

<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 | <09/18/24> | <Jordan Purdy> | First draft |
| 1.1 | <10/04/24 | <Jordan Purdy> | Second draft |
| 1.2 | <10/14/24> | <Jordan Purdy> | Final Document |

**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 Client, The Gaming Room, has requested assistance streamlining development for their web-based gaming application. This application will be a multiplayer, server based, online application that will require multiple users to be able to access the same information and respond in real time. The application, and one instance of the game, must be able to be run on multiple machines at the same time and deliver the same information to all in a timely manner.

To do this, the client will be programmed on Java to allow multiple clients to run at the same time (multi-threading) and to allow for RPCs (remote procedure calls) to allow those multiple clients to access the same server. These RPCs will use processes called “pipes” to allow the Client’s users to read and write into the server file to answer these drawing questions at the same time.

To ensure that only one game instance will be active at once, a Singleton design should be used. This allows the program to create one instance of the game that cannot be copied.

To ensure that team names are unique, an Iterator design should be used. This allows the application to create unique teams and team names, then iterate through those names when the next teams are created.

## Requirements

* **Language Java**: To allow for use across multiple browsers and easily multi-thread and perform multiple client interactions.
* **Multiple devices**: To allow for testing across different devices, browsers, and user settings on those devices.
* **Multiple user testing**: To ensure that users with different bandwidth, service providers, devices, and browsers can all interact with each other across those systems without issue.

## [Design Constraints](#_2et92p0)

* **Server capacity:** The system will be required to use servers in some fashion in order to have a host server that communicates with all players. This can be a server hosted by the Client, or an individual player can be assigned as the current host of that server to save resources for the Client. If the client manages the server this will require more resources and upkeep but prevents gameplay issues where the current player host might not have an adequate system or network to host the current game server.
* **Browsers:** Being a web-based application, the application will have to be designed for and tested on multiple different browsers. Mozilla, IE, Chrome, Oracle, and many other browsers have differing settings and requirements when playing web-based games. Additionally, this will require the game to be run on any number of mobile devices that have access to web-based applications.
* **Flexibility:** Given that this is a multiplayer game, the application will have to remain flexible when players interact with the game. When new teams or players leave or join the game, the application will have to adapt to those new or removed players. This could mean deleting old teams to free up space or creating new threads when additional players join.
* **Picture library:** Depending on the host-server decision, the server will need constant access to the picture library at all times. It could be necessary to cache a small amount of pictures on a Client hosted server to ensure that if there is an interruption, the game client has access to at least some amount of pictures to ensure the game remains playable.

## [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.**

**Overview:** The diagram above contains all classes, methods, and variables that will be used for this application. The classes are Entity, Game, Team, Player, GameService, ProgramDriver, and SingletonTester.

**The ProgramDriver** initializes the game. This is essentially the main body of the program and works like the ignition for the program.

**The SingletonTester** is a way to ensure the Singleton design works, ensuring that there will genuinely only be one instance of GameService running at a time. The ProgramDriver and SingletonTester use each

other to test and initialize the program.

**GameService** is the primary class that creates and initializes the games, players, and teams in the program. GameService sets the constructor to private to ensure that the only way to instantiate it is through getInstance. This is an example of the Singleton design pattern and prevents multiple instances of GameService from being created. Gameservice contains the “getters” and “setters” for the games and the “getters” for the teams and players. This is done through the Iteration design method, ensuring that no players, games, or teams, will be duplicated.

**Entity** is the parent of the **Team**, **Player**, and **Game** subclasses, giving them a single location from which to inherit their important private attributes, such as id and name.

**Example:** When the driver class starts, GameService is initialized as a single instance through the Singleton design. Then, addGame() method will use the Iterator design to ensure no duplicate games are created before adding that game to the game list. Once the game is created, addTeam() may be used in a similar way to create a team once again using the Iterator design, to be then added to a team list. Once the team list is created, the addPlayer() method does the same as the last two, Iterating to ensure no duplicates, before adding the players to a player list.

