

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 1.0

## Table of Contents

[**CS 230 Project Software Design Template**](#_l6ti7uoag22u)1

[**Table of Contents**](#_30j0zll)2

[**Document Revision History**](#_grjogdjh5fi8)2

[**Executive Summary**](#_sbfa50wo7nsh)3

[**Design Constraints**](#_2et92p0)3

[**System Architecture View**](#_ilbxbyevv6b6)3

[**Domain Model**](#_8h2ehzxfam4o)3

[**Evaluation**](#_2o15spng8stw)3

[**Recommendations**](#_m8aleynsvzvc)5

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 5/19/2022 | Nicholas Ciesla | Initial document creation outlining Executive Summary, Design Constraints, and Domain Model. |
| 1.1 | 5/27/2022 | Nicholas Ciesla | Update of Design Constraints |
| 1.2 | 6/3/2022 | Nicholas Ciesla | Platform evaluation |
| 1.3 | 6/15/2022 | Nicholas Ciesla | Recommendations |

**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 is a video game company with an Android app, *Draw It or Lose It*, a game where a stock drawing is slowly rendered, and the players try to guess what is being drawn. The Gaming Room would like to create a web-based version of their game, and they need some help setting up the environment.

We are prepared to develop an early prototype of what the server side of *Draw It or Lose It* may look like. This is a Java application that consists of a **single** game service that can handle many games, games that can handle many teams, and teams that can handle many players. Each game will have a unique name and ID number. Within each game, every team will have a unique name and ID number. Within each team, every player will have a unique name and ID number.

## [Design Constraints](#_2et92p0)

* The Game Room’s servers must be able to handle the projected extra traffic or be upgraded to do so.
  + The move to the browser may result in increased traffic, especially with the popularity of these types of games with PC gamers. If the servers cannot seamlessly handle this new traffic, they may need to be updated.
* The Images for the game must be rendered quickly for the experience to be seamless.
  + In order for the players to be able to quickly submit their guess in order to score points, this means that the images must load quickly, a solution must be found to make this possible such as pre-rendering the images or file-size management.
* The Game Room’s servers must have adequate storage for the game images.
  + With such a large amount of image files, the storage of these images will be something to consider. Tying into the previous constraint, the drive speed may also need to be considered in order to get the images in front of the player as fast as possible.

## [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 GameService class is the class that governs the game structure itself. It is created with the Singleton Design pattern. This means that one and only one instance of the class can be created. This instance is owned by the class itself, and the constructor is private. While the instance is publicly available, this package can only have one GameService instance in memory at one time. This is essentially the highest level of encapsulation. All member variables are private and only accessible through methods and the class owns itself.

The Entity class is the base class for player organization. It contains an ID and a name (private), accessors and mutators for both, protecting the data structures themselves (encapsulation), and a toString method. The Game, Team, and Player classes inherit from this class. This means that they inherit the methods and member variables of the Entity class. They do, however, override the toString method from the Entity class. This is an example of polymorphism.

Because of the general simplicity and compartmentalization of the Entity family of classes, they could easily be used in a separate program. They are not overly complex and serve a very specific purpose. They are very portable in this way.

As discussed previously in this document, we have a bit of a nesting doll structure chain starting with the GameService class. The GameService has zero to many unique games, each Game has zero to many unique teams, and each Team has zero to many unique Players. The unique nature of the entities is achieved by using the Iterator Design pattern. Each time any of the add… functions are called, the list is checked for any duplicates, if one is found, the new entity is not created.

At first, it may seem unclear from this diagram how Game and Team generate IDs for the next layer. However, this is another benefit of the Singleton Design pattern. Because the singleton design pattern uses static variables and methods more often. The Game and Team classes can access the getNext[Team/Player]Id methods from the GameService class without having an instance because the variables holding the values needed are static.

