

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 3.0

## Table of Contents

[**CS 230 Project Software Design Template** 1](#_Toc111405088)

[Table of Contents 2](#_Toc111405089)

[Document Revision History 2](#_Toc111405090)

[Executive Summary 3](#_Toc111405091)

[Design Constraints 3](#_Toc111405092)

[System Architecture View 3](#_Toc111405093)

[Domain Model 3](#_Toc111405094)

[Evaluation 4](#_Toc111405095)

[Recommendations 7](#_Toc111405096)

## [Document Revision History](#_grjogdjh5fi8)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 07/14/2022 | Jonathan Boeglin | Filled in Cover, Exec Summary, Design Constraints, and Domain Model |
| 2.0 | 7/29/2022 | Jonathan Boeglin | Filled in Evaluation section |
| 3.0 | 8/14/2022 | Jonathan Boeglin | Filled in Recommendations section |

## [Executive Summary](#_sbfa50wo7nsh)

The Gaming Room wishes to develop a web-based version of their Android application, Draw It or Lose It. This web version needs to be able to pull stock images from a library and render them to completion after 30 seconds. The entire game lasts four rounds, one minute each, and needs to be uniquely named.

A web application that uses a Singleton Test to make sure only one instance of a game exists will be created as a solution. The program will check for unique team names as well.

## [Design Constraints](#_2et92p0)

* Each game and team must have unique names
* Needs to run on varying web services
* Must be able to adhere to game rules set by client (such as 30 second image render time)
* Implication: Lightweight app that meets client specs while being able to run and save multiple instances of itself.

## [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)

To start, the entire UML is contained in the “com.gamingroom” package. The “ProgramDriver” class is the main class that runs the entire program, using the “main()” method. This class also uses the class “SingletonTester” to check for unique instances of games. This is done with the “testSingleton()” method included in the class. The GameService class holds the necessary functions for the game to run and allows the program to check for existing games and teams. The Game, Team, and Player classes are all implement the Entity class, which demonstrate Inheritance. These classes also have a relationship of none to many, in order, meaning that even if none of one class exists, many of the next still can. Having these classes also demonstrates encapsulation, another main concept of object programming.

**"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)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | * Cost is moderate * Apple-only Hardware is used * Incredibly secure * Few commercial options * Can easily connect to iOS services as well | * Low cost * Hardware costs vary by user needs * Open source * Incredibly secure * Most commercial options available * Most reliable server software | * Medium cost * Lowest software requirements * Moderate security * Decent amount of commercial options * User-friendly, compared to Linux | * Low cost * Hosting is impractical beyond basic websites * Moderate to low security * No commercial options * Unintuitive process to set up and run |
| **Client Side** | * Cost: High (Mac machines and OS are costly) * Time: Medium (Mac interface is easier to use than Linux but not as supported as Windows) * Expertise: Medium (Not as easy as Windows but still more straightforward than Linux with decent resources)   The application needs to be tested regularly with all major browser platforms to ensure compatibility. | * Cost: Low (Free OS, only Hardware cost) * Time: Long (Linux harder to use and issues may be harder to find) * Expertise: High (Coding expertise needed just to make OS work normally)   The application needs to be tested regularly with all major browser platforms to ensure compatibility. | * Cost: Medium (Hardware and OS cost is mid-range for a decent machine) * Time: Short (Intuitive interface and well supported) * Expertise: Low (Large number of resources and support for Windows-based applications)   The application needs to be tested regularly with all major browser platforms to ensure compatibility. | * Cost: Varies (Depends on OS and phone type) * Time: Long (Flexible platform, but each device type needs specialized care) * Expertise: Varies (Depends on OS and phone type, but there are helpful resources)   The application needs to be tested regularly with all major mobile browser platforms to ensure compatibility. |
| **Development Tools** | Languages to use would include HTML, CSS, JavaScript, and Java. IDEs include Visual Studio, Xcode, CLion, and AppCode. Most require professional licenses to use. Depending on if the application implements different languages, more than one team may make the process smoother. | Languages to use would include Python, C++, C, Perl, and Java. IDEs include Eclipse, Atom, NetBeans, and IDLE. Most require professional licenses to use. Depending on if the application implements different languages, more than one team may make the process smoother. | Languages to use would include C++, JavaScript, Python, and Java. IDEs include Eclipse, NetBeans, Xcode, and Visual Studio. Most require professional licenses to use. Depending on if the application implements different languages, more than one team may make the process smoother. | Languages to use would include Swift, C++, Python, and Java. IDEs include Visual Studio, Xamarin, Koder, and CodeSnack. Most require professional licenses to use. Depending on if the application implements different languages, more than one team may make the process smoother. |

## 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**: A Linux-based platform is best, such as Ubuntu Server
2. **Operating Systems Architectures**: Major architectures supported by Ubuntu Server:
   1. x86-64: 64-bit processor that can handle large memory loads and fast processes
   2. ARM: lower power consumption, meant for mobile devices
   3. POWER: designed for server use, runs as well if not better than x86
3. **Storage Management**: Solid state storage has the fastest access rate. However, it is also the most expensive. With discs LOOK or C-LOOK scheduling should be used for requests and a combined scheme for allocation.
4. **Memory Management**:
   1. Swapping allows the platform to switch out which processes are in memory on the fly to take up less space. If a game isn’t currently being played or a team not being used, it is sent back to storage until requested.
   2. Paging keeps all processes in memory grouped in same sized chunks. That way they can be moved from primary to secondary memory, or the other direction, no matter which two chunks are being swapped. Any game, team or player can be swapped out with any other.
   3. Compaction moves unused memory into larger chunks to reduce fragmentation. Inactive pages are grouped together to make room for new ones. In the app, this means that teams who were not currently taking a turn would be compacted into an area with other inactive teams until they were changed out with the current active team.
   4. Segmentation breaks up the base code into varying sizes before insertion into memory. In the app, this means that while instances of the games, teams, and players would be paged, the main function and other non-created-object functions would be segments.
5. **Distributed Systems and Networks**:
   1. Connectivity: Using a server-based distribution, the user platforms only must connect to the server to communicate with each other. This can be done through connection to the company’s website from any web browser or a downloadable app that launches a shell program that basically does the same thing.
   2. Outages: Server-side outages would mean no users that connected to that specific server could stay connected. New users would be routed to a different server in the stack until the optimal one came back online. If the outage is on the user side, the connection would be broken until the user found another way to connect to their account. If the outage is from a network issue, then contacting the service provider for the company’s internet connection may be necessary.
   3. Solutions: Outage solutions include redundant power protection and generation, a reliable service provider, and memory allocated for users who can reconnect within a reasonable amount of time. Connectivity solutions include an easily accessible website, and a simple app dedicated to connecting the user to the game.
6. **Security**:
   1. Users: Two-factor authorization necessary, and reminders to change passwords if they are compromised should be displayed on the log-in page. Optional security questions or email verification for account recovery is also a good addition.
   2. Severs: Constant updates to software will patch vulnerabilities. Check accounts for correct access: over-privileged, lack of password, etc. Check for rootkits. Because of Linux’s natural security, anti-virus software is mostly unnecessary, and any issues with Ubuntu Server are currently covered by Canonical until 2032.