

Draw It or Lose It

# **CS 230 Project Software Design Template**

Version 2.0

## 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 | 03/16/2023 | Kyle Bauer | Completed requirements, constraints, and domain model |
| 2.0 | 03/30/2023 | Kyle Bauer | Added operating system evaluations and official recommendation for platform to use to begin development |
| 3.0 | 04/16/2023 | Kyle Bauer | Added additional recommendations and requirements analysis |

**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)

Our client, The Gaming Room, wishes to a develop a multi-platform web-based game that is based off their existing Android-only game. The game, which is called Draw It or Lose it, consists of teams of players competing to guess what “thing” is being drawn onto the screen. The Gaming Room is not familiar with the process of setting up a web game, so we have been contracted to formalize a software design document and begin developing the game in accordance with their requirements.

## Requirements

*<* Please note: While this section is not being assessed, it will support your outline of the design constraints below. *In your summary, identify each of the client’s business and technical requirements in a clear and concise manner.>*

## [Design Constraints](#_2et92p0)

* Web game must be publicly available worldwide at a static domain name. This may require redirecting the user to different top-level domain name for localization and compliance purposes.
* Web game must be able to support Microsoft Edge, Google Chrome, Mozilla Firefox, Opera, Safari, and Samsung Internet browsers in order to support a large user base.
* When players create game instances within the web game, they must choose a name. This will be unique to that specific game across the entire web game.
* Game rooms must be able to support an arbitrary number of players.
* Games can use free-for-all or team-based scoring. If team-based scoring is being used, teams must have an equal number of players.
* If team-based scoring is being used, teams must be given a unique name. This team name is unique across the entire web game.

<Identify the design constraints for developing the game application in a web-based distributed environment and explain the implications of the design constraints on application development.>

## [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 ProgramDriver class contains the main() function to be executed. This the entry point of the application. The ProgramDriver class also contains an instance of SingletonTester, which ensures that only one instance of GameService can be initialized.

Entity is the base object from which Game, Team, and Player inherit. This inheritance relationship allows the three subclasses to make use of the same functionality with it having to be duplicated across each class. Specifically, the id and name data members are available for use in Game, Team, and Player objects. These values are set in each constructor and allow the programmer to guarantee unique names and ids for each instance of each class, which is a requirement of the project. The Entity class also allows for retrieval of the id, name, and string representation of the object, which again are methods also available to three subclasses due to this inheritance relationship.

The GameService class is a singleton which is responsible for managing the games in the application. As a singleton, only one instance of this class can be in existence at a time. This is achieved by making the constructor to the object private and instead granting access to the class via the public getInstance() method. This will either return the service data member if it is not null, or initialize the service data member if it is null and return it afterwards. There are other methods in this class that expose behaviors such as adding games, retrieving games by name or id, and getting the next id numbers that will be assigned to games, players, and teams. Since GameService is a singleton, the data variables are all static so that they can be access at the class level instead of the object level. However, not all of the member functions are static. Finally, the GameService class has a 0 to many relationship to the Game class, meaning that a GameService can contain 0 or more instances of a Game.

Likewise, a Game object can contain 0 o more instances of a Team object, and a Team object can contain 0 or more instances of a Player object. This is an example of composition. While inheritance describes an “is-a” relationship (Game is an Entity), composition describes a “has-a” relationship (Game has a Team). This is another object-oriented programming technique that can be used to improve code efficiency by reducing repeating the same code in multiple places. This allows us to isolate behavior that can easily be reused across the program. Additionally, this implementation helps fulfill the requirements of the project. The project requirements state that a game should have the ability to have one more teams involved, and that each time will have multiple players assigned to it. By using composition, this can be accomplished.

The Game class contains the addTeam() method and the Team class contains the addPlayer() method. Although different implementations, they work in roughly the same way. They check the name provided to make sure that a team or player does not already exist with the provided name. If no match is found, they create the team or player. If a match is found, it simply returns the instance that was found. This how the program prevents creating multiple teams or players with the same name. However, this only applies for specific instances of the object. This means that two different teams can in fact have two different players with the same name, but the same team cannot have multiple players with the same name.

