

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 | 01/26/25 | Randall Decker | Creation |
| 1.1 | 02/09/25 | Randall Decker | Filled empty sections |
| 1.2 | 02/23/25 | Randall Decker | Clean-up |

**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 team has their game idea, Draw It or Lose It, and are ready to get started developing it. The problem presented is there is no agreed upon environment to create a web-based version of the game. Information and comparisons will be made first and a recommendation will be made at the end.

## Requirements

As specified by The Gaming Room the following requirements are expected: each game has the ability to run with one or more teams, each team will have multiple players assigned to it, the names of lobbies and team names must be unique and known to players at the time of name creation, and only one instance of the game can exist in memory at any given time, achieved through a Singleton pattern design.

## [Design Constraints](#_2et92p0)

I will focus on the design constraints from going from one architecture to another and constraints because of the changes needed for storage and memory management.

Because the application is going from one architecture, mobile platform, to another architecture, client-server, there are a few differences on how each works:

* Mobile platform – Application information and is stored locally, game logic happens locally as well and the device only communicates with a server to update information, e.g., leaderboards and events
* Client-Server – Responsibilities are divided up between the client device, maintains the user interface, user interaction and rendering, and the server, which handles the game logic, game state and user data storage.

With the inclusion of client-server architecture there are new potential problems like constant network communication, the server has to handle game states and logic for multiple users, so it has to divide up resources, and establish a system to keep user data secure, e.g., user login.

Storage and memory management requirements change as well since on mobile devices everything needed to run the game is usually stored locally from game data to rendering. This means, in most cases, the application could be run offline with no network connection.

With client-server architecture users would only need to pull the bare minimum of data from the server storage to run the application. Data gotten from the server can be stored by *caching* the data locally to stop repeating downloads. Since clients need updates in real time, game state and game assets, storing data locally can help relieve the burden on the server.

Because a server has to handle multiple concurrent user requests, sync game states with clients and maintain data integrity there has to be efficient data structures and algorithms to optimize memory usage and provide responsiveness. Different types of data structures provide unique benefits such as:

* Arrays and Lists – useful for data such as leaderboards or game objects because it can be indexed
* Hash Tables(Maps) – useful for player profiles, inventories, and game states because multiple types of player data can be retrieved with a unique identifier

Algorithm types include:

* Caching – Store frequently used data, e.g., player profile and game state, in memory to reduce database load and increase responsiveness
* Delta Compression – allows you to synchronize the game state by only sending the changes made to the game state instead of the entire game state, this helps reduce bandwidth

## [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)

There are two sets of classes that relate to each other, first the ProgramDriver and SingletonTester classes.

The ProgramDriver class contains a main() method and has a uses dependency on SingletonTester class indicated by the <<uses>> arrow to SingletonTester class. SingletonTester class tests the Singleton design pattern as seen by the testSingleton() method. It is used by the ProgramDriver class.

The second set of classes are the Entity, GameService, Game, Team, and Player classes.

The Entity class is the superclass that provides common attributes to the Game, Team, and Player classes. This shows an example of the Inheritance principle because the Game, Team, and Player classes inherit the attributes *id*, and *name* and the methods *getId*(), *getName*(), and *toString*(). It also shows an example of the Encapsulation principle by creating the private attributes *id* and *name* and allows access to them through *getId()* and *getName()* method.

The GameService class manages the games with methods to add games or retrieve game states from existing games through a list called games. The GameService class has the following attributes, *games, nextGameId, nextPlayerId, nextTeamId,* and *service.* It has the following methods, *getInstance(), addGame(), getGame(), getGameCount(), getNextPlayerId(),* and *getNextTeamId*(). The GameService class uses the Singleton pattern so there is only one instance of GameService exists at one time in memory through the use of *getInstance*() method. The SingletonTester class tests to check this. GameService class has a one-to-many relationship with the Game class, 0..\*.

The Game class contains the list of *teams* and inherits from the Entity class. It has the following methods *addTeam*() and *toString()*. The Game class has a one-to-many relationship with the Team class, 0..\*.

The Team class contains the list of *players* and inherits from the Entity class. It has the following methods *addPlayer*() and *toString()*. The Team class has a one-to-many relationship with the Player class, 0..\*.

The Player class inherits from the Entity class and has only one method, *toString*().

The “has-a” relationship between the GameService, Game, Team, and Player classes demonstrates a Composition principle. The GameService class contains a list of Game objects, the Game class contains a list of Team objects, and the Team class contains a list of Player objects.

Lastly, there is a Polymorphism principle demonstrated by the Game, Team and Player classes. Each of these classes has their own *toString*() method to override the Entity class’s method *toString*().

