

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_Toc94723243)

[Table of Contents 2](#_Toc94723244)

[Document Revision History 2](#_Toc94723245)

[Executive Summary 3](#_Toc94723246)

[Design Constraints 3](#_Toc94723247)

[System Architecture View 3](#_Toc94723248)

[Domain Model 3](#_Toc94723249)

[Evaluation 4](#_Toc94723250)

[Recommendations 7](#_Toc94723251)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | Jan 22, 2022 | Brandon Hobbs | First Draft |
| 1.1 | Feb 5, 2022 | Brandon Hobbs | Second Draft |
| 1.2 | Feb 14, 2022 | Brandon Hobbs | Third Draft |

## [Executive Summary](#_sbfa50wo7nsh)

CTS was asked by The Gaming Room to design a web-based version their Android-based game Draw It or Lose It. The game should allow multiple teams with multiple players per team. Each game-instance, team, or player should only occur once.

A singleton creation pattern has been adopted for object creation to prevent multiple game instances and an iterator pattern will prevent conflicting teams and team members.

## [Design Constraints](#_2et92p0)

The Gaming Room has an existing Android-based deployment of the Draw It or Lose It. CTS has been asked to extend this to the web and thus the tech stack needs to be compatible with web-deployment - Java has been selected for this purpose. Java being the native Android SDK language should ease this new deployment effort.

Any existing APIs serving the Android platform need to be reviewed or extended for mobile usage.

## [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 for the proposed design is shown below.

**"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 application consists of a main driver class that will be used to initiate the creation of the games, teams, and players. The actual creation is through the GameService class and follows a singleton design pattern so that only a single GameService class may exist at any time in memory.

GameService blocks its creation of any instances by setting its constructor to private. The only way to instantiate a GameService is through the getInstance() method. getInstance() checks whether GameService has been started and only starts if it is not present in memory.

Once GameService is running, the driver class is able to call the method addGame(). addGame() uses the iterator pattern to prevent similarly named Game objects from being created. This new Game object is then added to the List *games*.

After the game is created a team may be added to the game with the addTeam() method. addTeam() uses the iterator pattern to prevent similarly named Team object from being added to the game. This new Team object is then added to the List *teams*.

After the team is created a player may be added to the team with the addPlayer() method. addPlayer() uses the iterator pattern to prevent similarly named Player objects from being added to the team. This new Player object is then added to the List *players*.

Game, Team, and Player classes are all subclasses of Entity. Entity has 2 protected attributes: *id* and *name*. The default constructor is also protected so null objects are blocked at creation and only the overloaded constructors may be used.

The designed UML shows multiple Object-oriented program techniques. Polymorphism and inheritance are used in the extension of the Entity class and the overloading of constructors. Encapsulation and Abstraction are also apparent in the techniques used to add teams. A Team object may not be created directly since accessing the constructor is blocked but the user is still able to with the addTeam() method – without the user knowing how that team was added.

## [Evaluation](#_2o15spng8stw)

There are a few potential targets for development, e.g., Macintosh, Windows, Linux, mobile platforms, for either hosting/serving or acting as the client. The following table discusses the strengths and weakness of each platform.

