

Drawing Room

# **CS 230 Project Software Design Template**

Version 1.0

By:

John Miller

## 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 | 03/24/2024 | John Miller | Version 1.0: Summary, Requirements, Design Constraints, Domain Model |
| Version | Date | Author | Comments |
| 2.0 | 04/07/2024 | John Miller | Version 2.0: Evaluation |
| 3.0 | 04/21/2024 | John Miller | Version 3.0: 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 proposed solution is to design a web-based Draw It or Lose It by The Game Room. The program will use object-orientated programming principles to deliver a scalable, secure, and modular application. By leveraging a distributed architecture, the solution will be able to withstand many users. Encapsulating various classes will enhance program stability and mitigate typical computational errors.

## Requirements

* Web-based platform
* The ability to have many games, each with its own unique name and ID.
* The ability to have many teams, each with its own unique name and ID.
* The ability to have many players, each with their own unique name and ID.

## [Design Constraints](#_2et92p0)

Since the application will be on a web-based platform, it will have to be scalable to accommodate many concurrent users. To accomplish this, a modular design with special attention to class relations and cardinality is required. Using methods such as Singleton will ensure that, at most, one GameService is created, helping keep latency within acceptable ranges. Data management will also have to be considered so that data is saved or recoverable in the event of a crash. Therefore, the Game State should be saved in a distributed network, helping to alleviate these concerns. The code should also be able to run on various browsers to reach the widest possible audience. Laws and regulations must also be addressed depending on the age range and users' location.

## [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 UML diagram demonstrates the four principles of object-orientated programming. Inheritance is shown through the Entity class, the parent or superclass of the child classes Game, Team, and Player. The child classes all have a one-to-one relationship with the Parent Class Entity. The default constructor shows this for the Entity class being private. The Entity class stores some functionality and the data members’ ID and name for the subclasses. This demonstrates encapsulation. The code displayed also uses polymorphism, with certain functions returning different objects, allowing function chaining. Polymorphism is also shown with certain classes having multiple constructors and various classes having the same function changing the program's behavior based on which object is calling the function.

**"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** | **Strengths**:  Apple Ecosystem Integration  Known for strong hardware.  Strong developer catalog  **Weaknesses**:  Cost  Limited cross-platform compatibility  Scalability | **Strengths**:  Open-source  Scalable  Reliable  Strong Compatibility with other Unix-based systems such as macOS  **Weaknesses**:  Steep learning curve  Limited commercial support | **Strengths**:  Scalable  Strong commercial support  Variety of server software  **Weaknesses**:  Licensing costs can vary and may be expensive.  Limited compatibility and performance compared to Unix based systems. | **Strengths**:  Connectivity- most if not all have built in Wi-fi, cellular, and Bluetooth.  Portable  **Weaknesses**:  Processing power  Hardware concerns as they are typically not meant to run 24/7 |
| **Client Side** | Initial Cost will be high as various Apple Hardware will be necessary for testing. There is also an annual Apple Developer Fee.  Must include ample time to ensure game works well with Apple’s unique interface. (magic Mouse, trackpad, magic keyboard etc.) Also to meet Apple Developer requirements to be included on the Mac App store.  Developers should be proficient in Swift to have the application run natively. Also should have familiarity with Apples authentication such as face or touch ID. | Linux is open-source so initial cost will be low.  Extensive testing is required for compatibility across multiple versions.  Developers proficient in C++ or C# will be necessary. Must have a profound understanding of Linux data synchronization to ensure smooth cross platform play. | Paid and opensource materials are available for development.  Time for development can vary depending on the complexity of the game. Must be tested on a variety of hardware and versions to ensure compatibility.  Developers should be proficient in C++ or C#. | Upfront cost will include licensing fees to be available in App and Play stores.  There will need extensive testing time dedicated across multiple devices and multiple OS and versions.  Developers well versed in Swift for iOS and Java for Android are required. They should also have a strong understanding of mobile device UI such as touch screens. |
| **Development Tools** | Apple devices all run the programming language Swift natively. To develop an app that excels on Apple devices this language is recommended.  Apple’s flagship IDE XCode and Apple Developers tools are recommended. | C++ is the recommended language for developing game software on Linux. This is due to performance and compatibility with a variety of game engines.  Rust is another language with growing popularity due to enhance performance and security features.  Visual Studio Code is a Microsoft developed IDE with plug-ins for developing in Rust or C++.  A Library for creating multimedia software will also be necessary. | C++ is also widely used for game development on Windows machines.  The full version of Visual Studio is recommended for development on Windows machines due to its high level of integration.  A library to create multimedia objects will also be necessary. | Swift is Recommended for iOS devices.  Android devices are able to run the JVM so Java is recommended there.  If the goal is to save time and money at the cost of some integration platforms such as React Native allow cross-platform development written in JavaScript and then translated to run natively on devices.  The app will need the ability to interact with a phone’s limited hardware screen size and touch screen capabilities. |

## Recommendations

Analyze the characteristics and techniques specific to various systems architectures and make a recommendation for The Gaming Room. Specifically, address the following:

1. **Operating Platform**: I recommend using a cloud-based service to develop the game's server side. Cloud services such as Microsoft’s Azure or Amazon’s AWS offer a pay-per-use model that efficiently scales the platform as more users join. These services efficiently store and provide appropriate access to the necessary data. Specific coding languages and IDEs for back-end development can be left to the developers' discretion. However, the front end should run natively on each device. C++ is recommended for Linux and Windows, Swift is recommended for Mac and iPhone, and Java is recommended for Android devices.
2. **Operating Systems Architectures**: Cloud services are excellent for building a client-server architecture. They also allow a Micro-service architecture, which operates similarly to a client-server architecture but allows different features to be decoupled or run independently. This modularization of services enables each feature to be implemented in its own language and interact with an API, such as REST, to direct and authenticate requests. Cloud services take responsibility for storing and organizing information, allowing developers to ignore many low-level programming needs and focus on high-level functionality.
3. **Storage Management**: Cloud services automatically manage how data is stored. Therefore, developers will not have to worry about how to store data specifically on servers, such as allocating memory blocks. The Microservice architecture also modularizes information storage, allowing each feature to have individualized storage, decreasing the likelihood of memory leaks.
4. **Memory Management:** Memory use should be limited to ensure consistent gameplay among less powerful devices such as smartphones. Developers must use memory well to establish smooth gameplay across different devices. Images to be rendered must be cached into memory to render smoothly without losing quality. Techniques such as memory paging make this possible and efficient. Memory will also capture the current game state and player-specific information such as name and team. A singleton pattern must be included so that no more than one game state can be included per device.
5. **Distributed Systems and Networks**: An API such as REST is essential to establish effective communication between the distributed systems. This allows the program to be developed in several languages and has one tool to direct the system calls for different functionality. This enables the client-side development to be platform-specific, but all interact with the same server infrastructure, reducing the need for redundant code. The RESTful API can ensure secure communication between users and the server and allow game-state synchronization between multiple different devices and OS.
6. **Security**: The game should also implement Role-Based Access Control (RBAC) following the principle of least privilege to protect secure data an intended user would not need to access. An authenticator can be built into an API referencing an encrypted database; cloud-based services often handle data encryption to ensure secure access to users and administrators. An iterator pattern will search the database for a username and password and allow access based on the role upon a match.