The domain model satisfies the following requirements for the game:

1. **A game will have one or more teams involved** – The Game class contains a list of Team objects, the *addTeam*() method in the Game class allows teams to be added, and the relationship between the Game and Team classes is one-to-many. Because the Game class has a list of Team objects it demonstrates the Composition principle as well as allowing iteration through the list of teams for easy management. All this together satisfies the first requirement
2. **Each team will have multiple players assigned to it** – The Team class contains a list of Player objects through the *addPlayer*() method. The relationship between the Team and Player classes is also one-to-many indicating another Composition principle. A list of players means the class can iterate over the player list showing an iterator pattern and all together satisfying both requirements 1 and 2.
3. **The game, team, and player must have unique IDs and names, and users must be able to check to see if a name is in use when choosing a team name –** The Entity class is the base superclass for the Game, Team, and Player classes providing methods to access common attributes because the attributes, *id* and *name*, in the Entity class are private. Using the Encapsulation principle in the Entity class to make private attributes ensures controlled access to the data. By inheriting from the Entity class the Game, Team, and Player classes will all have unique IDs and names. The GameService class manages the creation of games, teams, and players to guarantee unique IDs with the *nextGameId(), nextTeamId()*, and *nextPlayerId()* methods. The GameService class has an iterator in the *addGame*() method as well to ensure unique game IDs. All the information so far satisfies requirement 3.
4. **Only one game instance can exist in memory at any given time –** The GameService class uses the Singlton pattern through a private contructor, *-GameService()*, a static instance variable, *service: GameService,* and a public method, *getInstance()*, to return a single instance of GameService. Because the GameService class manages all game instances this means only one instance can be in memory at a time. The Singleton pattern, and the inner workings of it, satisfies requirement 4.

**"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.

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** | MacOS offers security, a user-friendly interface, and a stable OS. It does not scale into high traffic environments well, it’s more expensive than Linux, and potential software compatibility issues. | Linux is open-sourced and highly customizable. It is cheaper, free, easily scales up to meet higher traffic demands, and about as secure as macOS. The downside is it’s not user friendly, has a steep learning curve, potential software compatibility issues, and it relies on command line interface for server administration. | Windows is a popular choice for servers because of its user-friendly interface, has plenty of guides/help due to popularity, and compatibility with most software. Unfortunately, it can be less expensive than macOS but more expensive than Linux, more people use Windows, so security isn’t guaranteed, administration is more complex than macOS, and less customizable than Linux. | Using a mobile device like Android provides unique advantages and disadvantages. Since it is a mobile device, you can take it anywhere and access it anytime, it’s easy to set up and is good for small projects. Disadvantages being limited processing power, decrease in battery life, security issues, can’t scale, if removed from wifi range it has an unreliable connection. |
| **Client Side** | There is no cost, time, or expertise needed to have users run a web-based game on macOS since Java is compatible with it. | There is no cost, time, or expertise needed to have users run a web-based game on Linux since Java is compatible with it. | There is no cost, time, or expertise needed to have users run a web-based game on Windows since Java is compatible with it. | There is no cost, time, or expertise needed because there is already an Android version available. |
| **Development Tools** | The current development tools for Java on macOS are IntelliJ IDEA, Eclipse, and Netbeans. | The current development tools for Java on Linux are IntelliJ IDEA, Eclipse, BlueJ, Dr. Java, and Apache Netbeans. | The current development tools for Java on Windows are Eclipse, Netbeans, IntelliJ IDEA, BlueJ, and JDeveloper. | The current development tools for Java on Android are Android Studio, Netbeans, and Eclipse.  The current development tools for Java on iOS is primarily Xcode. |

## 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**: Windows is my middle ground recommended operating system. Java can run on any of the major operating systems and there are plenty of development tools on each but Windows is easy to learn and has plenty of documentation for training.
2. **Operating Systems Architectures**: The Windows Server network architecture is based on the OSI model, a security model with authentication protocols like Kerberos, NTLM, and Schannel to verify user authorization, a kernel mode to separate users from critical operating systems such as device drivers and processors. It also features Windows Server’s Hardware Abstraction Layer (HAL) to allow different hardware devices without custom drivers.
3. **Storage Management**: Windows Server comes with its’ own storage management system. It has a built-in interface to manage disks and partitions. It can create virtual disks with different levels of data redundancy to protect data. There is also compatibility with storage devices connected from outside the server. Lastly, there is a built-in resource manager to set restrictions, generate reports on storage usage, and more.
4. **Memory Management**: Windows Server manages memory through paging, division of memory into fixed sizes called pages and physical memory into similar sizes called page frames, virtual memory, allowing more addresses than the current physical memory, virtual address spaces, with each process having its own address space.
5. **Distributed Systems and Networks**: Distributed system refers to multiple servers workingtogether and sharing resources across a network to improve performance, create fault tolerance, and allow for scalability. With multiple servers if one fails the rest of the group can take over its tasks through failover clustering, also multiple servers’ network traffic can be split up and sent across multiple servers to improve performance with Network Load Balancing (NLB) and allow data access from multiple servers as if it were one big server with a Distributed File System (DFS).
6. **Security**: Windows Server has a multi-layered protection system built in with features like Windows Firewall, antivirus protection (Windows Defender), user account management with set permissions and resources, and Credential Guard to separate sensitive user data and isolate it so it can’t be accessed.