

The Gaming Room

# **CS 230 Project Software Design Template**

Version 3.0

## 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 | 05/21/2024 | Marcus Doucette | Filled out all required sections |
| 2.0 | 06/04/2024 | Marcus Doucette | Added extra considerations for Project 2 prompts |
| 3.0 | 06/18/2024 | Marcus Doucette | Final Review |

**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 problem we are trying to solve is how to get the existing Android application to be ported to a web-based player. They also want the software to maintain the following features of the Android version:

* One or more teams per game
* Teams will be composed of multiple players
* Each Team needs a unique name
* Only one game can be played at a time

## Requirements

* Port game to web-based client
* Maintain the following
  + One or more teams per game
  + Teams composed of multiple players
  + Each team needs a unique name
  + Only one game can be played at any given time
* The application will need to have both client and server components
  + The client will let the player see the game and perform their actions
  + Web-server will dispatch the web client when the user requests it from their browser
  + Game server will manage gameplay events to keep all clients on the same page and following the rules (this might be something that won't need changing to port from Android to web)

## [Design Constraints](#_2et92p0)

* As a web application, the client code needs to be sent to the client machine at runtime, meaning it should be lightweight, or if a loading time is necessary, we should make a loading page that keeps the user engaged while they wait
* Since this is a game, there should be minimal delay between an action the user takes and a response from the server.
* Security measures should also be considered to minimize the potential for cheating (only send the bits of the image as they are being shown to the players)
* Both servers will need to be multithreaded in order to handle simultaneous requests from many different users at the same time.
* If each instance of the server software is only able to run one instance of the game at a time, then we will probably want a manager to create and destroy instances of the server software dynamically to deal with varying levels of demand.

## [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 seven classes in the UML diagram. First, the ProgramDriver and SingletonTester are used for running and testing the program and will likely be replaced with different driver classes when the application is actually deployed to a server.

The next and most important class is the GameService. This is a singleton that will manage the different games or limit the number of games being run to 1, depending on how we want to set it up. It looks like this class will also be responsible for storing references to new teams and players before they actually join a game.

Game, Team, and Player all work together to define the structure of clients in a game. Each client will be represented as one player; the players will then be grouped into teams in order to interact with the game. The game then concerns itself with responding to the different actions that the teams take in order to properly play the game.

The last Class in the diagram is the entity class, which is the base class for game, team, and player. This helps make sure that each one of them has unique identifiers and provides a good place to add any hooks to all three subclasses, like a “ready” or “update” function that all classes can implement.

**"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** | The requirement of Mac to run on proprietary hardware makes this OS not a good choice for the server-side operating system. | Linux is definitely the most popular server side operating system, likely because of its customizability, which lets it be optimized for whatever use case the server host needs. | While Windows also have paid licenses like Mac, which makes it less appealing than something free like Linux, the hardware requirements are much more flexible, making it a reasonable option even if still leaning on the impractical side. | Android is technically just a version of Linux optimized for small devices like phones. While you could theoretically run it on a server, deploying software to it is a hassle. IOS is likely much the same. |
| **Client Side** | Mac computers can run browsers, which means they will be able to run our web app. | Linux computers can run browsers, which means they will be able to run our web app. | Windows computers can run browsers, which means they will be able to run our web app. | Mobile devices can run browsers, meaning they will be able to run our web app; however, they are less powerful, and there already is a native option for Android, so a link to that would mean we could be less strict about the client's performance. |
| **Development Tools** | To build a web app we will need to use JavaScript for the front end. There are many ideas that support JavaScript, but I think VSCode has the widest support. For a server app on Mac, we have more options, though JavaScript and Java are the most popular choices. Java development is most commonly done with the Eclipse ide. | To build a web app, we will need to use JavaScript for the client. VSCode is the most popular JavaScript ide. For the backend, JavaScript and Java are the most popular choices, though C++ is also popular for Linux systems. Eclipse is the most popular Java ide, and Visual Studio is the most popular ide for C++. | To build a web app, JavaScript is required for the client. VSCode is the most popular JavaScript ide. Backends can be written in anything, though, like with Linux and Mac, JavaScript and Java are the most popular options. Eclipse is the most popular Java ide. | JavaScript is the only option for building a web app client. VSCode is the most popular JavaScript ide. Mobile devices are much more limited in what languages can easily be used to develop for them. Kotlin and Java are supported by Android. iOS prefers Swift, but it can also run Java. Android Studio would be the choice for Android development. Xcode is the idea of choice for iOS development. |