**"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.

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Mac OS Server has recently discontinued, but older versions are still available. Mac OS is more comparable to Linux when it comes to security, but lack of support may lead to security issues. Like Windows, it is very easy to use, possibly easier than Windows with its own user interface. | Linux is a free, open source, and highly modular OS. Linux is very secure and has great support. However, it has a steep learning curve. That steep learning curve is mostly since it is controlled through the command line and the sheer number of things it is capable of. | Windows Server is a paid license OS by Microsoft. Compared to Linux, Windows is not very secure. However, Windows is much more user friendly as it can be controlled with a user interface. However, it is more limited in its capability. Costs range from $600-$4000 | Mobile devices are not suited for web hosting. It is theoretically possible with tools like Tiny Web Server for Android, but it would scale very poorly and generally be a waste of time. |
| **Client Side** | Because the Client application will be running in a web browser, the browser version of the game can be the same for all platforms. So long as cross-browser compatibility\* is in mind from the start, this will be a very time and cost-efficient process as all the tools needed will be open source and free to use. The browser commonly used on Mac is Safari, it is proprietary to Mac. There are many other third-party options as well. | Because the Client application will be running in a web browser, the browser version of the game can be the same for all platforms. So long as cross-browser compatibility\* is in mind from the start, this will be a very time and cost-efficient process as all the tools needed will be open source and free to use. There are enumerable browser options for Linux. Most third-party browsers are available alongside Linux specific options. | Because the Client application will be running in a web browser, the browser version of the game can be the same for all platforms. So long as cross-browser compatibility\* is in mind from the start, this will be a very time and cost-efficient process as all the tools needed will be open source and free to use. Most Windows users use third party browsers like Chrome, but older versions of Internet Explorer may cause some trouble. | Because the Android app already exists, the look and functionality of the iOS application could be replicated. However, both mobile apps would have to be modified to interface with the new server side to ensure cross platform compatibility. It will also cost The Gaming Room $99 per year to join Apple’s Development Program. |
| **Development Tools** | Developing the browser-based client leaves us with many options for development, to keep things simple, the app can be developed using HTML and JavaScript as this is generally the most used and well documented method. There are many IDE options available on Mac, the most popular we dev IDE, Microsoft’s Visual Studio Code is available and would be best to use because it is free and for its ease of use. VS Code’s modular nature will also let us install addons or plugins for anything that we may need. | Developing the browser-based client leaves us with many options for development, to keep things simple, the app can be developed using HTML and JavaScript as this is generally the most used and well documented method. There are many IDE options available on Linux, the most popular we dev IDE, Microsoft’s Visual Studio Code is available and would be best to use because it is free and for its ease of use. VS Code’s modular nature will also let us install addons or plugins for anything that we may need. | Developing the browser-based client leaves us with many options for development, to keep things simple, the app can be developed using HTML and JavaScript as this is generally the most used and well documented method. There are many IDE options available on Linux, the most popular we dev IDE, Microsoft’s Visual Studio Code is available and would be best to use because it is free and for its ease of use. VS Code’s modular nature will also let us install addons or plugins for anything that we may need. | Development of the iOS app would require the use of the Swift programming language to run natively on iOS devices. Although many windows and Linux IDE’s support Swift, it would be difficult to test your application on either platform. It would be best to develop the iOS app on Mac using their proprietary IDE XCode. XCode is included in Apple’s Developer program. The Gaming Room team may want to consider creating this app themselves as they have the most knowledge with their mobile app. It would be impractical to do any development on mobile itself as few tools are available and any that do exist will not have the features needed. |

**\*Cross browser development methods include declaring HTML Doctype, using CSS Reset rules like Normalize.css, using cross-browser compatible JavaScript frameworks like Bootstrap, and preforming cross-browser testing.**

## 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**:

When recommending a platform to host the servers for *Draw It or Lose It*, Linux is the clear answer. Linux allows for more customization of the system, has better security, and has more tools available for running this type of system.

1. **Operating Systems Architectures**:

Linux is a free, open source, and highly modular OS. These qualities will allow us to easily create an environment that it perfect for hosting and managing the server side of *Draw It or Lose It.* Learning to use Linux can seem daunting, but it has very dedicated and passionate community that has provided a wealth of information over the past decades. Linux also has a more diverse set of development tools than any other OS examined.

For the game system to be most secure, I recommend the use of the Singleton design pattern for the game service. As seen in the UML diagram above, the Game service manages its own instance. A separate instance cannot be created outside of that one instance. On the system communication side of things, this will follow the client server pattern.

1. **Storage Management**:

To retrieve the images in the fastest way possible, we should implement Contiguous Allocation as our storage method. This entails storing all pieces of each file or “bytes” end to end in the drive. To promote even more speed, we should store each set of images in a contiguous chunk on the drive, so we do not need to search for the next image in the set. The game files can be stored at the “top” or beginning of this drive to allow for fast startup.

This approach allows for something called “fragmentation” where empty spaces are left in the drive when files or sets of files are deleted. We can combat this by copying all game files to a separate disk and then loading them back to “compact” the files back into one contiguous chunk.

This storage method could even be managed by one drive on Linux using partitions. Linux’s built in Logical Volume Manager (depending on distro). Partitions are essentially separate pools of data that can help us to sequester parts of the drive to perform the compacting operation without the need for multiple drives.

1. **Memory Management**:

To allow the images to render as quickly as possible for *Draw It or Lose It,* the small initial number of images allows us to store every image required for the game in memory. However, if the number of images were to grow, it may be wise to switch to a system that brings in image sets as they are requested by a game instance.

Linux allows for much more control over all aspects of operation , this includes memory. Linux even allows for Contiguous Memory Allocation out of the box, so this would increase speeds over the other operating systems.

1. **Distributed Systems and Networks**:

To allow for all players to communicate with the server at once, we could use a RESTful API to facilitate this communication. This includes using an API like Maven or REST to allow the users to send HTTP requests to the serve, and the server to send HTTP responses. This is a very flexible system that can work on any modern browser. This can even allow the game to expand to consoles using things like the Xbox. Services API. With this method, the client and the server operate on different codebases on different machines, creating a distributed system. Because of the large difference between browsers and consoles, different responses may be required for them.

This method will network connectivity to work on both sides. If the server is down, the game is down; if the user is not connected to the internet, the game will not work for them. However, this is the nature of a distributed system and an online game.

1. **Security**:

The most logical way to protect user information would be to not store any, but this is not very appealing to modern sensibilities as it doesn’t allow for much expansion of the user experience. To protect user info on the client side, we require authentication. We could user usernames and passwords, but passwords can be procedurally guessed. Multifactor authentication via SMS would be our best bet client side.

Server side, Linux is one of the most secure operating systems because of its open-source nature and its robust permissions system. Most exploits are caught quickly and dealt with. To ensure further user security, passwords should be stored in a hash form instead of in plain text in the database. Additionally, the database can be password protected and logs can be kept as Linux records system events. An abnormal amount of login attempts can be reviewed and security issues can be dealt with quickly.