

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 | <01/28/24> | Jason Pullara | Initial Recommendation |
| 1.1 | <02/11/24> | Jason Pullara | Minor revisions for grammar. |
| 1.2 | <02/25/24> | Jason Pullara | Final revision for final project. |

**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 client requires a web-based version of their Android app game “Draw It or Lose It” which allows teams to compete in guessing puzzles based on stock drawings. The new version needs to be accessible across various platforms, maintain a single live instance of the game, ensure unique identifiers for game sessions, tams, and players, and support multiple teams with multiple players each.

The address this, Creative Technology Solutions (CTS) proposes a software design that includes a robust backend system capable of handling concurrent game sessions, a user-friendly interface for game interaction, and a scalable architecture to support growth. The solution will be developed in a web-based environment, leveraging modern web technologies to ensure cross-platform compatibility and real-time gameplay experience. Unique identifiers will be implemented to maintain game integrity and exclusivity of names, while a singleton pattern will ensure a single instance of the game in memory.

## Requirements

*<* Please note: While this section is not being assessed, it will support your outline of the design constraints below. *In your summary, identify each of the client’s business and technical requirements in a clear and concise manner.>*

## [Design Constraints](#_2et92p0)

**Single Instance**: Ensuring only one game instance runs at a time necessitates a singleton pattern, influencing the application’s state management and concurrency.

**Cross-Platform Compatibility**: The game must function seamlessly across different browsers and devices, which necessities a responsible design and careful selection of technologies.

**Network**: Real-time game functionality requires a persistent and stable network connection, as well as efficient implementation of networking technologies at multiple levels in the OSI model, including in code.

**Data Persistence**: A reliable database schema will need to be engineered for storing, retrieving, and validating user and team data.

**Concurrency**: Multiple teams with multiple players will require robust session management and real-time data synchronization.

## [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 UML class diagram provided outlines the structure of “Draw It or Lose It.’ The Entity class acts as a base class, ensuring unique identifiers thought a long-type id and a name String. Derived classes include Game, Team, and Player, each with specific attributes and methods relevant to their function within the application.

id and name fields are inherited across Entity, Game, Team, and Player, demonstrating the OOP principle of inheritance, which reduces redundancy and increases maintainability. Each class encapsulates its data and behavior, exposing only the methods necessary to interact with the object through setters and getters. The OOPS concept of abstraction is outlined as the Entity class provides an abstraction layer for common properties, which simplifies the representation of all entities within the game.

SingletonTester shows the intention to implement the singleton design pattern which will ensure that only one instance of the game service exists, meeting the client requirements. Aggregation and association are also present in the UML diagram across the Game, Team, and Player classes as well as ProgramDriver and GameService showing usage association.

**"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** | Advantages: Stable Unix-based environment (FreeBSD) good for running web servers and development stacks.  Weaknesses: Less common for hosting servers compared to Linux with potentially higher licensing costs. | Advantages: Most common server OS, open-source, wide community support, compatible with most server software, highly customizable, scalable, and cost-effective.  Weaknesses: A steeper learning curve for those unfamiliar with the Unix-like environment | Advantages: Integrates Microsoft technologies (.NET, IIS), user-friendly interface, excellent support for enterprise-level applications.  Weaknesses: licenses and support costs can be prohibitive, less common for web server hosting, can be resource-intense | Not typically used as a server platform, not suitable for server-side processing due to hardware, connectivity, and stability restraints. |
| **Client Side** | Advantages: Consistent user-experience, high quality graphics support, good security features.  Weaknesses: Hardware is expensive, less market share compared to Mobile and Windows. | Advantages: Customizable, open-source, good hardware support, potentially less resource-intensive.  Weaknesses: Fragmentation can lead to inconsistent experience, not as well supported as a desktop platform. | Advantages: largest computer market share behind Mobile, broad hardware support, wide range of software compatibility.  Weaknesses: Target for the most malware, difficult o develop for compared to standard Unix-like environments, costly development cycle. | Advantages: Ubiquity, touch interface, ease of access, always connected.  Weaknesses: Limited screen real estate, wide gap in performance across devices and networks, extremely divided and fragmented operating systems across devices. |
| **Development Tools** | Advantages: Native support for development with Apple products, good select of modern tools (Xcode, Atom), Unix-based terminal for server-side tools.  Weaknesses: Development for non-Apple platforms can be difficult. | Advantages: variety of open-source development tools, customizable environments, preference of many developers for server-side tools, open-source IDE, and dev tools (VS Code, GCC, etc.)  Weaknesses: Lacks integration with proprietary hardware and software. | Advantages: Visual Studio is the gold standard IDE for development, great game development support (DirectX), .NET framework support without Mono, and other Microsoft technologies.  Weaknesses: Less optimal for developing in certain languages, difficult barrier of entry, can be difficult to use tools and libraries specifically built for Unix/Linux environments. | Advantages: IDEs available to develop and test on the go with simulators/emulators for mobile app development.  Weaknesses: Not as powerful or feature-rich as desktop environments, difficult to write code natively, more suitable for small code changes or using “no-code” tool chains. |