**Object oriented techniques** used in this program include polymorphism and inheritance, when extending the 3 child classes from Entity and overloading the constructors within. Use of the private variables in Entity can also be considered encapsulation, or data-hiding, to protect the data from being overwritten elsewhere in the program. Design patterns used are Singleton, to prevent multiple instances of the GameService class, and Iteration, to sift through all already created games, teams, and players to ensure there are no duplicates.

## [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** | As of 2022, macOS no longer has OS X server, and requires the developer to find third party servers like Google or Identity for some server needs like User Account information.  However, many requirements like Caching Servers, File Sharing Servers, and Time Machine Servers are bundled with the newest installations of macOS | Linux is an open-source webhosting OS. It is free to develop with. Server-side, many options are available to use such as RedHat or Fedora, which use a variant of Linux to handle web services and databases. All server hosting requirements will be third party. However, most of them are free or cheap to license with. | Windows is a very user-friendly OS that has servers with GUI interfaces. Windows Server has a variety of built in features with programs like Azure, allowing a wide variety of features for development and hosting. Microsoft has years of experience developing and hosting servers, making this option very reliable.  However, Windows Servers tend to be the highest in cost, with licensing ranging from $500 to $6100, depending on server needs. | Mobile devices all have entirely different requirements depending on the manufacturer or OS provided within the device.  Mobile devices can self-host to some capacity, but for multi-user requirements, third party servers’ development through companies like AWS, Back4app, or Firebase will need to be used. Widely differing in cost depending on requirements. |
| **Client Side** | When developing for macOS, a computer with the latest version of Xcode (developer kit) is required.  Knowledge of languages like SWIFT, C#, and Kotlin will help speed the development process.  Currently, macOS is used by 20% of the market share, so it should be given considerably less resources and development time designated to it, due to a lack of return on time-value. | Development done with C or Java should be quick and straightforward. Python can also be used to speed the process if memory management is not as much of a concern.  While Linux is well-known for it’s usage in servers, websites, and databases, it does not have a very wide-spread client userbase, only being use on 6.28% of desktops. | Like Linux, Windows is also very straightforward, using languages like C# and dotNET to develop.  Windows has been used as a multi-user platform since 2001.  Windows usage is the highest of the most common operating systems at 72%, giving a much wider reach in audience and engagement. | Mobile devices are not intended for multi-user use. Depending on the device, a different operating system will need to be used for development, and a different language will need to be used.  Given that usage for both mobile phone brands is very high, and tend to be almost 50/50 depending on the country, it is very profitable to develop for both systems at the same time.  T there are many third-party frameworks like PhoneGap and AppCellerator that can develop for both platforms at the same time, but they will be more costly and give less overall control of the finer details of both systems. |
| **Development Tools** | Xcode is the primary IDE environment for macOS. It uses C# and SWIFT as its coding languages. It uses Xcode cloud to store data and builds, allowing for a remote work or outsourcing possibilities.  The current version of Xcode 16 is free to develop, $99 for individual developers to release on the App Store, and $299 for the Enterprise program for larger groups.  Additionally, Xcode Cloud has limits based on compute hours. Ranging from free to $400 per month to use. | Python, C, C++, and Java are the main languages used for developing in Linux.  Many IDEs are available, with the most common ones being VSCode and VIM. VSCode is free to use and tends to be very cost effective. Additionally, it supports many easy-to-use plugins for different languages or packages. | The most common languages used to develop with Windows are C++, C# and dotNET. However, many other languages can be used such as WINUI, React, and Python.  Microsoft Visual Studio is the IDE used for development and has a vast number of plugins for development.  Visual Studio is free for individual use, but has monthly subscription pricing ranging from $45 to $250 per user per month, depending on requirements and additional features. | For Android phones, Android SDK will be used. It primarily uses JAVA in the Android Studio IDE. It is free to download and use, with a one time $25 fee to create a developer account.  Apple phones will use iOS. Like Mac, iOS uses Xcode, which has the same cost requirements.  There are other, minor operating systems that exist like Lineage OS and Calyx OS, but they are used by a significant minority of mobile phone users and are usually not worth the development cost to create for. |

