

Prepared by Joshua Shoemaker

# **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 6**

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 09/22/2024 | Joshua Shoemaker | Executive Summary – Describe the software design problem and present solutions.  Requirements – client’s listed requirements.  Design Constraints – constraints and implications. |

**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)

Draw It or Lose It is a web-based game application that requires the environment to host various games, teams, and players. Each game must be unique and consist of teams. Each team must be unique and consist of unique players. There must only be one instance of each game, team, and player stored in memory at any given time. To ensure only one instance of the environment can be created to store games, teams, and players we use a Singleton design pattern. The Iterator design pattern is implemented to validate and handle the creation of new games, teams, and players, ensuring that existing names and identifiers are unique.

## Requirements

* Unique Games – Each game session, team, and player should have a unique identifier.
* Portability – Web-based development should be open to web browsers on different platforms.
* Composition – Games should consist of teams, and teams consist of players.
* Single Instance – Only one instance of each game, team, and player should exist at one time.

## [Design Constraints](#_2et92p0)

* Unique Games & Single Instance – Only one instance of each unique game, team, and player will be available at one time. Implementing a singleton design pattern for the GameService on the server side will allow only one instance and maintain data. The environment must enforce uniqueness by validating the identifiers of each game session, team, and player. An Iterator pattern will enable the traversal of each game, team and player list to validate uniqueness.
* Portability – Different web browsers will be used across different platforms to access the web application. Optimizing performance for client-side development is necessary to ensure the game functions properly on platforms with limited processing capabilities.
* Composition – The game must be comprised of teams and the team must be comprised of players. Composition is a fundamental part of object-oriented programming and allows us to do this easily.
* Scalability – The architecture must support scaling to meet increased demand and usage as more people play Draw It or Lose It.
* Performance – Low latency is necessary to fairly give players the opportunity to guess the image and to provide a smooth interaction with the game.

## [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)

In The Gaming Room UML Diagram, the Game, Team, and Player classes extend the Entity class. This is a principle of object-oriented programming. Inheritance allows sub-classes to inherit parent-class fields and methods. This ensures that each Game, Team, and Player has unique identifiers and have the basic functions to modify those attributes. Encapsulation is used by making the fields of these classes private and only allowing defined methods to access or modify these values. Composition is expressed between the GameService, Game, Team, and Player classes. This is another form of encapsulation. This allows for modularity and is better for maintenance. The Entity class is an abstract class, another principle of object-oriented programming. The toString() method in the Entity class is abstract, meaning it has to be defined in each of the child classes. Polymorphism allows for this method overriding. The ProgramDriver class initializes the GameService instance then establishes some games, teams, and players. The ProgramDriver then creates a tester using the Singleton Tester class to test the Singleton Design pattern utilized in the GameService class. This ensures that only one instance of each game, team, and player can exist at a time. The unique identifiers or IDs for each Game, Team, and Player are held in the GameService.

**"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** | Limited hardware.  Horizontal exclusive scaling.  Less support and less common.  OS uses more resources.  Costly hardware. Stable, Secure. | Open source.  Lightweight OS.  Stable, Secure, Customizeable, Popular as Server side OS for RESTful api. High community support. Low Commercial support. Enterprise versions can be costly. Scalable | Scalable, licensing fees increase substantial for dedicated servers and enterprise versions. Extensive commercial and community support. Very compatible with server software and tools. | Not suitable to host server. |
| **Client Side** | Expertise in JavaScript and Libraries such as React or Angular. Cost for browser testers range free - $30/ month. Time for mac specifically is devoted to safari unit testing. | No Specific browser comes with Linux. Firefox is usually the default browser. The only specific resources need will be time for Firefox unit testing. | No additional costs or Expertise. Time will be increased for Edge unit testing. | Responsive Web Design Expertise will be needed for mobile browser integration. Time will be devoted to unit testing on small screens on safari and |
| **Development Tools** | Java, Maven, JavaScript, React, Dropwizard, Eclipse, and other IDE or Code editors are free and opensource. A RESTful application promotes two development teams, one for server side and one for Client side. Mac’s specific investment will primarily be in the hardware and Mac OS. | Linux can use all the same languages and tools as Mac. Linux OS is open source and free. | Windows has the same option for these development tools. | No extra development tools required for Mobile browser development. |

## 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 distributions make for a powerful, inexpensive, secure, reliable, and supported platform. Because Linux is open source there is a wide variety of community support and documentation available for free. Being open source, Linux distributions have also been rigorously tested and modified to be flexible and compatible with various technologies and client-side programs.
2. **Operating Systems Architectures**: A modular architecture offers the greatest vertical and horizontal scaling versatility. It is also the most efficient when processing information. A modular system has excellent security due to the permissions and modules. Modulars also lend to the client-server architecture of a RESTful API allowing The Gaming Room to include only necessary components. Nonetheless, a modular architecture can be complex and difficult to manage. If a smaller more manageable architecture is desired a layered system can be utilized and grow into a modular architecture as scaling resources are required.
3. **Storage Management**: Cloud storage is the most practical solution for a scaling project considering its “pay as you use” aspect. This ensures storage, and access to that storage through bandwidth is available at the rate required to produce a smooth and enjoyable user experience.
4. **Memory Management**: Linux uses virtual memory to ensure *Draw It or Lose It* runs smoothly, even with multiple game instances active. It manages memory by moving inactive data to disk when necessary, prioritizing resources for real-time gameplay. Shared memory allows common assets to be reused across game sessions, reducing overhead. Linux efficiently allocates and frees memory to prevent issues like slowdowns or crashes, ensuring stable performance for all users.
5. **Distributed Systems and Networks**: The server handling the game logic and player interactions, using a RESTful API, will manage connectivity issues. This allows players to reconnect without losing progress if the connection drops. The server will continuously send updated data ensuring the player receives synchronized data.
6. **Security**: To protect user information in *Draw It or Lose It*, strong encryption protocols, such as Transport Layer Security 1.3, will secure data during transmission between clients and the server over HTTPS using TLS. Users can make an account storing usernames and passwords utilizing Argon2id which hashes and salts passwords. OAuth2.0 can be optionally added so users can use their Google, Facebook, or other social media accounts for convenience. Access control measures will restrict user actions based on roles, utilizing the principle of least privilege. Regular updates to the Linux operating system will address security vulnerabilities, complemented by network security tools like firewalls to monitor traffic and prevent unauthorized access. Compliance with data protection regulations will ensure user privacy by minimizing data collection and giving users control over their personal information.