

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 | 05/21/24 | Krissi Yan | Initial Commit |

**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 client requires a web-based game application that allows multiple teams to participate, with each team having multiple players assigned to it. The game must have a time limit for solving a puzzle, and if the team does not solve the puzzle before time expires, they will have an opportunity to offer one guess each to solve the puzzle within a 15-second time limit. Our task is to prepare a software design document and begin developing the game application, addressing their software requirements for their game server backend. We must help them create the backend for managing the game servers, teams, and players.

## Requirements

* Develop a web-based game application backend that allows multiple teams to participate, with each team having multiple players assigned to it.
* There can only be one game server, within which only unique games can exist and each of those have unique entries for teams and players.
* The game backend must be developed in Java.

## [Design Constraints](#_2et92p0)

* Ensuring scalability and performance to handle multiple users and teams simultaneously.
* Providing a user-friendly interface for players to interact with the game and each other.
* Implementing security measures to protect player data and prevent cheating or exploitation of vulnerabilities.
* Optimizing the application's load time and overall performance to ensure an enjoyable gaming experience.

## [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.

But here’s my answer anyway :D!

The system architecture view for this project includes a web-based distributed environment, with physical components consisting of servers, databases (probably), and network infrastructure. The UML class diagram provided represents the model of the game application, which consists of Entity classes that hold common attributes and behaviors – a common practice in game programming. Object-oriented programming principles demonstrated in the diagram include inheritance, encapsulation, and singleton designs which are used to fulfill the software requirements efficiently.

## [Domain Model](#_8h2ehzxfam4o)

The UML class diagram provided represents the Domain Model of the game application, consisting of Entity classes that hold common attributes and behaviors. The following entities are represented in the diagram:

* GameService: Represents a static ‘factory’ of games.
* Game: Represents a lobby of teams. Inherits from the Entity class.
* Team: Represents a group of players assigned to a puzzle. Inherits from the Entity class.
* Player: Represents an individual player participating in a team. Inherits from the Entity class.

**"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** | Pros: User-friendly, high performance, large community.  Cons: Not suitable for running backends usually. Almost zero rack-mounted solutions. | Pros: Ideal for servers. Slim, daemon-focused deployments available. Many rackable options.  Cons: Learning curve for ops-team ☺ | Pros: Commonly known with many development options.  Cons: Horrible stability when ran at scale over time. Very few CLI-only options. | Pros: It’s mobile.  Cons: Your server is mobile. Not enough power to expand the system. Not designed for hosting daemon services for client applications. |
| **Client Side** | Pros: User-friendly. Commonly used for light gaming. Many frameworks exist that will allow cross-platform support without code modifications  Cons: Expensive. Restrictive in certain regions. | Pros: The linux community will recognize you as their friend.  Cons: Nobody games on Linux (aside from steam decks). GPU Drivers and SDKs for games are spotty and unstable. | Pros: Huge gaming community. Common platform for this use-case. Many supported GPU drivers and SDKs for target development.  Cons: Windows sucks. | Pros: Huge gaming community. Very common platform for this use-case. Broad audience. Intuitive control schemes.  Cons: Low power, mobile internet connectivity, restrictive video capabilities. |
| **Development Tools** | Pros: Great third-party tools for development. When paired with deployment to Linux, it offers a Unix shell that often can work as a mini-dev environment.  Cons: Expensive. | Pros: Developing on the target platform results in faster testing and deployment.  Cons: Developers must know VIM on a robotic level. Sluggish UI (or lack there-of) for new users. | Pros: Great vendor-driven toolsets for development. Can deploy to multiple targets. Comfortable IDEs.  Cons: Windows sucks :D | Pros: None  Cons: All of them.  Don’t develop on mobile! |

## 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 recommend taking the code-base off Java and building in C++ or GO for targeting Linux containers to run on GCP or AWS Kubernetes. Since the code base is currently just a few files, the opportunity to use the proper setup should not be passed. The scale-out for the containers can be tied to an auto-scaler to consume more or less resources based on load by creating new replica-sets of the container.
2. **Operating Systems Architectures**: Linux will easily service the needs for this use-case. A container should be built that initializes the system and advertises its services to the load-balancer once it’s ready to accept data. The host Kubernetes infra will directly expose its resources to the cluster and allow for full performance with minimal OS-layer overhead.
3. **Storage Management**: Kubernetes persistent volumes should be used for any stateful data. The storage controller can vary depending on the deployment target, but any form will increase the flexibility of the system in comparison to a traditional VM or bare-metal deployment. NFS and other shared-storage drivers should be avoided to minimize dependencies.
4. **Memory Management**: The manifests for the replicasets or deployments for the build should use a burstable QOS pattern for Kubernetes using the limits and requirements key. This will allow the system to obtain any resources it initially needs, but allows it to exceed those caps if necessary. The horizontal pod auto-scaler should be used in place of allowing these pods to increase their resources to infinity.
5. **Distributed Systems and Networks**: By hosting the game servers in GCP or AWS, we can select multiple availability zones that will guarantee uptime for our customers. From this, we can use a simple REST API running on the TCP/IP stack to ingress data to our application and replicate it across zones if necessary (i.e. high scores or team names).
6. **Security**: TLS1.2 should be used for the REST API and proper third-party certs should be used to encrypt traffic to the API. Java Web Tokens can be used to create a form of expiring sessions for players logging into the game. These session tokens can be used to allow only authorized people access to the REST API for utilizing the game servers.