## 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**: The most cost-effective and multi-purpose operating platform to host the game server is going to be Linux, specifically Linux Ubuntu. Linux Ubuntu is capable of running on all major hardware platforms, with the additional ability to run virtual machines of other operating systems if need be. The front-end aspects of the application can be designed and coded in their native required language (Swift for iOS or c# for Windows), but the server aspect should be created and hosted in Ubuntu. The front end and backend can also be connected through APIs.

The OS is open-source and free to use, freeing up funding for more expensive requirements like cloud-storage. Additionally, Ubuntu is considered a highly secure environment, using kernel enhancements like AppArmor to limit security gaps of programs and resources.

1. **Operating Systems Architectures**: The Linux Ubuntu OS uses a variation of the Debian architecture, one of the oldest operating systems on the Linux kernel. The OS uses kernels that are responsible for directly interacting with hardware. The kernel is responsible for system memory management, software program management, hardware management, and filesystem management. The OS also uses real-time systems to reduce latency, making it useful for a game like this requiring accurate timing and rapid response from players.
2. **Storage Management**: Given the cost savings from using an open-source OS, the storage management should ideally be cloud based to ensure that only the needed amount of storage is being used for server-based information. Openstack, Kubernetes, or Ubuntu Server can be used for this.

Additionally, the application can cache some elements on the client-side to allow for things like user settings to be saved.

1. **Memory Management**: Linux uses many different techniques for memory management, such as on demand paging, anonymous paging, file system paging, swapping, and file systems caching. Linux breaks down memory into “pages” that is a chunk of memory that Linux can move, use, and story easily. If RAM is ever full, Linux will swap out some of these pages to the swap space on the hard drive, then when needed, it will swap them back into RAM. This allows for the most important aspects of the application to be quickly accessible. This is a useful concept when used in conjunction with the game and its need to quickly retrieve images and animations for multiple players running the game in a competitive state.
2. **Distributed Systems and Networks**: When using a cloud provider, there is less concern for things like outages and uptime. Many of these cloud providers run their entire business model on having backups and redundancies for issues just like this, to ensure their customers are always able to access their data.

For communication between front end and back end, RESTful APIs can be used to ensure that the back end linux servers can be connected to a variety of front-end clients using Android, Windows, or iOS. The back end provides the data, and the front-end requests and displays this data. There are various libraries like Retrofit and Volley for Android or URLSession for iOS that allow easy and efficient backend requests.

1. **Security**: There are many different ways to protect user data on Linux Ubuntu. Uncomplicated firewalls can be used to control access to the server, only allowing things like HTTP, HTTPS and SSH to pass through. Instead of using passwords, SSH keys can be used to provide authentication. Linux can also use a tool called fail2ban that blocks specific IPs after repeated failed authentication attempts. There are additional intrusion detection tools like Snort or Tripwire that can be utilized to enhance this security.

Data encryption should also be enforced by ensuring databases, like MySQL, are set with encrypted connections and are configured to enforce strong passwords. In transit, it is important to always use HTTPS for communication between front end and back end. An SSL certificate can be set up using companies like Let’s Encrypt, BuyPass or ACM.

Other basic methods of security can also be included like 2-factor authentication, JWT tokens, and OAuth tokens.

Additionally, it is important to minimize user data collection and only store the minimum amount of personal information needed to run the application and authenticate users. Any additional information gathered from the user is subject to loss if any security leaks do eventually happen.

Finally, ensure backups are created and encrypted in the event of a massive loss of data. Tools like Duplicity, AWS S3, or features included in your chosen cloud service can be used for this.