## 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**: The web platform was the desired client target. Since the web needs a server backend, Linux will likely be the most cost-effective choice for both cloud computing and private infrastructure.
2. **Operating Systems Architectures**: Since the target is the web platform, any architecture should be able to run the client as long as there is a browser. For the server, the architecture will likely be the easiest to deploy at scale. Any operating system would let whatever code we write run on a variety of architectures under the hood, but Linux is a common choice because it’s extra flexible.
3. **Storage Management**: web clients store save data type stuff in the local storage object, which functionally is just the browser's own storage area on disk. This storage space probably won't be used much anyway, likely only storing things like user authentication tokens to help speed up logging on for returning players. The server will need to have hard drive space to store the pictures involved in the game as well as any metadata or logs that CTS would like to have stored for later. Access to these storage locations will be facilitated by the OS.
4. **Memory Management**: for the web client, the memory of the application is just extra memory the browser application will request. Since we don’t know the architecture the client will be running on, we should try to keep this minimal, but it’s not something that should be a problem. For the server, we will need a proportionate amount of memory for the number of games being run. For now, we have this solved by capping the number of games that can be played at once; however, in the future, if we want to host multiple games at once, we will need to consider how to request and deallocate more server memory dynamically as well as consider what the program should do if it starts running out of memory so it doesn’t disrupt the games of all the players at once.
5. **Distributed Systems and Networks**: Having the game server have a consistent interface among the different client platforms would mean that any client that knows how to connect can. To minimize clients that aren’t ours from running the game, we will probably want to have our clients have a means of validating themselves to help prevent the server from needing to handle the requests of bad actors.
6. **Security**: For this game, the best kind of security is to not need security. Avoiding the client sending any sensitive information where possible is important, but where such things are necessary, session validation and other measures like two-factor authentication can be invaluable. If we can keep this kind of sensitive information away from the channels, the next biggest concerns are fake clients trying to get something from the servers or fake servers trying to get something from the clients. The only thing the client might have that would be of use to a hacker would be some kind of exploit, which would likely be avoided by trying to keep the client’s job simpler, so there are fewer places where mistakes can be made. For the server, we can only really mitigate attacks because clients can come and go with significant irregularity. The validation of client software published by a trusted source, as mentioned above, would likely be the best deterrent against such attacks, but all of these kinds of measures are more minimizing risk rather than eliminating it.

**Other server-side considerations:**

- the web server and game server serve separate purposes (the web server will send a web page to a requesting browser while the game server will connect players and handle game logic), but they can be built with consideration for each other or even as multiple endpoints on one hostname server. All clients should be designed to make use of the game server, and any device should be able to access the web server, though whether or not the web client works will depend on which browsers we target.

- server hosting will undoubtedly incur fees based on how much traffic goes through. This will be the same regardless of if it’s done as one or two. Depending on the hosting service we would want to use, there could be pros and cons to both options. Having the client host the servers themselves will have a greater upfront cost with better returns over time, though with the added risk of having the servers not meet demand, this could be assuaged by having extra capacity be directed to a cloud server, though this would require extra engineering time. Other than that, some platforms like Android and IOS seem to have a surcharge when you post something to their storefronts. It's something to consider, but in the long run, probably not that much. Any other services like payment transactions or external API calls to something like Google might also have some marginal added costs, so which ones are being used or not should be decided on sooner rather than later

**Other client-side considerations.**

**-** The price of supporting multiple clients is once again a choice between many tradeoffs. Some languages like Java incur small performance hits in order to try and work on as many clients as possible, while other languages like C++ need either unique builds for each platform or a smart enough build system to be able to target the projects to different platforms dynamically. The web platform is often an appealing choice for many projects because other clients opt for it rather than needing to be targeted directly, but these kinds of apps require you to host a web server and often significantly limit the resources you can expect the application to run with. On top of that, you will still need to try to make sure the program functions on many different kinds of browsers.

**Other development tools considerations**

- while there can be more or less popular development environments, with the exception of actual build tools like maven, Gradle, cmake, or compilers like gnu or java, the development environment will do little for the actual functionality of the project. Different developers will prefer different tools, though all developers should be capable of most tools if they already know what they’re doing. While some IDEs have licensing fees, I think it’s ultimately unnecessary to pay for them without some tangible advantage coming along with it.

- for this project, multiple different programs will need to be written. Depending on how we want this to be structured, it could be reasonable to split the team up into a few, and there are even different reasonable choices on how to split it up. My preferred option would probably be to have one team focus on the game itself, worrying about rules, interface, and the compatibility of the interface with different devices, while another team focuses on the web services like the landing page for the game website and login credentials. While this makes both teams work on multiple programs, it does ensure that each team will be able to focus on different goals to improve the end product experience. A more common but less sexy option would be to have one development team work on the client while another works on the servers. This might be easier to describe to a non-technical worker; however, it leaves the door open to several cross-cutting concerns, which means that the planning phases become more important, and a lack of communication between the teams has the potential to become a serious roadblock if things start getting uncertain.