

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 | 05/22/2024 | Zach Harland | Added: Executive Summary, Requirements, Design Constraints and Domain Model description |

**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 needs a website based on their application that allows users to join teams and play the game online. The hosting of this game service and the management of the unique entities therein is the focus of this project.

## Requirements

The player should be able to create and manage their own name and team(s). These teams should also be able to be joined by other players. All user names and team names need to be unique and the existing names should be searchable in real-time to allow the players to choose valid (see unique) names.

There should only exist one instance of a game in memory at a time. Each game should also be able to be uniquely identified to avoid conflicts.

The game should match the original application in that there are four rounds, each lasting one minute, where in each round a clip-art picture is incrementally revealed over 30 seconds. If the first team cannot provide a correct answer, then a challenging team(s) would have 15 seconds to give a single guess and ‘steal’ the round.

The game process should be real-time allowing the drawing to be revealed at the same rate for every player as equally as possible. The name searches need to be quick as well, allowing for fast name choices.

The website should be both reliable and scalable with enough bandwidth, computationally and through networking, to allow for multiple games, teams, and players to play simultaneously.

A system should be designed to allow for the registration of unique names, either as a profile to be logged into or as a guest user.

Presumably this software and service will be monetized and the method by which that can be done should be decided on.

## [Design Constraints](#_2et92p0)

The logic of the game itself is largely going to be a copy of the current application’s logic but adapting that logic for integration into a website and hosting service may require extra handling and the exposition of points through an API or other pointers.

Ensuring uniqueness of the ‘entities’ (the games, teams, and players) is paramount to prevent confusion between entities. Enforcing the uniqueness of the entities will require indexing and searchability of the existing entities at the time of instantiation.

Because the game is reliant on timing and cooperation, the interactivity of the players needs to be as low latency, simple, and obvious as possible to aid in the enjoyment of the game.

The UI will be hosted on a website and that means a different stack of tools, services, and limitations than the game logic. It will also require separate hardware or infrastructure knowledge dependent upon the hardware the website is hosted on.

As the game grows in popularity, the stability and scalability of the platform should be considered during the development.

Compatibility through browsers is also a concern, but that is largely paired down to a couple of general platforms. Still, how a UI is dynamically scaled should be considered during its development.

The real risk of entering in names that are, or are based on, real identities and having those names in storage in a server and displayed publicly poses security issues for the users.

## [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 driver for the service is the ProgramDriver class. The ProgramDriver class contains the main method which instantiates the GameService and uses the SingletonTester.

The SingletonTester tests whether the GameService is currently instantiated and reports the number of games available to run with its testSingleton method.

The Entity class is a parent class with some basic attributes: id and name. This ensures that all child classes of Entity have these attributes. It also includes the default private constructor, the public constructor, the accessors, and an output method to return an entity as a String. Note there are no mutators as the Entity should be created with all the appropriate information when instantiated.

The Game class is a child class of Entity. Any number, or no number of games can belong to the GameService class. Each Game class can have 0 to many Team classes instantiated by and belonging to it. The Game class has an attribute that contains a list of all the Teams associated with it. It also has a public constructor that must be called with the id and name attributes as arguments. It also has a method to addTeams by name and a method to output the Game class as a string.

The Team class is a child class of Entity. It can have 0 to many Player classes associated with it. This list of Player classes is kept as a List in its attributes. It also has a public constructor that requires an id and name argument. There is also an addPlayer method to add Player classes to the list of Player classes in the attributes as well as a method to output the class as a String.

The Player class is a child class of Entity. It has a public constructor that uses the id and name attributes as arguments as well as a method to output the class to a String.

Finally, the GameService class can have zero to many Game classes associated with it. In its attributes it contains a list of the Game classes it has associated with it, an attribute for the nextGameId, the nextPlayerId, and the nextTeamId to track the games, teams and players available. It also contains a private constructor and Singleton pattern method to ensure only one service is running in memory at a time. It also has a method to add a Game class to its list of games in the attributes. It can also search the Game class list to find a game by id or name with two separate methods. It also has a method to count the number of games in its Game class list. Lastly, it has a method to get the next Player and Team id as identified in its attributes.