## 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**: For server-side we recommend Linux for its compatibility, scalability, developer, and community support, as well as its frameworks and open-source libraries. On the client side we recommend Cross-Platform Web Technologies such as HTML5, CSS3, Javascript, ReactJs, Angular, etc. which are universally supported across browsers and bypass the difficulty of cross-platform development. For development tools we recommend Microsoft’s Visual Studio Code for unified support across developers’ hardware and operating systems, and Git for version control.
2. **Operating Systems Architectures**: Linux is a free and open-source operating system widely used for server-side applications. Modern systems support multiple processor architectures, including x86\_64 and ARM, as well as having excellent package manager support which helps automate the installation of new software and maintenance of system libraries and utilities. On the client side, the web-based architecture is designed to run in different web browsers making the client-side platform agnostic.
3. **Storage Management**: Multiple storage systems are appropriate for this system. A hybrid approach using a relational database system, such as PostgresSQL, for data that requires relational storage, a caching and real-time system such as Redis for session management and caching frequently accessed data, and an object storage system such as AWS S3 for larger volumes of data.
4. **Memory Management**: The Linux operating system environment has several memory management techniques we can take advantage of. This includes Virtual Memory, allowing the physical RAM to be expanded to the disk drive in the event the system RAM fills to capacity. The Page Cache, Slab Allocator, OOM Killer, and Transparent Huge Pages will allow Draw It or Lose It to grow as it needs to without the need for direct human intervention to maintain the stability of the system.
5. **Distributed Systems and Networks**: Draw It or Lose It will use a Client-Server model where the client is a web application accessible from any device with a compatible web browser which connects to a central server application running on a Linux system. We will use A RESTful API and WebSocket’s to facilitate communication between the client and server, while implementing this in a microservice architecture which will allow the game to rapidly expand. On the network side, we will utilize a Content Delivery Network to deliver static content to the worldwide play database, load balancer to distribute incoming network traffic so that no single server is overwhelmed or gate rushed, multiple redundancy with a failover system so system failures have a minimal impact on existing players, and a distributed database system that ensures there isn’t reliant on a single monolithic database which could be a single point of failure.
6. **Security**: Server side, we will implement data encryption and hashing (such as using a one-way hash to store and compare user passwords and other sensitive details that do not need to be accessed, just verified), connection encryption using TLS to secure data, implement strict access controls, place servers behind firewalls and other network security appliances to ensure that only the services we designate can be accessed from outside the network, implement robust security update protocols, using OAuth, OpenID, or SAML for secure authentication methods, using Fail2Ban as our Intrusion Detection System, and conduct regular security audits (both on the network and in code) with third party vendors to find and mitigate potential vulnerabilities. The client side will receive similar treatment using Cross-Site Scripting protection, Cross-Site Request Forgery protection, Content Security Policies, secure cookies, and regular audits from a third-party vendor to ensure that these systems are being implemented and maintained well.