

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 | 05-21-2023 | Christopher Roelle | Creation and base requirements of the 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 Gaming Room would like to have a web-based game that can serve multiple platforms for their current game, Draw It or Lose It (currently only available on Android). The game involves images being rendered in set timeframes, where teams will compete to guess the puzzle based on these images (a la Win, Lose, Draw). If team A doesn’t guess within the given time, the other teams will have an allotted timeframe (and single guess) to try to solve the puzzle.

## Requirements

*Application should serve multiple platforms.*

*The game should allow for one or more teams to be involved.*

*Each team can have multiple players assigned to it.*

*Game and Team names should be unique and thus be identifiers.*

*Only one game instance in memory at a time. Use unique identifiers.*

## [Design Constraints](#_2et92p0)

Networking bandwidth and Server configuration. What is the expected player-base size versus the initial server/network configuration? What are the allotted resources we have to run game instances on?  
  
Age of hardware. What is the oldest hardware specification/model we should target to maximize player-base, but minimize extraneous support?  
  
Could game rules change? If rules could change (for seasonal or special events, or alternative game modes) we may want to investigate providing a means to handle this.

## [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 entire project is encapsulated within the com.gamingroom package.

All objects in the game (Games themselves, Teams, and players) would inherit from the Entity object. This allows us to inherit the base information/functionality that all objects should have in the game (id, name, getters/setters).

Each team object will consist of a list of player objects (none-to-many relationship) to keep track of who is on what team and can add players as well as display pertinent team information. Since a none-to-many relationship is employed, we can have an empty team, possibly due to instantiation, or all players leave the team.

Each Game would contain a list of Teams that are partaking in the current instance and can add teams to themselves when needed. This too employs a none-to-many relationship, allowing for a game to have no teams, possibly at the start of a session or end of a game.  
  
GameService will be what handles Games. It will follow a singleton pattern, to ensure we don’t start generating duplicate game instances, or other conflicts. The object keeps track of all game instances within a list and knows the next Player, Team, and Game ID available for instantiation and instantiate the new game sessions (with a check of course, to ensure uniqueness). This employs a none to many relationship on the Games, since there could be times where there are 0 active game sessions (outage, all matches end at the same time, system reset, etc).  
  
Program driver will run the game via the main() method.  
  
SingletonTester will check that the singleton pattern works as intended (Single GameService Object). The Program driver uses the functionality here to ensure that the GameService is the only instance of itself.

**"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 must 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** | Generally, MacOS is not used for hosting servers. It is possible, but is not an industry standard for WebHosting. | Well-known.  Industry Standard.  Takes some learning to deploy and manage to host a game service.  Open-Source, which can, in some cases make it safer, as patching can be done from a fork of the main branch.  Free. | Well-known. Industry Standard.  Can easily handle the implementation of a game service host.  Updates only when Microsoft releases a patch.  Comes at a higher cost. | Not well suited for hosting the application, in more than an ad-hoc sense due to hardware limitations. |
| **Client Side** | Mac hardware is generally the same with differences only being between a handful of variants per revision. If targeting the lowest rated Mac of a recent revision, most other Macs of the same or newer revision should be able to run the Game. | When targeting Linux, the different distributions as well as the hardware that may be running on the machine will need to be considered. What works on one distro may not work on another, and the same goes for the hardware being run. | With Windows, the OS will be less of an issue in this regard, since modern revisions can generally run the same software (and compatibility mode can resolve other issues).  Hardware will be the main hurdle here, as with Linux, Windows can be installed on vastly varying computer builds. | Android is already being handled here (the game already runs on it).  iOS, much like Mac will have generally similar hardware per revision (if not closer in spec per model in this regard, storage being the main difference between models). Targetting a base version of iOS/model would be the smart option since newer models would usually be able to run apps meant for older hardware. |
| **Development Tools** | Eclipse, Xcode (for iOS dev), vsCode (free) or Jetbrains would work for the IDE to develop Java applications for Mac.  If released to the Apple Store, we will need to purchase a license to distribute.  If the application will support GUI, we may need Adobe Suite (illustrator primarily) or a freeware alternative such as Inkscape to design art assets. | Eclipse, vsCode (free) or Jetbrains would work for the IDE to develop Java applications for Linux.  If the application will support GUI, we may need Adobe Suite (illustrator primarily) or a freeware alternative such as Inkscape to design art assets. | Eclipse, vsCode (free) or Jetbrains would work for the IDE to develop Java applications for Windows.  If released on the Windows Store, we will need to purchase a license to distribute.  If the application will support GUI, we may need Adobe Suite (illustrator primarily) or a freeware alternative such as Inkscape to design art assets. | Writing code probably wont happen as much on the mobile device.  Art assets could be made on a mobile device (preferably with a stylus).  May want to test the app on hardware, in this case, we would need the necessary cables to connect the device to the dev machine, and an associated IDE (Android Studio or Xcode) to bridge to the device. The device would also need Dev Mode enabled. |

## 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**:

Since Windows Server is the most user-friendly and versatile platform that is used in the industry, I highly recommend running this OS.

1. **Operating Systems Architectures**: The server’s OS should use a multi-tier architecture. This would allow for separation of the presentation, application and data of the application, which can lead to better scalability down the road as well as improved fault tolerance.
2. **Storage Management**: The server would need to house games, teams, and player data when not in use, this could all be stored via a database within tables, and would probably not need more than a couple GB to store information. This should also be managed in a RAID configuration to prevent data loss in the result of a system fault or power loss.
3. **Memory Management**: Windows provides virtual memory address space per process, which can allow for up to 4GB of memory viewing at a time (on 32-bit architectures) and 128TB on 64-Bit.  
   The OS also prevents threads from having access to memory addresses that are in use by another process. This greatly reduces faults and damage from memory reading/writing.
4. **Distributed Systems and Networks**: We would want to potentially have a backup server in case the main server is down for whatever reason. The general rule of thumb is to have it offsite (preferably stored quite a distance away), in case something happens to the location of Server A, Server B should be okay. RAID support will help in this regard as well. Having good routing and multiple network bridges can assist in keeping a consistent uptime for users. The additional backup server can be user to keep the system online during patching windows, where the system can then be swapped back to the main server when the backup is receiving the same patching.
5. **Security**: Encryption of data would be used regardless of OS, and would be a primary line of defense. The server should stay up-to-date to receive the latest security patches, and a widely used corporate Anti-virus like Carbon Black should be installed to prevent malicious attacks.  
     
   Since the service provided is a game, the app could provide a side-door entry to the server with minimal read/write access, while any machines needing direct access (dev machines, etc) should be MAC filtered at the least, to prevent unknown access. This will require more upfront security work to give access to the machines, but would greatly minimize the openings to data.