**"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)

| **Development Requirements** | **Mac** | **Linux** | **Windows** | **Mobile Devices** |
| --- | --- | --- | --- | --- |
| **Server Side** | Macs are popular systems for developers, so it might feel more natural working in a Mac server environment. However, Mac devices tend to be more expensive which may increase operating costs. | Linux operating systems are also common among developers and also very frequently used in server configurations. There are free, open-source distributions that can be easily installed on virtually any hardware. However, the level of customizability might make it hard to troubleshoot issues in certain cases. | Windows is a very popular operating system that virtually everyone has used at some point. It is owned by Microsoft and would be more expensive than a free Linux distribution. It can also be more prone to certain types of malware and viruses due to its popularity. However, Windows does offer a lot of support for a lot of different programs. | Android and iOS devices typically operate at a fraction of the power of Mac, Linux, and Window devices for an even higher cost. These devices are built for portability but not necessarily performance. They are not a common first choice in running a web server, and are really only appropriate for smaller or hobbyist projects. |
| **Client Side** | Safari is the top browser for MacOS devices. This means that special attention will have to be paid to make sure the client software does not use any features that are not supported by recent releases of the Safari browser.  The cost of supporting Safari should not be too expensive. I would estimate that given the framework for Firefox and Chrome, adjusting for the Safari browser should only be a week or two of work for a single developer.  Assuming a salary of $80,000 a year, the cost of this would be roughly $1540-$3080. | Chrome and Firefox tend to be the most popular browsers on Linux devices. This means focus should be carefully placed in these two areas, as the majority of users are likely to use one of these two browsers.  Focus on these two browsers is the highest priority, so it should be constantly developed for the lifetime of the project. Assuming two developers each at a salary of $80,000 and a six month development cycle, estimated cost would be roughly $80,000. | Chrome and Firefox are also very popular on Windows, but Edge cannot be ignored as it is the default browser supported by Microsoft. Focus can be put mainly on Chrome and Firefox, but attention must also be given to Edge to make sure no unsupported features are used.  The cost of Firefox and Chrome have already been determined. I would estimate the same timeframe for Edge as supporting Safari.  Assuming a salary of $80,000 a year for a single developer, and one to two weeks of work, the cost of this would be roughly $1540-$3080. | Safari is the default browser on iOS devices. Android devices tend to have different default browsers depending on the manufacturer. These need to be supported in order ensure the majority of users are supported.  The cost of Firefox, Chrome, and Safari have already been determined. I would estimate one to two weeks of additional work to fully support each additional Android browser. Assuming there will be three to support, we’ll assign one developer to each browser.  Assuming a salary of $80,000, estimated cost would be roughly $4620-$13,860 |
| **Development Tools** | Languages: Swift, C, C++, Python, Java  Tools: Xcode, CLion, VSCode, Eclipse, Visual Studio for Mac, git | Languages: C, C++, Python, Java  Tools: Vim, Emacs, Visual Studio Code, Eclipse, PyCharm, git | Languages: C#, C, C++, Python, Java  Tools: Visual Studio, VS Code, Eclipse, Code::Blocks, PyCharm, git | Languages: Java, Kotlin, Swift, React Native, Objective-C  Tools: Android Studio, Xcode, Visual Studio |

## 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 am recommending a Linux-based operating system, preferably one of the latest versions of Ubuntu. I believe this is the best choice because Linux environments are very common for software development, and there are a lot of open-source tools available for little to no cost. We will use the ngnix web server to serve our content, and the game will be developed using node for the server backend and React for the client frontend. Both of these see tremendous amounts of use in the web development world and are heavily supported by their communities.
2. **Operating Systems Architectures**: Ubuntu works with x86, AMD64, ARM, and PowerPC architectures. This should cover the majority of hardware currently available on the market. The official Ubuntu documentation goes on to state that Ubuntu uses a monolithic-type architecture, but does also make use of other types of architecture as needed. For instance, there is some use of microkernel-type architecture as well for system services.   
     
