * A team belongs to a specific game and contains multiple players.
   * It also has its own unique ID and name.
5. **Player Class**
   * Each player belongs to a specific team.
   * Players also have their own unique ID and name.
6. **ProgramDriver and SingletonTester Classes**
   * These are used to run and test the application.
   * ProgramDriver starts the game.
   * SingletonTester ensures that the singleton setup is working correctly.

**How the Classes Work Together**

* GameService creates and manages games.
* Each Game can have multiple Teams.
* Each Team can have multiple Players.
* The shared Entity class ensures that every game, team, and player has a consistent way to handle names and IDs.

**Object-Oriented Programming Principles Used**

1. **Inheritance**
   * Game, Team, and Player inherit from Entity. This avoids duplicate code and makes updates easier across all types of objects.
2. **Encapsulation**
   * Each class keeps its own data and behavior. For example, teams manage their own list of players without exposing the internal list directly.
3. **Abstraction**
   * The system hides complex internal logic (like ID assignment and game management) behind simple methods like addGame or addTeam.
4. **Singleton Pattern**
   * Only one GameService object can be created, ensuring that only one version of the game runs at a time. This directly supports the client's requirement.

**"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** | macOS can be used for hosting, but it's not typically recommended. It has more limitations when it comes to scalability and is not commonly used in production environments. It's more expensive due to hardware and software costs. | Linux is widely used for web hosting. It’s open-source, secure, stable, and free to use, which helps keep costs down. It also works well with cloud services and is highly customizable. | Windows Server supports web hosting and is easy to use, especially with .NET applications. However, it usually comes with licensing fees, which makes it more expensive than Linux. | Mobile devices are not used for hosting. They are the clients, not the servers, so this category doesn’t apply here. |
| **Client Side** | Mac users expect high performance and a polished user experience. Testing on Safari is important. Development can be costly due to Apple hardware, but it supports standard browsers. | Linux users are less common, but the platform is flexible. Chrome and Firefox are popular here. It's cost-effective, but users may expect more technical features. | Windows has the largest desktop user base, so it’s essential to test for compatibility with Edge and Chrome. Development and testing are generally low-cost. | Android and iOS both need careful testing due to different screen sizes, input styles, and browser behavior. iOS may require more time because of stricter testing and approval processes. |
| **Development Tools** | macOS supports tools like Xcode, Visual Studio Code, and Terminal. It's especially useful for developing iOS-compatible apps. Most web development tools run well on Mac. | Linux supports many powerful open-source tools like VS Code, Git, and a variety of server and scripting languages. It's great for backend development and command-line workflows. | Windows works with tools like Visual Studio, VS Code, and .NET. It’s familiar to many developers and supports both front-end and back-end development well. | Mobile development often uses emulators and real device testing. Android Studio and Xcode are common tools. Web development for mobile uses responsive frameworks like Bootstrap or 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**: It is recommended that a Linux-based server be used as the main system for hosting Draw It or Lose It. Linux is known for being reliable, secure, and highly flexible, which makes it a strong choice for supporting a growing number of users. It is also widely supported by cloud service providers, making it easy to scale up when needed. Unlike some other platforms, Linux does not require expensive licenses, which helps reduce costs for The Gaming Room over time.
2. **Operating Systems Architectures**: The Linux system will use a modern design that allows it to handle many users at once while keeping things running efficiently. Common web server setups, such as those that include Linux, a web server (like Apache or Nginx), a database, and application code, are well-suited for this kind of project. This setup makes it easier to update the system, fix issues quickly, and add new features as the game grows. Since Linux is customizable, it can be adjusted to meet the game’s specific needs without adding extra programs or tools that are not necessary.
3. **Storage Management**: It is recommended that the game’s images and user data be stored using cloud-based services that are designed to handle large amounts of information. These services are reliable, flexible, and secure, making them a good fit for storing the game’s large library of high-quality images. Files that are needed temporarily—such as logs or temporary copies of data—can be saved directly on the server. Backup systems and safety measures should also be in place to prevent the loss of any important information. Because each image is about 8MB and there are 200 of them, special tools can help reduce loading times without lowering quality.
4. **Memory Management**: The Linux system is designed to use memory efficiently, which helps keep the game running smoothly even when many users are playing at once. To speed things up even more, it is recommended that the system use tools that temporarily store frequently used data in memory. This way, things like game images and active game rounds can load faster. This is especially helpful since drawings in the game are revealed over time and need to appear on the screen quickly for all players.
5. **Distributed Systems and Networks**: Because the game needs to work across different devices and allow many users to play at the same time, it is recommended that the system be built in separate parts that each do a specific job. These parts can run independently and grow as needed. For example, one part can handle team setup, another can deliver the drawings, and another can keep score. These parts can talk to each other through a secure connection to keep everything in sync. Extra tools like load balancers and file delivery networks can help prevent slowdowns, especially for mobile users who may have weaker connections.
6. **Security**: Protecting user information and keeping the system secure is essential. All communication between players and the game should be protected using secure internet protocols. Users should log in through secure systems that make sure only they can access their information. The Linux server should also be locked down so only approved users can make changes or access sensitive parts of the system. Additional protection, like monitoring for threats and protecting against internet attacks, should be added through cloud security tools. All personal information stored by the game should be encrypted and watched carefully for any unauthorized access.