**"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** | Apple has discontinued its dedicated enterprise hardware and server operating system line so hosting the software may be an issue if only commercial hardware and operating systems are used. Some Macintosh hardware exists that could do the job ( e.g. Mac Pro 2023) but the implementation of this hardware is limited by cost and value. However, the OS is included in the cost of the hardware. The hosting software stack may not be free, however, and the implementation of the software on the ARM based hardware may involve containers running translation layers adding cycles to the application’s processing. | Linux is the most popular OS option for hosting websites. Most popular Linux distributions such as Fedora, Ubuntu Server, and OpenSUSE have server-oriented OS versions each with versions that promise extended support. Each of these distributions can integrate with popular software stacks for web hosting and, in some cases, are free for personal use. Enterprise software stacks do have associated costs, however, but most, if not all, of the most popular hosting software runs natively on Linux servers. Anything else can be run in containers using tools like Docker and Kubernetes. | While not the most popular choice for hosting a website, the Windows Server 2019 is still used for a large percentage of the rest of the websites and internal intranets. The software natively available and the ability to run containers and VMs make up for nearly everything else. While the server OS is available, it is not free, but the cost is nominal. However, nearly every option that is available as FOS for Linux can also be used in Windows, but again, for enterprise use, there may be a need to pay. | While it is technically possible to host a website on a mobile device, the amount of modification to the hardware and OS, sometimes down to the kernel, makes this option a nonstarter. |
| **Client Side** | The app as we are building can be run in a browser. Using even the stock browser will require a little research and debugging to ensure the HTML is parsed as we expect. If we are developing an application native to MacOS we can use a platform agnostic system and reduce the development time, but any idiosyncratic behavior of the MacOS regarding the application will need to be addressed and debugged. Someone with expertise in that area would reduce that greatly. | The app we are building can be run in a browser. There is no stock browser, per se, for Linux as a whole, but there are the usual suspects such as Chromium based browsers and Firefox. However, if we are considering building a native app, there is no need to double our work and using the same approach to the MacOS and Windows platforms is completely fine. Some Linux specific behaviors, and its clearly delineated permissions may change how data and files are managed. Debugging time will be the biggest addition to porting the app, if we continue to use a platform agnostic approach. | The app we are building can be run in a browser. Windows comes with its own browser, but being as it is Chromium based, this consideration will already be addressed in the development of the app server side. However, if we are developing a native app using our OS agnostic approach, then the only considerations that may extend the development time will be those that address the behaviors that are specific to Windows and its permissions. | The app we are building can be run in a browser. However, if we want to develop a native app for the platforms, we should consider some options available to deploy cross-platform. As each Android and iOS/iPadOS have their own specific requirements, the greatest challenge will be chasing down the edge cases where the cross-platform options do not meet our needs perfectly. This may add significant development time. |
| **Development Tools** | Our application is going to be run mostly on the server and interfacing with a browser. However, if we were to build a native application, we are not limited by a specific language and can use Java via the Eclipse IDE as a cross-platform solution. However, the basic function of the application and how it is set up may need significant adjustment between this and other OSes and the most modern Macs have ARM based chipsets that may change how the instructions are interpreted or may force the application to run in a container with a translation layer. There may be some merit in developing a native app via Xcode, the IDE supported by Apple, which uses the Swift language for application development. This same IDE is also used for iPadOS and iOS applications. As well, we can use a game engine such as Unity or Godot to develop game specific cross-platform applications. Regardless of the native application, someone familiar with Xcode and the idiosyncrasies of MacOS will likely be needed on-hand. And for all Apple applications, it is necessary to have Apple hardware to test on. | While our application will be able to be run from any modern browser, if we were to develop a native application for Linux, I would recommend against it. Few people use this OS as a desktop OS and those that do run Windows apps with a translation layer (like Proton or Wine) to great success. However, if we wish to use a Java codebase, we can deploy it to Linux without much trouble, as it is supported. However, we must ensure the dependencies we use are compatible with the Linux kernel. In this case, Eclipse or IntelliJ are both popular IDEs with OS localization plugins and options. Another option to build a game is to use a game engine such as Godot or Unity which can both be used to develop for Linux and other platforms. As development for Linux outside of the server space is limited, the dev team may find themselves chasing edge cases and bugs for this novel use case. | The most popular modern browsers are available on Windows and should be enough for the application as it is, but if we were to develop a native app Windows supports C# in their IDE Visual Studio for application development, and it also supports Java via Eclipse, Visual Studio, or IntelliJ. We can also choose to use a game engine, specifically Godot or Unity, to develop a game application that is able to be deployed to Windows, Linux and MacOS. | Most popular mobile browsers are able to render games and should be sufficient for our use-case, however, if we were to develop a cross-platform solution, we could use something like Kotlin Multiplatform to develop for both Android and iOS (iOS via Xcode). Also, we can use a game engine such as Godot or Unity (each configured through the respective SDKs) to develop with a game centric IDE. Because mobile development is significantly different from the other OSes, it is important to have someone available who knows the workflow for the submission of applications to the respective app stores. |

## 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**: I would recommend the Amazon Web Services (AWS) platform for hosting the application:

* Support for Java: AWS has comprehensive support for Java applications, with various services such as Amazon Elastic Beanstalk, AWS Lambda, and EC2.
* Economically Sound: The pay-as-you-go model and the free tier make AWS a cost-effective choice. For long-term usage, savings plans and reserved instances can reduce costs.
* Scalability: AWS provides auto-scaling, load balancing, and a wide range of managed services that can handle traffic spikes and ensure high availability.
* Java Ecosystem: AWS offers managed services like Amazon RDS for databases, AWS Lambda for serverless Java functions, and Elastic Beanstalk for easy deployment and management of Java applications.