   In essence, this means that the Ubuntu operating system can be thought of as a single program that takes care of the majority of operating needs, such as file management, memory management, and device management.
3. **Storage Management**: Ubuntu has its own predetermined ways of handling storage. By default it uses ext4 for its filesystem, and I see no reason to override that option. It also provides partitioning, so we can logically separate areas of the disk as necessary for our needs.   
     
   In terms of the web game storage management, we will SQLite for data persistence. We will store information in a database and retrieve it as necessary. Any data which does not need to persist will not be saved into the database. Examples of data that is likely to be saved includes user information, user analytics, game metadata, and visitor history.  
     
   Should we need access to more disk space, we can easily upgrade to the next tier of hosting packages through Digital Ocean which will provide more disk space for an increased cost. To start, I recommend a General Purpose Droplet containing GiB of memory and 200 GiB of storage space for $272 monthly.
4. **Memory Management**: Ubuntu handles memory allocation and deallocation via its own specialized algorithms. Additionally, Ubuntu makes use of virtual memory techniques so that programs can hotswap their virtual memory into the actual RAM as the operating system switches from process to process. It also uses paging, which allows data that needs to be accessed more often to exist on pages within the RAM, while data that is not being accessed as often can return to disk until it is needed.   
     
   In terms of the web game memory management, we will be using Javascript for the node environment, which will be where the server logic lives. We will make use of appropriate data structures and containers to store information in active server memory. For example, we will need an object to keep track of players on the server, as well as some sort of container to hold a list of all the active players. We also need an object to keep track of games, as well as a container to keep track of all the active games.  
     
   Should we need access to more memory, we can easily upgrade to the next tier of hosting packages through Digital Ocean which will provide more resources for an increased cost.  
     
   When a player connects to the server, or when a new game is created, an object will be created in memory to represent the player or game. This object will then be stored in a list. Whenever the player disconnects or the game ends, an algorithm will be called to locate the object within the list and remove it. The Javascript language does have its own garbage collector, which is an automatic process that runs to free memory that the application code no longer has a usable reference to. This will alleviate some of the burden of fine-tuning memory management techniques at the cost of some runtime performance, but it should not have a notable effect.
5. **Distributed Systems and Networks**: The application should be hosted in a cloud environment, such as DigitalOcean. In this way, we can focus on delivering the best gaming experience for the end user without having to micromanage at the hardware level. The user’s web browser will connect to the DigitalOcean server via HTTP requests and information will be sent back in the same way. Once connected to the server, the node environment will execute the game application logic. Any server outages will be addressed by the Digital Ocean team and should be minimal.   
     
   After the user is connected to the server, information sent from the client to the server can then be sent from the server to other clients, thus allowing for communications between various clients and device hardware.  
     
   On the Digital Ocean droplet, we will use the ngnix web server to make our content accessible on the web. This will also come with the side effect of us having to manually configure nameservers for our domain name to point to the hosting package, as well as to manage any individual subdomain records.
6. **Security**: Although we will be hosting with Digital Ocean, security will still be our responsibility. We will implement proper authentication procedures using the Google Authentication platform. Additionally, we will handle the user login use case by sending a POST request from their browser to our server. Once successfully authenticated, the authentication token will be valid for one hour.  
     
   The site will have a valid SSL certificate issued to it. Our ngnix web server will have the proper firewalls and configurations in place to prevent unauthorized access to parts of the file system not needed for interaction with the web game.

Any user data that is stored should be encrypted for the protection of the users. Should a bad actor happen to gain access to the systems, encryption will add an extra layer of defense for everybody’s personal information. While encryption techniques cannot be guaranteed to prevent exposing sensitive information, it can be a very effective deterrent.