

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 | 3/16/2024 | Joe Clancy | Executive Summary, Design Constraints |
| 1.1 | 3/23/2024 | Joe Clancy | Evaluation |
| 1.2 | 3/29/2024 | Joe Clancy | Recommendations |

## [Executive Summary](#_sbfa50wo7nsh)

Creative Technology Solutions (CTS) has acquired a new client, The Gaming Room, who wants to develop a web-based game that serves multiple platforms based on their current game Draw It or Lose It. Draw It or Lose It is currently only available as an Android app. The game should allow multiple unique games, each with multiple unique teams, each with multiple unique players.

## Requirements

* A game will have the ability to have one or more teams involved.
* Each team will have multiple players assigned to it.
* Game and team names must be unique to allow users to check whether a name is in use when choosing a team name.
* Only one instance of the game can exist in memory at any given time. This can be accomplished by creating unique identifiers for each instance of a game, team, or player.

## [Design Constraints](#_2et92p0)

* New instances of games, teams, and players must be unique.
* The game must operate correctly across various devices, browsers, and operating systems.
* Security is a high priority in a web-based environment. Authentication, authorization, and encryption mechanisms should be implemented to protect against cheating or data breaches.
* Bandwidth limitations may affect either the game server or clients. The game should be optimized to minimize bandwidth usage.
* Distributed systems are subject to network interruptions; the game must be resistant to faults caused by network reliability.

## [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 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.**

The ProgramDriver class contains the main method for the program and is used to initialize games, team, and players.

The GameService class follows a singleton design pattern to ensure only one instance of the class exists in memory at a time. The getInstance method can be used to create a reference to the GameService instance. The addGame method is used to create a new Game instance. If addGame is used to attempt to create a Game instance with an existing name, the existing Game instance will be returned. New Game instances are added to the games list.

The Entity superclass has two protected data members, id and name. The default constructor for Entity is protected to prevent parameterless instances of Entity from being created.

The Game class holds information about a single game. Game extends the Entity class. The addTeam method allows for the creation of a new instance of Team with a unique name. If an existing name is provided, addTeam will return the existing instance of Team. New instances of Team will be added to the teams list.

The Team class holds information about a single team. Team extends the Entity class. The addPlayer method allows for the creation of a new instance of Player with a unique name. If given an existing name, addPlayer will return the existing instance of Player with that name. New instances of Player will be added to the players list.

The Player class holds information about a single player, including player name and player ID.

This UML class diagram demonstrates object-oriented programming principles in the following ways:

**Abstraction.** Abstraction is demonstrated by hiding information about the functionality of classes. Game, Team, and Player instances are all created through methods of other objects. The actual mechanisms for instantiation are hidden.

**Polymorphism.** Several classes demonstrate polymorphism with overloaded constructors.

**Inheritance.** The Game, Team, and Player class all extend the Entity class.

**Encapsulation.** Private data members, like the teams list in Game, demonstrate encapsulation because they cannot be accessed outside of Game, and instead must be modified with mutator functions like addTeam.

## [Evaluation](#_2o15spng8stw)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | As of April 21, 2022, Apple has discontinued macOS Server.  Even if a copy could be obtained, using an unsupported server OS is inadvisable. | Linux is open-source, and many distributions are available for free, though specialized server distributions like Red Hat have licensing fees.  Because Linux is very popular for server hosting, tools and support are abundant. Linux-based hosting is also common among cloud-based hosting providers. | Windows Server is easy for many people to use, especially those without much experience with Linux.  Windows Server is well supported, and many cloud-based hosting platforms are available.  However, compared to alternatives, Windows Server licenses are very expensive, and license costs can increase significantly when supporting a large number of clients. | Mobile devices are generally not appropriate for hosting a multi-user webservice.  Mobile devices have limited hardware (memory, processing) and lack scalability to support the hardware requirements of a webservice.  Additionally, hosting tools largely do not exist for mobile devices and would need to be developed from scratch. |
| **Client Side** | Developing client software for MacOS will require at least one machine running MacOS capable of running Xcode.  Developers skilled in Swift or objective-C are ideal for MacOS projects, though other languages are supported as well to varying degrees.  MacOS uses Safari as its default web browser, and ensuring client software functions as intended on that browser is crucial. | Development on Linux presents few obstacles. Linux supports most popular languages (Java, Python, C/C++), and licenses of many Linux distributions are free.  Ensuring client software functions correctly on the Firefox browser is important.  Linux has very little market penetration as a client operating system, however, and it may be worth considering if Linux client support is worth the development cost. | Windows is the most popular client operating system and supports most languages and browsers.  Ensuring support for common browsers (Chrome, Edge, Firefox) is important, otherwise, no significant impediment to development for Windows exists.  Because Windows accounts for over 50% of all operating systems, support for Windows is the most cost-effective choice. | Because the game client software already runs on the Android platform, no additional expertise or hardware should be required to support it.  Supporting iOS will require a MacOS workstation and developers with expertise in Swift.  While there are other mobile operating systems, none has significant enough market penetration to warrant development effort. |
| **Development Tools** | MacOS uses primarily Swift and Objective-C for development, though other languages are supported. The Xcode IDE is only available on MacOS and has a cost of $99 per year if intending to upload to the app store. | Linux supports most programming languages and has many IDEs available for free.  Eclipse or IntelliJ IDEA for Java, PyCharm for Python, or VSCode for C/C++ are all popular choice. | Windows supports nearly all programming languages. Because much of the game code is already written in Java, Eclipse or IntelliJ IDEA are popular IDEs, with Eclipse being free. | The Android SDK and Android Studio IDE use Java and are available from Google for free on most workstation operating systems.  The iOS SDK uses primarily Swift and Objective-C. The Xcode IDE is only available on MacOS and has a cost of $99 per year if intending to upload to the app store. |