1. **Operating Systems Architectures**: If we use AWS I would use this style architecture:
   * **Client:** Web browser/mobile web application
   * **Route 53 (DNS Server):** Domain registration and DNS routing to Amazon services
   * **Cloudfront/ELB/EC2:** This would handle the image delivery, load balancing and compute of the game service.
   * **Amazon S3/Elastic Beanstalk:** Having the simple storage will give our application and data a place to exist. Deploying Elastic Beanstalk will give us the ability to deploy Java applications in a scalable and reliable way
   * **RDS:** Using this service will allow us to integrate the SQL server to our game services.
2. **Storage Management**: Using this platform we would use Amazon RDS and Amazon S3
   * Amazon RDS: Provides a managed relational database service that supports various engines like MySQL, PostgreSQL, Oracle, and SQL Server.
   * Amazon S3: Used for object storage, ideal for storing and retrieving large amounts of data, including application logs, media files, and backups.

**Memory Management**: Cloudfront CDN may be the best choice for this as it can cache images at distributed servers allowing more reliable access to large image files. Just in time or Lazy Loading practices will be more effective this way.

**Distributed Systems and Networks**: Our distributed systems architecture should look something like this:

* + Microservices Architecture:
    - Components: The game’s backend is divided into smaller, independent services (microservices), each responsible for a specific functionality (e.g., user management, game logic, chat).
    - Communication: These services communicate via APIs, typically RESTful APIs. For real-time communication, WebSockets or message brokers (e.g., Apache Kafka, RabbitMQ) can be used.
  + Data Synchronization and Persistence:
    - Databases: Use relational (e.g., Amazon RDS) databases to store persistent data such as user profiles, game states, and logs.
    - Cache: Implement caching (e.g., Amazon ElastiCache) to store frequently accessed data, reducing database load and improving performance.
  + Service Discovery and Load Balancing:
    - Service Discovery: Tools like AWS App Mesh or Consul can be used to automatically detect and route traffic to available service instances.
    - Load Balancing: Use Elastic Load Balancing (ELB) to distribute incoming requests evenly across multiple instances, ensuring no single instance becomes a bottleneck.
  + Real-time Communication:
    - WebSockets: Enable real-time communication between the client and server for gameplay updates and interactions.
    - Message Brokers: Use message brokers (e.g., Amazon SQS, Apache Kafka) to handle real-time event processing and communication between microservices.

**Security**: Implementing security for the client could be implemented something like this:

### **Secure Authentication and Authorization**

### **Authentication**

### **OAuth 2.0 and OpenID Connect:** Use industry-standard protocols like OAuth 2.0 for authorization and OpenID Connect for authentication. AWS Cognito can be used to implement these protocols, providing secure user sign-up, sign-in, and access control.

### **Authorization**

### **Role-Based Access Control (RBAC):** Implement RBAC to ensure users have access only to the resources they need. Define roles and permissions using AWS IAM.

### **Least Privilege Principle:** Always grant the minimum necessary permissions to users and services.

### **Secure Data Transmission**

### **Encryption in Transit**

### **HTTPS:** Use HTTPS for all client-server communications to ensure data encryption in transit. Obtain SSL/TLS certificates from AWS Certificate Manager (ACM) and enforce HTTPS on your servers.

### **WebSocket Security:** If using WebSockets, ensure secure connections (wss://) for real-time communication.

### **Secure Data Storage**

### **Encryption at Rest**

### **Client-Side Storage:** Encrypt sensitive data stored on the client side using web storage APIs securely. Use libraries like WebCrypto for encryption.

### **Server-Side Storage:** Ensure that data stored in AWS services (e.g., S3, RDS, DynamoDB) is encrypted at rest using AWS Key Management Service (KMS).

### **Input Validation and Output Encoding**

### **Input Validation:** Validate all user inputs on both client and server sides to prevent injection attacks (e.g., SQL injection, XSS).

### **Output Encoding:** Encode outputs to prevent XSS attacks. Use libraries like OWASP Java Encoder for encoding HTML, JavaScript, and URL parameters.

### **Secure Session Management**

### **Session Tokens:** Use secure, signed, and encrypted tokens (e.g., JWT) for session management. AWS Cognito provides secure token handling.

### **Token Expiry:** Implement token expiration and refresh mechanisms to limit the lifespan of session tokens.

### **Secure Cookies:** Store session tokens in secure, HTTP-only cookies to protect them from client-side script access.

### **Content Security Policy (CSP)**

### **CSP Headers:** Implement CSP headers to prevent various types of attacks, such as XSS and data injection. Configure the policy to allow only trusted sources for scripts, styles, and other resources