It should b pointed out that the server and client choices are not linked. That is, if Linux was chosen as the server OS windows may still be the preferred option for the client.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | OS X Server is available for Mac but unless the client wants to purchase their own hardware finding hosts will be difficult and expensive.  OS X Server is $499 USD for 10-clients. Or, $999 for unlimited. | Linux is the most popular webhosting OS. Because Linux is opensource, maintenance and license costs tend to be cheaper than closed OSes like Windows.  Moreover, typical cloud providers like Google and Amazon offer Linux preferentially over Windows. | Windows servers are nice because they are GUI based and many applications used in the office will also run on the server – so familiarity is abundant.  License costs, typically per user, tend to be very high – especially compared to Linux.  Windows server licenses range from $6,200 (up to 16 core licenses) to $500 (up to 50 clients) per installation per year.  Hosting platforms may be more limited as compared to Linux. | Mobile devices can be used as a personal webserver or file server, but they are not equipped for multi-user serving.  The hardware is typically more limited, e.g., RAM, and they are not scalable like blade servers.  Costs is unknown as the hosting tools would probably need to be designed and built in-house. |
| **Client Side** | To develop for Macs, you need a Mac computer running the latest version of XCode.  Moreover, the macOS SDK is in Objective-C or SWIFT which are lesser-known languages.  Lastly, Windows usage is 75% of the market vs. macOS’s 16%. This presents a smaller market opportunity. | Development in Linux should be straightforward as Java or C/C++ or Python could be the language of choice – which are all commonly used.   Moreover, multi-users support is available on the GNU/Linux platform. GNU/Linux development might have little value as there is no widespread use. | Windows is typically developed using C# or .NET which are both common. There would be no barrier to entry to development of a Windows client application.  Windows has been a native multi-user platform since Windows XP.  Windows is the preferred OS for 75% of computer users which makes for a better business case. | Mobile devices are not designed to be multi-user. However, design a client application for Android or iOS is straightforward.  Android SDK is Java based so code developed for Windows and Linux might be able to act as a jumping off point.  iOS is SWIFT based so the same requirements for Mac apply, including the hardware needs. |
| **Development Tools** | Mac use Objective-C and SWIFT for development languages.  XCode is the common IDE used for Mac development.  XCode is listed as $99 USD per year per developer. | Linux development may take the form of C/C++, Java, or Python.  Python IDEs are often free, e.g., NotePad++. PyCharm is another popular Python IDE.  C/C++ IDEs are numerous – but not all are available for Linux. Eclipse can do all of these and is free. | Windows is primarily developed using C# and primarily .NET.  Microsoft’s Visual Studio is an immensely popular IDE and offers many plugins and integration options, e.g., Jenkins, TestComplete, etc.  Visual Studio ranges from $45 – $250 USD per user, depending on features, per year. | Android SDK is Java based and the most widely used Android IDE is Android Studio which is developed by Google as the official development tool. Android Studio is free to download.  iOS’s Objective-C and SWIFT languages are almost exclusively developed in XCode.  XCode is listed as $99 USD per year per developer. |

## 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**: Regardless of the frontend software chosen using Linux-based servers should be the preferred solution. Linux servers reduces the license costs and does not limit access to data centers the same way that Windows servers can. The frontend may be agnostic to the backend and connect via APIs.  
     
   Linux offers good security and operability. Since it is the most common server platform there are many tools available including security software.  
     
   Since the front end will be agnostic it may be written in the preferred language for that platform, e.g., SWIFT for iOS, Java for Android, .NET for Windows.
2. **Operating Systems Architectures**: The suggested architecture would be for a backend server that manages the game environment and frontend rendering. Because Draw It or Lose It gameplay is not dependent on twitch reaction it is not imperative that there be low latency between the frontend and backend – any transmission could be done asynchronously.  
     
   A more modern backend running containerized microservices with Kubernetes or Docker could allow for scalability. The cloud provider would need to be selected before an exact architecture was determined as most cloud providers have some flavor of proprietary tooling.   
     
   Choosing to use the frontend for rendering allows the server to offload some of the more resource intense parts of the application which would reduce the monthly data center costs. Moreover, client-side rendering should also insulate gameplay from network issues as framerate is important to gameplay. The client could cache some number of subsequent images ahead of active gameplay ensuring a smooth rendering.  
     
   Some amount of decisions are still needed on if the application will be browser-based or some sort of Java app when run on PCs or Mac. It might be easier to choose to do a browser-based game and enable it via PWA.
3. **Storage Management**: Unless the Game Room wants to purchase their own hardware no decision on storage medium, HDD vs SSD, needs to be made. Either HDDs or SSDs should provide the performance needs of the application – especially with some sort of caching behavior and client-side rendering.
4. **Memory Management**: Linux uses the concept of pagecache for data stored in main memory virtual memory for any pages allocated. Linx uses demand paging which allows for lower memory usage because ages not actively being used will not be loaded into memory. Page replacement is based on the Least Recently Used (LRU) algorithm.  
     
   The Android Runtime (ART) and Dalvik virtual machine use paging and memory-mapping (mmapping) to manage memory. This means that any memory an app modifies—whether by allocating new objects or touching mmapped pages—remains resident in RAM and cannot be paged out.  
     
   Memory management in iOS was initially non-ARC (Automatic Reference Counting), where we have to retain and release the objects. Now, it supports ARC and we don't have to retain and release the objects. Xcode takes care of the job automatically in compile time.
5. **Distributed Systems and Networks**: Uptime considerations and outage prevention are the reasons so many applications are being built in cloud native architectures. Many cloud providers are able to replicate and shift services amongst different deployments to prevent large scale outages.  
     
   The frontend and backend will communicate through RESTful APIs asynchronously. RESTful API usage allows the client/server communication to be transparent to the deployed frontend, e.g., Android, Windows, iOS.
6. **Security**: <Security is a must-have for the client. Explain how to protect user information on and between various platforms. Consider the user protection and security capabilities of the recommended operating platform.>