## Recommendations

1. **Operating Platform**: I recommend The Gaming Room use a Linux-based system for its operating platform. Linux supports nearly all common programming languages, allowing The Gaming Room flexibility in how it implements the server application.

Licenses for Linux are usually significantly less expensive than alternatives. Because Linux is often the first choice for server platforms, most cloud-based service providers will provide support for Linux-based servers, giving The Gaming Room plenty of choices for hosting services. Alternatively, if The Gaming Room prefers to use on-premises servers, Linux support for server hardware is ubiquitous.

The client-server architecture The Gaming Room will employ can serve resources to client applications on any platform from the Linux-based server application, allowing Draw it Or Lose it to be deployed on other computing environments.

1. **Operating Systems Architectures**: As previously stated, Draw it or Lose it will use a client-server architecture to accommodate multi-platform support. The backend server application will be run on a Linux-based system hosted by a cloud-based service provider or on-premises hardware managed by The Gaming Room.

The client platforms will each run a variation of the client application. Ideally, Draw it or Lose it should use a browser-based implementation using HTML5, CSS3, and JavaScript to create the client application. The server will provide game assets like images and code on demand through a RESTful API to support the client application. This implementation will reduce the need to co-develop separate client applications for each platform, though testing should still be conducted for each browser and platform Draw It or Lose It will support.

Alternatively, stand-alone client applications could be built for each platform, but this would be more costly and require a broader range of specialized developer expertise. This may be cost-effective for mobile platforms, where users expect to interface with games through apps, and an Android-based client application already exists, but I do not recommend it for other client platforms.

1. **Storage Management**: Using solid-state storage (SSD) for our server platform will allow the fastest delivery of assets. However, because it is likely that asset-delivery to client applications will be primarily network-limited, lower-cost hard spinning hard disk drives may also be used at The Gaming Room’s discretion.

For client applications, our client-server architecture and delivery of game assets through API requests should render the need for non-volatile storage trivial or remove it altogether.

1. **Memory Management**: Because the backend server application will not need to accomplish any asset rendering, its memory demands should be limited. It will require enough memory to serve game assets to clients as they are requested and manage game, team, and player information for as many games, teams, and players are active concurrently. There is no need to keep image assets loaded in memory until they are requested or after they are delivered, so overhead should be low.

The entire client game application will need to be loaded into main memory while the game is active, however, because we do not intend to maintain a stand-alone application on client devices, as well as any assets that are currently in use by the game. This should amount to a user interface and a single high-definition image at any given time, however, and should not be cumbersome.

1. **Distributed Systems and Networks**: As previously discussed, the system will be distributed using a client-server architecture to allow most of the game logic to be managed by a server application. Game states, assets, and code on demand will be served to client applications via a RESTful API over the internet using standard TCP/IP protocols. While there is no obstacle to operating Draw It or Lose It on a smaller private network, there is also no real benefit.

Unless The Gaming Room intends to construct multiple datacenters, using cloud-based server hosting will provide the most redundancy for the server platform in the event of outages that could affect game availability. This should be an important consideration, because if the server platform experiences an outage for any reason, Draw It or Lose It will become unavailable to players.

Delivering assets and browser-based CSS or JavaScript via the REST API will allow the server backend to be agnostic to client platforms, supporting the distributed multi-platform support The Gaming Room desires.

1. **Security**: User credentials should be stored in a secure database with a separate interface to allow creation of new user accounts by arbitrary players. Additionally, The Gaming Room’s administrators will need privileged access to the database and a separate interface to manage existing accounts. Administrator accounts should require multi-factor authentication using a hardware token to limit the risk of compromise. The server platform should be protected by a firewall and receive regular security updates.

Both the server application and the database should be hosted together to limit the need for queries to transit the network. Role-based authentication should be conducted both at the API endpoints as well as on all logic internal to the server application to enforce the principle of least privilege. The API should be protected with TLS encryption and requests using outdated TLS (anything less than TLS 1.3 as of this writing) should be prohibited.